DEPENDENCY ANALYSIS AND LEARNING METHODS OF DECLARATIVE LANGUAGES

PhD Thesis

Author: Gyöngyi Szilágyi

Supervisor: Dr. Tibor Gyimóthy

Research Group on Artificial Intelligence, Hungarian Academy of Sciences

Contents

1 ACKNOWLEDGMENTS **1**

2 ABSTRACT **2**

3 INTRODUCTION **3**
3.1 Data Flow Analysis of Logic Programs 3
3.2 Data Flow Analysis of Constraint Logic Programs 4
3.3 Learning of Constraint Logic Programs 4
3.4 Learning Semantic Functions of Attribute Grammars 5
3.5 Organization of this Dissertation 6

I DATA FLOW ANALYSIS OF LOGIC PROGRAMS **7**

4 Logic Programming and Prolog **8**
4.1 Syntax of Logic Programs 8
4.2 Semantics of the Formulas 10
4.3 SLD Resolution 15
4.4 Derivation Tree 16
4.5 Prolog 17
4.6 Algorithmic Debugging 17

5 The Slicing of Logic Programs **19**

6 The Debug Slicing of Logic Programs **20**
6.1 Introduction 20
6.2 Basic structures and theorems for constructing the Debug Slice 21
6.2.1 The Augmented SLD-tree of Logic Programs 21
6.2.2 Skeleton(n) 24
6.2.3 Proof Tree Dependence Graph 27
6.2.4 Directed Proof Tree Dependence Graph 28
6.2.5 General Data Flow Slice 29
6.3 Debug Slice of Logic Programs 30
6.3.1 Potentially Dependent Predicates 31
6.3.2 Debug Slice 33
6.4 Prototype Implementation 36
6.5 Related Work and Discussion 37

II DATA FLOW ANALYSIS OF CLP PROGRAMS **39**

7 Constraint Logic Programming **40**
7.1 Syntax of Constraint Logic Programs 40
7.2 Logical Semantics of Constraint Logic Programs 42
7.3 Operational Semantics of Constraint Logic Programs (SLD-trees) 45

7.4 Derivation Tree 48
7.5 Program and Proof Tree Positions 49

8 Slicing of Constraint Logic Programs 51
8.1 Introduction 51
8.2 The Slicing Problem 52
8.3 Dependency-based slicing 54
8.3.1 Slicing sets of constraints 54
8.3.2 Slicing of derivation trees 55
8.3.3 Slicing of CLP programs 57
8.3.4 Discussion 59
8.4 Directional slicing 59
8.4.1 Groundness Annotations 60
8.4.2 Directional slicing of derivation trees 61
8.5 Using the calling context 62
8.6 A Prototype Implementation 64
8.7 Related Work 70
8.8 Conclusions 70

III LEARNING OF CONSTRAINT LOGIC PROGRAMS 72

9 The Learning of Constraint Logic Programs 73
9.1 Introduction 73
9.2 Preliminaries 75
9.2.1 Machine learning 75
9.2.2 The SPECTRE algorithm 75
9.3 Specialization of CLP Programs by Unfolding 76
9.3.1 The specialization problem 76
9.3.2 Unfolding 77
9.3.3 Clause Removal 82
9.3.4 The CLP_SPEC Algorithm 82
9.3.5 The Correctness of the CLP_SPEC algorithm 84
9.4 Improving the CLP_SPEC Algorithm by Slicing 88
9.4.1 Combining Algorithmic Debugging and Slicing for Constraint Logic Programs 88
9.4.2 The CLP_SPEC_SLICE Algorithm 89
9.4.3 The Correctness of the CLP_SPEC_SLICE Algorithm 90
9.5 Prototype Implementation 90
9.6 Related Work 95
9.7 Summary 96

IV LEARNING OF ATTRIBUTE GRAMMARS 98

10 Learning Semantic Functions of Attribute Grammars 99
10.1 Introduction 99
10.2 Preliminaries 100
10.2.1 Attribute Grammars 100
10.3 The AGLEARN method 103
10.3.1 Learning semantic functions of S-attributed grammar 104
10.3.2 Learning L-attributed semantic functions 105
10.4 The Computational Model of PAGE 106
10.5 PAGELEARN: A parallel approach to AGLEARN 109
10.5.1 Parallel Learning of S-attribute grammars 109
10.5.2 How the method works for L-attributed grammars 114
10.5.3 Analysis of the method used 117
10.6 Summary 117

11 Summary 119

12 Összefoglalás 121
12.1 Logikai Programok Adatfolyam Analízise 121
12.2 Korlátozásos Logikai Programok Adatfolyam Analízise 122
12.3 Korlátozásos Logikai Programok Tanulása 123
12.4 Attribútum nyelvtanok Szemantikus Függvényeinek Tanulása 123

13 Related papers of the author 125

List of Figures

1 The Trace-tree for the goal a(Y) and the Debug slice in frames. 24
2 The pruning effect of the cut in Example 5 for the goal a(X). 24
3 Skeleton(5) for Example 4. 27
4 The Proof Tree Dependence Graph for $Skeleton(5)$ 28
5 The Directed Proof Tree Dependence Graph and the data flow slice with respect to $(5, 0, 1, \{Y\})$ 29
6 A skeleton for the CLP program of Example 12. 49
7 A tree representation of a term 49
8 The identification of a "term's leaf" 50
9 A slice of a constraint set, a proof tree and a program. 52
10 Program dependence represented in graphical form along with the backward slice with respect to Z in p(X,Y,Z)) 58
11 The annotated proof tree for Example 15 60
12 A two-pass static slice computation for the variable s(X) 63
13 The Program Dependency Graph and two-pass static slice for the variable Sum2 65
14 Proof Tree, Static, Dynamic, and Program Position Slice. 66
15 Displayed program and proof tree slices. 67
16 Comparison of the Proof Tree Slice, Dynamic Slice, Program Position Slice and Static Slice. 68
17 Slice Size distribution. 69
18 The skeleton of the SLD-tree of $P'^{(0)}$ for the goal $fishlightmeal(A, M)$ . . 86
19 The skeleton of the SLD-tree of $P'^{(1)}$ for the goal $fishlightmeal(A, M)$ . . 87
20 *Slave* Processes generation. 107
21 Network Supervisor Structure. 108
22 Node Structure. 108
23 General Algorithm of the PAGELEARN Method. 110
24 The circuity problem: the waiting processes. 111
25 The circuity problem: the dependency graph. 111
26 Checking Algorithm for the Circuity Problem 112
27 Process mapping for the Example 25 114
28 Derivation tree for the Example 25 115
29 Dependency graph for detecting the circuity problem of Example 25 115
30 General algorithm for the L-attributed grammar 116

List of Tables

1 Test Results of the Debug Slice Algorithm 37
2 Test Results of The Proof Tree Slice Algorithm 68
3 The generated propositional table 105
4 The circuity problem: Neighbouring Matrix M. 110
5 The T(a) table of Example 25 113

1 ACKNOWLEDGMENTS

I would like to express my heart-felt thanks to Dr Tibor Gyimóthy who sparked my interest in this area of research and was so supportive during the period of my PhD studies.
In addition I would like to thank Dr János Csirik, Dr Zoltán Ésik and Dr Zoltán Fülöp for their kindness and inspiration.
I am immensely grateful to Dr. Jan Małuszyński whom I learned so much during my stay in Sweden.
Other people who deserve a mention are László Harmath and Gabriella Kókai who were good colleagues.
Lastly, I have to express my gratitude to my husband, parents and grandmother for their patience and understanding.
They are wonderful people in my life.

"Where there is no counsel, purposes are disappointed;
But in the multitude of counsellors they are established."

Proverbs 15.22

Köszönetnyilvánítás

Ezúton szeretném köszönetemet kifejezni témavezetőmnek Dr. Gyimóthy Tibornak, aki érdeklődésemet ezen terület felé irányította és munkámat az elmúlt évek során messzemenően támogatta.
Köszönöm Dr. Ésik Zoltán, és Dr. Csirik János támogatását, Dr. Fülöp Zoltán inspiráló óráit.
Hálával tartozom Dr. Jan Małuszyńskinek, akitől szintén nagyon sokat tanultam.
Köszönöm továbbá kollegáimnak Harmath Lászlónak, és Kókai Gabriellának támogató együttműködésüket.
Hálás vagyok szüleimnek, nagymamámnak, akik lehetővé tették tanulmányaimat, és férjemnek támogatásáért, türelméért, hogy elkészülhetett ez a tézis.

2 ABSTRACT

*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 various entities.*

*This dissertation concentrates on three declarative paradigms, namely **Logic Programming (LP)** [50, 44], **Constraint Logic Programming (CLP)** [46, 32] and **Attribute Grammars (AG)** [1]. There is a close relationship between Attribute Grammars (AG) and Logic Programming (LP) [14], and Logic Programming (LP) can be viewed as a special case of Constraint Logic Programming (CLP) [32].*

*One of the results of this dissertation is the slicing of Logic and Constraint Logic Programs. **Slicing** [29] is a program analysis technique which facilitates a better 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 this thesis a **dynamic slicing algorithm for Logic Programs** [78, 28, 27] 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 for the failure branches of the SLD-tree helps improve the existing debugging techniques. It can help to detect dead code and is useful in program maintenance.*

***Constraint Logic Programming (CLP)** [32, 46] is a fusion of two declarative paradigms, namely Constraint Solving and Logic Programming. In this thesis we lay the theoretical foundations for the **slicing of Constraint Logic Programs** [74, 75, 76], which will 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 [32, 46] and Machine (Inductive) Learning (ILP) [49, 42]. In this dissertation **a specialization technique is provided for learning of CLP programs** [79]. 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)** [91, 38, 1] are a generalization of the concept of Context-Free Grammars [36]. 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 us find good definitions of AGs, which usually require a lot of effort.*

*Based on the correspondence between Attribute Grammars and Logic Programs [14], some of the learning technique developed for Logic Programs (ILP) [49, 42] 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) [77]. This parallelism can help provide a more efficient approach than the sequential one, in both execution time and interactions needed.*

3 INTRODUCTION

Besides the **imperative**programming paradigm an other main type is the **declarative** paradigm. Declarative programs state what a program will do, as opposed to imperative programs where the emphasis on how the program will do it. As a result, declarative programs are closer to user requirements, and in general much shorter and easier to understand and reason with.
The first implementations of declarative programming languages dates back to the mid 1960's with the McCarthy invention of LISP (1958). Since then Prolog (1972), Standard ML (1984) and many other systems have paved the way the use of declarative programming as practicle tools.
This dissertation deals with four main subjects in the domain of declarative paradigms, namely the **data flow analysis and slicing of Logic Programs** [78, 28, 27], **data flow analysis and slicing of Constraint Logic Programs** [74, 75, 76], **learning of Constraint Logic Programs** [79] and **learning semantic functions of Attribute Grammars** [77].
There is close relationship between these areas [53, 54], in [14] a grammatical view (AG) of Logic Programming (LP) is provided, and Logic Programming (LP) can be viewed as a special case of Constraint Logic Programming (CLP) [32].
Based on these correspondence some of the learning technique developed for Logic Programs (ILP) could be applied to AGs, and learning methods of Logic Programs can be adopted for CLP programs and vice versa.

3.1 Data Flow Analysis of Logic Programs

The idea of **Logic Programming** (LP) [50, 44] is to use a computer for drawing conclusions from declarative descriptions. Such descriptions -called Logic Programs - consist of finite sets of logic formulas. In order to be able to apply a relatively simple and powerful inference rule (called the SLD-resolution principle), restrictions are introduced on the language of formulas.
Prolog [50], which is a well-known logic programming language, is based on first-order predicate logic. It uses a restricted form of formulas called Horn clauses.
The **Data flow analysis of Logic Programs** plays an important role for example in debugging, testing and program maintenance.
Slicing [29]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 to detect dead code or to find parallelism in programs [56, 73]. 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 scope and optimality of previous algorithmic debugging techniques of Prolog programs [69] using slicing techniques based

on a dependence 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 the data flow analysis for the failure branches of the SLD-tree helps improve existing debugging techniques, it can help one detect dead code and is useful in program maintenance.
A tool has been developed for debugging Prolog programs which also handles specific programming techniques (cut, if-then, or).
The relevant papers of the author for this are: [78, 28, 27].

3.2 Data Flow Analysis of Constraint Logic Programs

One other significant result of this dissertation is the **slicing of Constraint Logic Programs**.
Constraint Logic Programming (CLP) [32, 46] is an emerging software technology with a growing number of applications. It has the power to tackle those difficult combinatorial problems encountered for instance in job scheduling, timetabling, and routing which stretch conventional programming techniques to their limit.
Constraint Logic programming is a fusion of two declarative paradigms: constraint solving and logic programming. The framework extends classical logic programming [50, 44] by removing the restriction on programming within the Herbrand universe alone; unification is replaced by the more general notion of constraint satisfaction.
Data flow in constraint programs is not explicit, and for this reason the concepts of slice and the slicing techniques of imperative languages [4, 29, 30, 31] are not directly applicable.
This part of this work formulates **declarative notions of slice** suitable for CLP. (The problem of finding minimal slices may be undecidable in general, since satisfiability itself 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 which implements the idea presented above is briefly described together with the results of some slicing experiments.

The aim of this dissertation was to lay the precise theoretical background of the various slicing methods of Logic and Constraint Logic Programs and not to provide a detailed description of the implementation.
The relevant papers of the author for this are: [74, 75, 76].

3.3 Learning of Constraint Logic Programs

Inductive Constraint Logic Programming (ICLP) is a new research topic on the intersection of Constraint Logic Programming (CLP) [32, 46] and Inductive Logic Programming (ILP) [49, 42].
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 run very efficiently by focussing on a particular problem domain and they are good at handling numerical data as well.
This dissertation discusses the **specialization of Constraint Logic Programs by applying unfolding**, and as an improvement of the method, by combining unfolding with slicing technique. The transformation rule unfolding together with clause removal is a method (called SPECTRE [7]) for specialization of Logic Programs (LP).
Logic programming can be viewed as a special case of constraint logic programming, while CLP extends logic programs with constraints. This is a substantial extension, so CLP not only has syntactics distinct from LP but also distinct semantics.
In this part of the thesis we first formulate the exact definitions of the basic elements of an unfolding learning algorithm such as the specialization and unfolding transformation of CLP programs, providing their semantics as well. We prove that the unfolding transformation defined 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 which integrates an algorithmic debugging algorithm and the slicing method with the specialization algorithm for CLP programs.
A prototype learner of LP and CLP programs implementing the presented ideas is briefly described.
The relevant paper of the author for this is: [79].

3.4 Learning Semantic Functions of Attribute Grammars

In the framework of compilation oriented language implementation, **Attribute Grammars (AG)** [91, 38, 1] are the most widely applied semantic formalism. Attribute Grammars are generalizations of the concept of Context-Free Grammars [37]. 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 of Attribute Grammars and Logic Programs [14] some of the learning technique developed for Logic Programs (ILP) could be applied to AGs. Introducing an AG based description language in ILP implies the definition of an Attribute Grammar Learner. In this dissertation we present **a parallel method for learning semantic functions of Attribute Grammars** based on an ILP [49, 42] 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 the interactive learning process, so the parallel method is more efficient in both execution time and interaction needed than the sequential one.
The relevant paper of the author for this is: [77].

3.5 Organization of this Dissertation

This dissertation is organized as follows.
After the **abstract** and a general **introduction** about the scope, golas and main results of this work, we grouped the results into four main themes:

I. Data flow analysis of Logic programs (including Chapters 4, 5 and 6.)
II. Data flow analysis of Constraint Logic programs (including Chapters 7 and 8)
III. Learning of Constraint Logic programs (including Chapter 9.)
IV. Learning semantic functions of Attribute Grammars (including Chapter 10.)

Basic concepts of Logic Programming (LP) and Prolog (such as syntax, semantics, derivation tree, SLD-tree and algorithmic debugging) are reviewed in **Chapter 4**.
A general introduction about slicing is provided in **Chapter 5**.
One of the main result of this thesis, the Debug slicing of Logic Programs is discussed in **Chapter 6**.
The second main part of the thesis (II) includes an introduction to Constraint Logic Programs (CLP) (syntax, semantics, derivation and SLD-tree - see **Chapter 7**), and the detailed description of the slicing of Constraint Logic Programs (**Chapter 8**).
The results of Chapter 8 are then also used in **Chapter 9**, which lays the theoretical foundations and formalizes two learning methods of Constraint Logic Programs.
In **Chapter 10** a parallel learning method of semantic functions of Attribute Grammars is introduced.
Finally, **Chapter 11** and **12** contains a **summary** of this thesis (written in English and Hungarian) including a list of the new results (definitions, theorems and algorithms) of this research.
A **list of the related papers of the author** (**Chapter 13**) and the **Bibliography** is included at the end of the thesis.
An **extended abstract of the thesis** (about 15 pages) written in Hungarian and English is also included with the dissertation.

Part I
DATA FLOW ANALYSIS OF LOGIC PROGRAMS

The results of this part of the thesis are covered in [78, 28, 27].

4 Logic Programming and Prolog

The idea of logic programming is to use a computer for drawing conclusions from declarative descriptions. Such descriptions -called *logic programs* - consist of finite sets of logic formulas. Thus, the idea has its roots in the research on *automatic theorem proving*. In order to be able to apply a relatively simple and powerful inference rule (called the *SLD-resolution principle*), restrictions are introduced on the language of formulas.
This chapter introduces a restricted language of definite logic programs, their semantics and computational principles.

4.1 Syntax of Logic Programs

From the syntactic point of view logic formulas are finite sequences of symbols such as variables, functors and predicate symbols. Thus the **alphabet** of the language of the language of predicate logic consists of the following classes of symbols:

- **variables**, which will be represented by an upper case letter followed by a string of lower case letters and/or digits. Examples of variables are *X, List.*
- **function symbols**, which are alphanumeric identifiers represented by a lower case letter followed by a string of lower case letters and/or digits, and associated with an arity > 0. An example of a function is $f(X)/1$ where 1 is the arity of f.
- **constants**, which include integers and atoms; a constant is a function symbol of arity 0. The symbol for an **atom** can be any sequence of characters. Examples of constants are *mother, peter*.
- **predicate symbols**, which are alphanumeric identifiers represented by a lower case letter followed by a string of lower case letters and/or digits, and associated with an $arity > 0$. An example of a predicate is $p(X, Y)/2$ where 2 is the arity of p.
- **logical connectives**, which are $\wedge$ (conjunction), $\vee$ (disjunction), $\neg$ (negation), $\rightarrow$ (implication) and $\leftrightarrow$ (logical equivalence).
- **quantifiers**: $\forall$ (universal) and $\exists$ (existential).
- **auxiliary symbols** like parentheses, commas.

Sentences of natural language consist of words where objects of the described world are represented by nouns. In the formalized language of predicate logic objects will be represented by strings called terms whose syntax is defined in the following way:

Definition 1 ***Terms***
The set T of terms over a given alphabet A is the smallest set such that:

- *any constant in A is in T*
- *any variable in A is in T*

- *if f/n is a functor in A and $t_1, \cdots, t_n \in T$ then $f(t_1, \cdots, t_n) \in T$.*

Thus $f(g(X), head)$ is a term when f, g and $head$ are function symbols and X is a variable.

In natural language only certain combinations of words are meaningful sentences. The counterpart of sentences in predicate logic are special constructs built from terms. These are called *formulas* or well-formed formulas (*wff*) and their syntax is defined in the following way:

Definition 2 ***Formulas***
Let T be the set of terms over the alphabet A. The set F of wff *(with respect to A) is the smallest set such that:*

- *if p/n is a predicate symbol in A and $t_1, \cdots, t_n \in T$ then $p(t_1, \cdots, t_n) \in F$*
- *if G_1 and $G_2 \in F$ then so are $(\neg G_1)$, $(G_1 \wedge G_2)$, $(G_1 \vee G_2)$, $(G_1 \leftarrow G_2)$ and $(G_1 \leftrightarrow G_2)$*
- *if $G \in F$ and X is a variable in A then $(\forall XG)$ and $(\exists XG) \in F$*

*A predicate symbol immediately followed by a bracketed n-tuple of terms ($p(t_1, \cdots, t_n)$) is called an **atomic formula** (or simply **atom**).*

Definition 3 ***Bound Variables***
*Let F be a formula. An occurrence of the variable X in F is said to be **bound** either if the occurrence follows directly after a quantifier or if it appears inside the subformula which follows directly after "$\forall X$" or "$\exists X$". Otherwise the occurrence is said to be **free**.*

Definition 4 ***Closed and Ground Formulas***
*A formula with no free occurrences of variables is said to be **closed**. A formula or term which contains no variables is called **ground**.*

We will deal with a special type of declarative sentences of natural language that describe positive facts and rules. A sentence of this type either states that a relation holds between individuals (in the case of a fact), or that a relation holds between individuals provided that some other relations hold (in the case of a rule).

Definition 5 ***Horn Clauses***
Let $h, a_1, \cdots, a_m$ be atomic formulae for some $m \geq 0$ and let $X_1, \cdots, X_l$ be all variables occurring in these formulae.
Then the formula

$$\forall X_1 \cdots \forall X_l (h \leftarrow a_1 \wedge \cdots \wedge a_m)$$

or equivalently:

$$\forall X_1 \cdots \forall X_l (h \vee \neg a_1 \vee \cdots \vee \neg a_m)$$

*is called a **definite clause**.*
*If $m = 0$ the formula is called a **fact**.*
*The atomic formula h is called the **head** of the clause, while $a_1, \cdots, a_m$ is called its **body**.*
Since all variables of a definite clause are universally quantified we can omit the quantifiers, and comma is used instead of "$\wedge$".
So a definite clause can be written in the following form:

$$h \leftarrow a_1, \cdots, a_m.$$

Definition 6 ***Goal***
A **goal** *is a definite clause with empty head:*

$$\leftarrow G_1, \cdots, G_n.$$

or equivalently:

$$\neg G_1 \vee \cdots, \vee \neg G_n$$

where G_i *(*$i = 1, \cdots, n$*) are atoms.*

Definition 7 ***A (Definite) Logic Program***
A ***logic program*** *is a finite set of definite clauses having the form*

$$h_k \leftarrow a_{k_1}, \cdots, a_{k_m} \ (k = 1, \cdots, n; m >= 0),$$

where $h_k, a_{k_1}, \cdots, a_{k_m} \ (k = 1, \cdots, n)$ *are atoms.*

The following simple example makes use of the above definitions.

Example 1 *Consider the following simple logic program:*

```
1.  daughter(X,Y) : - female(X), parent(Y,X).
2.  female(mary).
3.  parent(ann,mary).
```

The clause

```
1.  daughter(X,Y) : - female(X), parent(Y,X).
```

is a definite clause, which corresponds to the following logical formula:

$$\forall X \forall Y (daughter(X, Y) \vee \neg female(X) \vee \neg parent(Y, X))$$

A fact is:

```
2.  female(mary).
```

A possible goal could have the following form:

```
:- daughter(X,mary).
```

– which means "who is the daughter of mary?"

4.2 Semantics of the Formulas

The previous section introduced the language of formulas as a formalization of a class of declarative statements of natural language. This section discusses the meaning of these formulas. In order to give the logical semantics of the program clauses (formulas) we have to define the notion of interpretation, semantics of terms, facts and program clauses.

Definition 8 ***Interpretation***
An interpretation $\Im$ *of an alphabet* A *is a nonempty domain* D *and a mapping that associates:*

- *each constant* $c \in A$ *with an element* $c_\Im \in D$
- *each n-ary function* $f \in A$ *with a function* $f_\Im : D^n \to D$
- *each n-ary predicate symbol* $p \in \Pi$ *with a relation* $p_\Im \subseteq D \times \cdots \times D$

The interpretation of constants, functions and predicate symbols provides a basis for assigning truth values to formulas of the language. The meaning of a formula will be defined as a function of the meanings of its components. First the meaning of terms will be defined since they are components of formulas. Since terms may contain variables the auxiliary notion of *valuation* is needed too.

Definition 9 Valuation
A valuation *of a set* S *of variables of an alphabet* A *is a mapping* φ *from* S *to the interpretation domain* D*.*

Thus, a valuation is a function which assigns objects of an interpretation to variables of the language.

Definition 10 Semantics of terms
Let $\Im$ *be an interpretation with the domain* D*,* φ *a valuation and* t *a term. Then the meaning* $\varphi_\Im(t)$ *of* t *is an element in* D *defined as follows:*

- *if* t *is a constant* c *then* $\varphi_\Im(t) := c_\Im$
- *if* t *is a variable* X *then* $\varphi_\Im(t) := \varphi(X)$
- *if* t *is of the form* $f(t_1, \cdots, t_n)$ *then* $\varphi_\Im(t) := f_\Im(\varphi_\Im(t_1), \cdots, \varphi_\Im(t_n))$*.*

Notice that the meaning of a compound term is obtained by applying the function denoted by its main functor to the meanings of its principal subterms, which are obtained by recursive application of this definition.

The meaning of a formula Q with respect to interpretation $\Im$ and valuation φ is a truth value denoted by $\Im \models_\varphi Q$. The meaning depends on the components of the formula. So the semantics of clauses (the corresponding logical formulas) is defined based on the semantics of predicates, facts, logical connectives and quantors.

Definition 11 Semantics of formulas (logical form of clauses)
Let $\Im$ *be an interpretation ,* φ *a valuation and* Q *a formula (a logical form of a definite clause). The meaning of* Q *with respect to* $\Im$ *and* φ *is defined as follows:*

- $\Im \models_\varphi p(t_1, \cdots, t_n)$ *iff* $\langle \varphi_\Im(t_1), \cdots, \varphi_\Im(t_n) \rangle \in p_\Im$
- $\Im \models_\varphi (\neg F)$ *iff* $\neg(\Im \models_\varphi F)$
- $\Im \models_\varphi (F \wedge G)$ *iff* $\Im \models_\varphi F$ *and* $\Im \models_\varphi G$

- $\Im \models_\varphi (F \vee G)$ *iff* $\Im \models_\varphi F$ *or* $\Im \models_\varphi G$
- $\Im \models_\varphi (F \rightarrow G)$ *iff* $\Im \models_\varphi G$ *whenever* $\Im \models_\varphi F$
- $\Im \models_\varphi (F \leftrightarrow G)$ *iff* $\Im \models_\varphi (F \rightarrow G)$ *and* $\Im \models_\varphi (G \rightarrow F)$
- $\Im \models_\varphi (\forall X F)$ *iff* $\Im \models_{\varphi[X \rightarrow t]} F$ *for every* $t \in D$
- $\Im \models_\varphi (\exists X F)$ *iff* $\Im \models_{\varphi[X \rightarrow t]} F$ *for some* $t \in D$

Model and Logical Consequence

The semantics of formulas as defined above relies on the auxiliary concept of valuation, which associates variables of the formulas with elements of the domain of the interpretation. It is easy to see that the truth value of a closed formula depends only on the interpretation. Since the valuation is of no importance for closed formulas it will be omitted when considering the meaning of such formulas.

Given a set of closed formulas P and an interpretation $\Im$ it is natural to ask whether the formulas of P give a proper account to the "world" described by P. This is the case if all formulas of P are true in $\Im$.

Definition 12 ***Model***
An interpretation $\Im$ is said to be a ***model*** *of a closed formula Q iff Q is true in $\Im$ (denoted $\Im \models Q$ - the formula is closed so φ can be omitted).*
$\Im$ is a D-model of a set of formula if it is model of every formula in the set.

Definition 13 ***Logical Consequence***
Let P be a set of closed formulas. A closed formula F is called a ***logical consequence*** *of P (denoted $P \models F$) iff F is true in every model of P.*

It may be rather difficult to prove that a formula is a logical consequence of a set of formulas. The reason is that one has to use the semantics of the language of formulas and to deal with all models of the formulas.
One possible way to prove $P \models F$ is to show that $\neg F$ is false in every model of P, or put alternatively, that the set of formulas $P \cup \{\neg F\}$ is unsatisfiable (has no model).

Theorem 1 ***Unsatisfiability***
Let P be a set of closed formulas and F a closed formula. Then $P \models F$ iff $P \cup \{\neg F\}$ is unsatisfiable.

The SLD- resolution is an inference rule which is appropriate for creating logical consequences just by the symbolic manipulation of a set of formulas, and is based on the above proposition.

Definite programs can only express positive knowledge. Therefore using the language of definite programs, it is not possible to construct contradictory descriptions, i.e. unsatisfiable

set of formulas. In other words every definite program has a model. It can be shown that every definite program has a well defined *least model* (with respect to the subset ordering). We will focus on models of a special ones known as *Herbrand models*.

Definition 14 ***Herbrand universe and Herbrand base***
Let A be an alphabet containing at least one constant symbol. The set U_A of all ground terms constructed from functions and constants in A is called the ***Herbrand universe*** *of A. The set B_A of all ground, atomic formulas over A is called the* ***Herbrand base*** *of A.*

The Herbrand universe and Herbrand base are often defined for a given program. In this case it is assumed that the alphabet of the program consists of exactly those symbols which appear in the program. It is also assumed that the program contains at least one constant.

Definition 15 ***Herbrand interpretation***
A Herbrand interpretation of P is an interpretation $\Im$ such that:

- *The domain of $\Im$ is U_P.*
- *For every constant c, $c_\Im$ is defined to be c itself.*
- *For every n-ary function f the function $f_\Im$ is defined as follows:*
  $$f_\Im(X_1, \cdots, X_n) := f(X_1, \cdots, X_n).$$
  That is, the function $f_\Im$ is applied to n ground terms composes them into the ground term with the principal function f.
- *For every n-ary predicate symbol p the relation $p_\Im$ is a subset of U_p^n (the set of all n-tuples of ground terms) such that*
  $$p_\Im = \{\langle t_1, \cdots, t_n\rangle \in U_p^n \mid \Im \models p(t_1, \cdots, t_n)\}.$$

Definition 16 ***Hebrand model***
A ***Herbrand model*** *of a set of (closed) formulas is a Herbrand interpretation that is a model of every formula in the set.*

For the restricted language of definite programs, it turns out that in order to determine whether an atomic formula A is a logical consequence of a definite program P it suffices to check that every Herbrand model of P is also a Herbrand model of A.

Theorem 2 ***Hebrand model***
Let P be a definite program and G a definite goal. If I is a model of $P \cup \{G\}$ then $\Im := \{A \in B_P \mid I \models A\}$ is a Hebrand model of $P \cup \{G\}$.

Theorem 3 ***Model Intersection Property***
Let M a non-empty family of Hebrand models of a definite program P. Then the intersection $\Im := \bigcap M$ is a Herbrand model of P.

Theorem 4 ***The least Herbrand model***
The least Herbrand model M_P of a definite program P is the set of all ground atomic logical consequences of the program.
That is, $M_P = \{A \in B_P \mid P \models A\}$.

Emden and Kowalski [16] created a consequence operator for constructing the least Herbrand model:

Definition 17 ***Immediate Consequence Operator***
Let $ground(P)$ be the set of all ground instances of clauses in P. T_P is a function of Herbrand interpretations of P defined as follows:

$$T_P(I) := \{A_0 \mid A_0 \leftarrow A_1, \cdots, A_m \in ground(P) \wedge \{A_1, \cdots, A_m\} \subseteq I\}$$

For definite programs it can be shown that there exists a least interpretation $\Im$ such that $T_P(\Im) = \Im$ and that $\Im$ is identical with the least Herbrand model M_P.

Substitutions

Definition 18 ***Substitution***
A **substitution** *is a finite set (possible empty) of pairs of the form X/t, where X is a variable and t is a term and all the variables X are distinct. For any substitution $\sigma = \{X_1 \rightarrow t_1, \cdots, X_n \rightarrow t_n\}$ and term s, the term $s\sigma$ denotes the result of replacing each occurrence of the variable X_i by t_i ($i = 1, \cdots, n$). The term $s\sigma$ is called an* **instance** *of s.*

Definition 19 Clause instance
A For any substitution σ the clause $c\sigma = (h : -b_1, \cdots, b_n)\sigma = h\sigma : -b_1\sigma, \cdots, b_n\sigma$ is called an **instance of the clause** *c.*

Definition 20 ***Composition of substitutions***
Let θ and σ be two substitutions:
$\theta := \{X_1/s_1, \cdots, X_m/s_m\}$, $\sigma := \{Y_1/t_1, \cdots, Y_n/t_n\}$.
The composition $\theta\sigma$ of θ and σ is obtained from the set:
$\theta := \{X_1/s_1\sigma, \cdots, X_m/s_m\sigma, Y_1/t_1, \cdots, Y_n/t_n\}$
by removing all $X_i/s_i\sigma$ for which $X_i = s_i\sigma$ $(1 \leq i \leq m)$ and by removing those Y_j/t_j for which $Y_j \in \{X_1, \cdots, X_m\}$ $(1 \leq j \leq n)$.

Definition 21 ***Unifiers***
A substitution σ is called a **unifier** *for two terms s_1 and s_2 if $s_1\sigma = s_2\sigma$. Such a substitution is called* **the most general unifier** *of s_1 and s_2 if for any other unifier σ_1 of s_1 and s_2, $s_1\sigma_1$ is an instance of $s_1\sigma$.*

If two terms are unifiable they have a unique most general unifier.

Example 2 *Let c_1 be a clause:*

c_1: `daughter(X,Y) :- female(X), parent(Y,X).`

and

$\sigma = \{X/mary, Y/ann\}$ *a substitution.*

Then,

$c_1\sigma$ = `daughter(mary,ann) :- female(mary),parent(ann,mary).`

Let c_3 be an other clause (fact):

c_3: `parent(ann,mary).`

Then σ is a unifier of c_3 and `parent(Y,X)`.

4.3 SLD Resolution

In the previous chapter the model-theoretic semantics of definite programs was discussed. The SLD-resolution principle, whose basis was first introduced by J. A. Robinson in the mid-sixties, makes it possible to draw correct conclusions from the program, thus providing a foundation for a logically sound operational semantics.

A **computation of a logic program** P [69] can be described informally as follows. The computation starts from some initial goal g and can have two results: success or failure. If a computation succeeds, then the final values of the variables in g are considered as the output of the computation. A given goal can have several successful computations, each resulting in a different output.

The computation progresses via nondeterministic goal reduction, and at each step we have some current goal $G = g_1, \cdots, g_n$. A clause $C = a \leftarrow b_1, \cdots, b_k$ in P is then chosen nondeterministically; the head of the clause a is then unified with g_1, say, with substitution σ, and the reduced goal is $G' = (b_1, \cdots, b_k, g_2, \cdots, g_n)\sigma$. Then G' is said to be **derived** from G and C. The computation terminates when the current goal is empty.

Definition 22 ***Derivation***

Let P be a logic program and G a goal. A **derivation** *of G from P is a possible infinite sequence of triples $< G_i, C_i, \sigma_i >$, $i = 0, 1, \cdots$ such that G_i is a goal, C_i is a clause in P with new variable symbols not occurring previously in the derivation, σ_i is a substitution, $G_0 = G$, and G_{i+1} is derived from G_i and C_i with substitution σ_i, for $i \geq 0$. A computation rule $\Re$ is used for choosing a goal to be unified at each step.*

Definition 23 ***An SLD-refutation***

If there is a derivation of G from P such that $G_l = \diamondsuit$ (the empty goal) for some $l \geq 0$ we say that P succeeds *on G and this kind of derivation is called an* **SLD-refutation**. *We assume by convention that in such a case $C_l = \diamondsuit$ and $\sigma_l = \{\}$.*

Definition 24 ***Failed derivation***

A derivation of a goal clause G_0 whose last element is not empty and cannot be resolved with any clause of the program is called a ***failed derivation***.

Definition 25 ***Computed answer substitution***

The computed substitution of an SLD-refutation of G_0 restricted to the variables in G_0 is called a ***computed answer substitution*** *for G_0.*

All such derivations may be represented by a possibly infinite tree called the SLD-tree of G_0 (using P and R).

Definition 26 ***SLD-tree***
Let P be a definite program, G_0 a definite goal and R a computation rule. The SLD-tree of G_0 (using P and R) is a (possible infinite) labelled tree satisfying the following conditions:

- *the root of the tree is labelled by G_0*
- *if the tree contains a node labelled by G_i and there is a renamed clause $C_i \in P$ such that G_{i+1} is derived from G_i and C_i via R then the node labelled by G_i has a child labelled by G_{i+1}. The edge connecting them is labelled by C_i.*

It can be proved that the SLD-resolution is correct (the conclusions produced by the system are correct, i. e. they are logical consequences of the program) and complete.

Theorem 5 ***Soundness of SLD-resolution***
Let P be a definite program, R a computation rule and θ an R-computed answer substitution for a goal $\leftarrow A_1, \cdots, A_m$. Then $\forall((\leftarrow A_1, \cdots, A_m)\theta)$ is a logical consequence of the program.

The completeness answer whether all correct answers for a given goal (i.e. all logical consequences) can be obtained by the SLD-resolution.

Theorem 6 ***Completeness of SLD-resolution***

Let P be a definite program, $\leftarrow A_1, \cdots, A_n$ a definite goal and R a computation rule. If $P \models \forall((\leftarrow A_1 \wedge \cdots \wedge A_n)\sigma)$, there exists a refutation of $\leftarrow A_1, \cdots, A_n$ via R with the computed answer substitution θ such that $(A_1 \wedge \cdots \wedge A_n)\sigma$ is an instance of $(A_1 \wedge \cdots \wedge A_n)\theta$.

4.4 Derivation Tree

One branch of the SLD-tree can be represented by another structure called a derivation tree.

Definition 27 Derivation Tree
For a program P a **derivation tree** *is any labeled, ordered tree T such that*

1. *Every node is labeled by an instance name of a clause of P.*
2. *Let n be a node labeled by an instance name $< c, \sigma >$, where c is the clause $h \leftarrow a_1, \cdots, a_m$ ($m \geq 0$) and σ is a substitution. Then n has m children, for $i = 1, \cdots, m$, the i-th child of n being labeled $< c'_i, \sigma'_i >$, where c'_i is a clause with head h'_i such that $a_i\sigma = h'_i\sigma'_i$.*

The reasoning behind this definition is that every tree is obtained by combining appropriate instances of program clauses. The precise meaning of ”appropiate instances” is expressed in condition 2. A logic program defines a set of derivation trees. This may be viewed as a semantic of definite programs.

4.5 Prolog

Prolog is a simple but powerful programming language developed at the University of Marseille [65] as a practical tool for programming in logic [41]. Prolog [50] is a logical and a declarative programming language that has been successfully applied in a number of commercial applications including telecommunications, CAD, digital design, electronics design, process scheduling, manufacturing design, train dispatching, legal reasoning, expert systems, theorem proving, and natural language processing.
The name itself, Prolog, is short for **PRO**gramming in **LOG**ic. Prolog's heritage includes the work on theorem provers and other automated deduction systems developed in the 1960s and 1970s.
Prolog is the major example of a fourth generation programming language supporting the declarative programming paradigm. The Japanese Fifth-Generation Computer Project, announced in 1981, adopted Prolog as a development language.
There are a number of different Prolog-implementations nowdays. The M-Prolog system was developed by Hungarian researchers [17, 18]. Here we have used Sicstus Prolog [72] as the implementation language of our algorithms.
The "pure" part of Prolog is simply a realization of refutations of logic programs on a sequential machine.
The Prolog interpreter searches the SLD-tree to find success branches. The Prolog system always selects the leftmost atom in a goal along with a depth-first search rule. The program clauses are then tested in their original order in the program.
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 of 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. However, in the right-hand part execution goes on as usual.
Prolog's sequential proof procedure is correct, as any refutation it finds is indeed a refutation, but is incomplete, as it may fail to find a refutation although one exists. This may happen if there are both refutations and infinite computations for a given goal, which arise from different choices of clauses, and Prolog happens to start and explore an infinite computation first.

4.6 Algorithmic Debugging

Algorithmic debugging [69, 57] is a process where the user and a debugging system interactively try to locate the source of an externally visible bug in the program [69]. There are a number of features of Prolog program which distinguish it from other programming languages. A Prolog program can have both a declarative and a procedural reading, may or may not be multi-directional, and it can even be nondeterminate. The computation model of Prolog is based on goal invocation as well as goal success and failure. Thus errors in Prolog programs occur when, for example, they finitely fail on goals that should succeed, or succeed on goals that should fail.
Shapiro's algorithm is an interactive diagnosis algorithm which identifies a bug in a program that behaves incorrectly. The theoretical foundations for the debugging of Logic Programs are presented in [69].

To debug an incorrect program one needs to know the expected behavior of the target program. Therefore we assume the existence of an agent, that knows the target program and may answer queries concering its behavior.
A debugging algorithm accepts as input a program to be debugged and a list of input / output samples, which partly define the behavior of the target program. It executes the program on the inputs of the samples, builds up the corresponding derivation trees and, whenever the program is found to return an incorrect output, it traverses the proof tree of a program in different ways and asks the user about the expected behavior of each resolved goal.
The *bottom-up method* traverses the proof tree in postorder manner and asks the oracle about the correctness of the computed values of the nodes. If the result at a node is incorrect and all sons of this node are evaluated correctly the algorithm identifies the clause applied to this node as a buggy one. The query complexity of this method is linear in the size of the tree.
The second method investigates the nodes in a *top-down* manner. If the result computed at a node is evaluated correctly by the oracle then the algorithm does not visit the nodes inside the sub-tree. Using this approach the query complexity can be reduced to a linear dependence in the depth of the proof tree.
The most efficient technique is the *divide-and-query* strategy, which requires a number of queries logarithmic in the size of the proof tree. The divide-and-query algorithm splits the proof tree into two approximately equal parts, and makes a query for the node at the splitting point. If this node gives an incorrect evaluation the algorithm goes on recursively to the sub-tree associated with this node. If the node's answer is correct its sub-tree is removed from the tree and a new mid-point is computed.

5 The Slicing of Logic Programs

Slicing is a program analysis technique originally developed for imperative languages [90]. Later improvements are presented in [81, 70, 40, 34].
Intuitively a program slice with respect to a specific variable V at some program point p (which can be a variable or an argument position of a predicate) contains all those program points that may affect the value of the variable or may be affected by the value of the variable. The tuple $< V, p >$ is called a *slicing criterion* and a slice is computed with respect to one.
Slicing techniques can also be classified into *static* and *dynamic* ones.
Static slicing is based on an analysis of the program without executing it so it may be imprecise if it contains data flow which is actually not manifested during a particular execution.
Dynamic slicing is based on the program's execution and hence extracts the precise data flow. A dynamic slice may be different for each execution and so shall always be produced separately whenever the program run.
In addition, slicing can be classified into *forward* and *backward* types. Suppose our slicing criterion is $< V, p >$. Forward slicing with respect to $< V, p >$ contains all those program points that may have its value modified if $< V, p >$ is modified. Backward slicing contains all those program points which, if modified, might change the value of V.
In the case of imperative languages a possible program representation for the program's dependences is the Program Dependence Graph [19, 51, 31].
The problem of slicing logic programs is more complicated than that for the imperative case. Before a slice over a logic program can be produced, its implicit data flow has to be approximated. To approximate the data flow the implicit input/output data dependences have to be extracted from the program.
Program slicing has been widely studied for imperative programs [81], but research on slicing logic programs is just beginning. To our knowledge, only a few papers deal with the problem of slicing logic programs [23, 66, 93, 76].
One of the aims of this work is to furnish a dynamic slicing algorithm which augments the data flow analysis with control-flow dependences so as to make the slicing algorithm more suitable for debugging. The precise definitions associated with slicing are given in Section 6.2.

6 The Debug Slicing of Logic Programs

This part of the dissertation extends the scope and optimality of previous algorithmic debugging techniques of Prolog programs using slicing techniques.
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 to the failure branches of the SLD-tree helps improve the existing debugging techniques. It can also help one to detect dead code and is useful in program maintenance.
The defined dynamic slicing method (called Debug slice) augments the data flow analysis with control-flow dependences in order to identify the source of a bug present in a program.

6.1 Introduction

Slicing methods are widely used for the debugging [89], testing [5] and maintenance of imperative programs [4, 29]. Intuitively, a slice should contain all those parts of a program that may affect the variables in a set V at a program point p [90]. Slicing algorithms can be classified according to whether they only use statically available information (*static slicing*), or compute those program points which influence the value of a variable occurrence for a specific program input (*dynamic slice*). Dynamic slicing methods are more appropriate for debugging than static ones as during debugging we generally investigate the program behaviour under a specific test case. The main advantage of using a dynamic slice during debugging is that many statements can be ignored in the process of bug localization.
Various dynamic slicing methods have been introduced for debugging imperative programs [81]. Most of these methods are based on a dependence graph which contains the explicit control dependences and data dependences of the program. In [23, 39] a slicing method was introduced for logic programs, and this method was used to improve the efficiency of the Shapiro's algorithmic debugging algorithm [69]. The slice presented in [23] contains those parts of a program that actually have an influence on the value of an argument of a predicate. This type of slice (called *data flow slice*) is safe if the structure of the proof tree for a goal is not changed [76].
However, during debugging when we seek to locate the source of a bug (i.e. a bug instance) we also need to identify those predicates that actually did not affect an argument in a predicate but could have affected it had they been evaluated differently (had their boolean outcome been different). We can say that these predicates are in the Potentially Dependent Predicate Set. Note that a different evaluation of the predicates in this set could change the success branch of the SLD-tree (where the bug was manifested).
Consider the following example.

Example 3 ***The Data Flow slice is insufficient to locate 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) :- $\mathbf{A} > \mathbf{0}$*, X is 2.*	*2. q(A,X) :-* $\mathbf{A} = \mathbf{0}$*, 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 there must be a bug in the program somewhere.
Creating the dynamic data flow slice for an instance of X, this 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 $A > 0$ (see Section 6.3.2).

In this section a new type of slicing is introduced, called *Debug slicing* for Prolog programs without side effects. A Debug slice of an Augmented SLD-tree includes those predicates that may affect the value of an argument in any success branch's predicate. So this slice is very suitable for debugging. The Debug slice is the set of predicates which contains the Potentially Dependent Predicates and their data dependences.
This slicing method has been integrated into an interactive algorithmic debugging tool to reduce the number of questions posed to the user during a debugging session [39]. The size of the debug slice is larger than the size of the data flow slice, but the data flow slice is not safe for debugging. On the other hand the Debug slice contains all parts of the program that may be responsible for the incorrect behaviour at some selected argument position.
The next section provides a detailed description of the construction of those structures needed in an outline of the Debug slice algorithm (Augmented SLD tree, Skeleton(n), (Directed) Proof Tree Dependence Graph, General Data Flow Slice). The computation of the Debug slice on the basis of these structures is described in Section 6.3. The first results of a prototype implementation of Debug slice algorithm are discussed in Section 6.4. Finally, in Section 6.5 we summarize related work and outline further studies.

6.2 Basic structures and theorems for constructing the Debug Slice

Our slicing is based on a dependence based approach. In [23] a Dependence Graph was constructed for a proof tree i.e. for a success branch of the SLD-tree. We would also like to extend this definition to the failure branches of the SLD tree. That is why this section provides a detailed description of the necessary structures needed to outline this extension: the Augmented SLD-tree, Skeleton(n) (a modified derivation tree), (Directed) Proof Tree Dependence Graph (PTDG) and General data flow slice. The computation of the Debug slice on the basis of these structures is described in the next section.
The *Augmented SLD-tree* shows the execution order of the statements for a given input.
Skeleton(n) is always built for one branch of the Augmented SLD-tree, and its nodes represent the data flow information needed for preparing the *Proof Tree Dependence Graph*.
A *General slice* of a logic program with respect to a variable V is constructed using the Proof Tree Dependence Graph, which contains those predicates of a derivation that may affect the value of V.

6.2.1 The Augmented SLD-tree of Logic Programs

The derivation of a goal from a program P can be represented by a tree called the SLD-tree. Each branch of the SLD-tree [61] 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*. The Prolog interpreter searches the SLD-tree to find success branches. The Prolog system always selects the leftmost atom in a goal along with a depth-first search rule. The program clauses are then tested in their original order in the program.
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. The atom "!" is handled as an ordinary atom in the body of a clause. When a cut is selected for resolution it succeeds immediately (with the empty substitution) [50]. The node where cut is selected will be called the *cut node*. A cut node may be reached again during backtracking. In this case the normal order of tree traversal is altered - from the definition of cut the backtracking continues above the node origin (!). (If cut occurs in the initial goal, the execution simply terminates). So cut has the following effect: after success of "!" no backtracking to the literals in the left-hand part is possible. However, in the right-hand part execution goes on as usual.
We add these pieces of information to the SLD-tree, identify each node with an unique mark, and use a list ($pred\_def\_ref()$) in order to know which program clauses (corresponding to the selected predicate) are used at a node to execute the next step. We also deal with the pruning effect of cuts. The following definition provides a formal description of the modified SLD-tree. The node label contains the whole list of goals ($(G', R) = (a_1, a_2, \cdots, a_n)$). The actual goal ($G'$) is the first in this list (a_1 in our case). A node has a child for every program clause whose head (h_m) could be unified with the actual goal. The list of these clauses for every node is given in *pred_def_ref()* (Definition 1.1). If the actual goal were *cut(!)*, the corresponding branches of the tree would be pruned (Definition 28.2 and 28.3).

Definition 28 ***Augmented SLD-tree***
Let P be a Prolog program and G a goal. An **augmented SLD-tree** *for $P \cup \{G\}$ is a tree which satisfies the following:*

- Each node label is triple $< Mark, (G', R), pred\_def\_ref(G') >$, where $Mark$ is a *unique identification* of the node, (G', R) is a (possibly empty) conjunction of goals (*resolvent*). G' is the leftmost goal in the resolvent (called the *selected goal*) and $pred\_def\_ref(G')$ is a *reference list* for the predicate definitions corresponding to the leftmost goal G'. We assign a a true value to the empty resolvents.

- The root node is $< Mark, (G, true), pred\_def\_ref(G) >$.

- Let $< M, (a_1, a_2, \cdots, a_k), pred\_def\_ref(i_1, \cdots, i_l) >$ be a node in the tree (so a_1 is the selected atom), where $i_m (m = 1, \cdots, l)$ is the identity number of input clauses $h_m \leftarrow b_{m_1}, \cdots, b_{m_q}$ such that a_1 and h_m are unifiable with most general unifier σ.

  1. Then this node has a child
  $< M_{i_m}, (b_{m_1}, b_{m_2}, \cdots, b_{m_q}, a_2, \cdots, a_k)\sigma, pred\_def\_ref(b_{m_1}) >$ for each $i_m (m = 1, \cdots, l)$. The edges immediately below a node and also the $pred\_def\_ref()$ list are ordered from left to right, according to the program clause order.

2. If $h_m \leftarrow b_{m_1}, \cdots, b_{m_q}$ has cuts the child is
   $< M_{i_m}, (b`_{m_1}, b'_{m_2}, \cdots, b'_{m_q}, a_2, \cdots, a_k)\sigma, pred\_def\_ref(b'_{m_1}) >,$
   where $b'_{m_1}, \cdots, b'_{m_q}$ are obtained from $b_{m_1}, \cdots, b_{m_q}$, replacing all cuts with the same unique annotation such as "cut(M)", where M is the node's identification mark.
3. If the following selected atom ($b`_{m_1}$ or a_2) were cut(M), we use the pruning rule below, and the next element of the list that follows cut(M) will be the selected goal.
   The pruning rule: If "Mark" is the argument of cut(), consider the path W from the actual node up to the node marked Mark. All descendants of this node to the right of W are removed.

- Nodes with an empty R list in resolvent have no children.

We will refer to the Augmented SLD-tree simply as the SLD-tree. During the execution of a program for a goal to find the first success branch of the SLD-tree, only a part of it is walked by the Prolog interpreter. We will call this part of the SLD-tree the **Trace-tree** because we can build the Trace-tree from the trace of a program for a goal given by the interpreter. Figure 1 shows the Trace-tree of the program in Example 4 for the goal $a(Y)$.

Example 4 ***The definition of the Trace-tree***
We will now make use of our definition of the Trace-tree in a simple example:

1. a(Y):-b(X), c(Y), d(X).
2. c(Y):-b(Y), e(Y).
3. b(1).
4. b(2).
5. b(3).
6. d(2).
7. e(3).

Example 5 ***The pruning effect of the cuts***
Figure 2 shows the pruning effect of the cuts in the following example for the goal $a(X)$.

1. a(X) :- b(X), c(X).
2. a(X) :- g(X).
3. b(X) :- c(X), !, d(X).
4. b(X) :- e(X), h(X).
5. c(1). *8. e(1).*
6. c(2). *9. h(1).*
7. d(2). *10.g(3).*

The removed part of the Trace-tree is depicted by a broken line (the pruning effect of the cut).

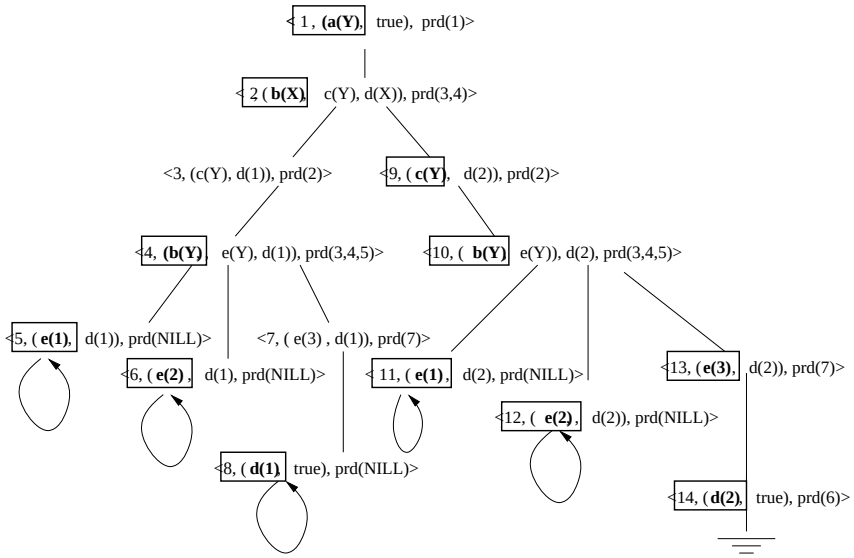

Figure 1: The Trace-tree for the goal a(Y) and the Debug slice in frames.

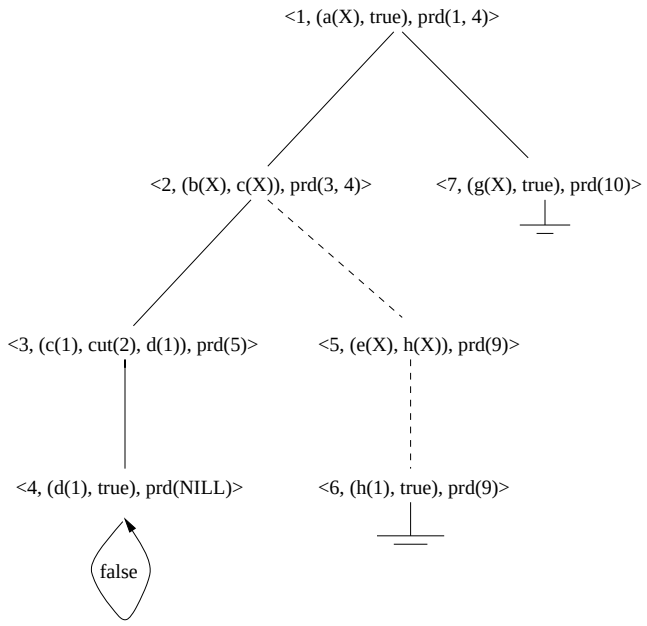

Figure 2: The pruning effect of the cut in Example 5 for the goal a(X).

6.2.2 Skeleton(n)

The SLD-tree representation is unsuitable for representing the data flow information of a logic program (for a given goal). The structure $Skeleton(n)$ [50] is used to represent this information, where n identifies a leaf node of the SLD-tree. $Skeleton(n)$ is basically a derivation tree defined for one branch of the SLD-tree (from the root to the node marked by n). To improve the approximation of the implicit data flow $Skeleton(n)$ contains directionality information as well.

We will use the notion of *clause instance* ($c\sigma$) which means that a substitution σ is applied for every predicate of c.
We extended the definition of the derivation tree used in [50] to our case.

Definition 29 Proof Tree
For a program P a **proof tree** *is any labeled, ordered tree T such that*

1. *Every node is labeled by an instance name of a clause of P.*
2. *Let n be a node labeled by an instance name $< c, \sigma >$, where c is the clause $h \leftarrow a_1, \cdots, a_m$ ($m \geq 0$) and σ is a substitution. Then n has m children, for $i = 1, \cdots, m$, the i-th child of n being labeled $< c'_i, \sigma'_i >$, where c'_i is a clause with head h'_i such that $a_i\sigma = h'_i\sigma'_i$.*

The reasoning behind this definition is that every tree is obtained by combining appropriate instances of program clauses. The precise meaning of "appropiate instances" is expressed in condition 2. A logic program defines a set of derivation trees. This may be viewed as a semantic of definite programs, and can be related to the concepts of proof defined in symbolic logic.
A derivation tree represents one branch of the SLD-tree (one derivation), but in a more suitable format for representing the data flow.
Let us modify this definition to suit our present needs.
We need not know exact the substitution itself; it is enough to know which variables are ground at call of a predicate and which are ground at success. Basically, having to investigate directionality information of the tree requires using some groundness annotation. Let us suppose that we can identify each argument position of the clauses of a derivation tree with a tuple (the formal definition of the argument position is given at the end of this subsection).

Groundness information associated with a derivation tree will be expressed as an annotation of its argument positions. The annotation classifies the argument positions of a derivation tree. The positions are classified as *inherited* (marked with $\downarrow$), *synthesized* ($\uparrow$) and *dual* ($\updownarrow$). An annotation is *partial* if some positions are dual.

Definition 30 ***Annotation***
An annotation is a mapping ν from the positions in the set $\{\downarrow, \uparrow, \updownarrow\}$ [14].

The intended meaning of the annotation is the following.

Definition 31 ***Inherited Argument Position***
An **inherited argument position** *is a position in which every variable is ground at the time of calling, that is when the equation involving this position is first created during the construction of the derivation tree.*

Definition 32 ***Synthesized Argument Position***
A **synthesized argument position** *is a position in which none of the variables are ground at the time of calling, and every variable is ground at success, that is when the subtree having the position in its root label is completed in the computation process.*

Definition 33 ***Dual Argument Positions***
The **dual argument positions** *of a proof are those which are neither inherited nor synthesized. The annotations are collected during the execution of a program for a given goal.*

We notice that the argument positions are annotated in the present version of our tool. But the annotation of the variable positions would provide more precise dependences so we are planning to extend the annotation to variable positions, too.
We now introduce the following auxiliary terminology relevant to the annotated positions of a LP program.

Definition 34 ***Input, Output and Dual Positions***
The inherited positions of the head atoms and the synthesized positions of the body atoms are called **input positions***. Similarly, the synthesized positions of the head atoms and inherited positions of the body atoms are called* **output positions***. The others are* **dual***. Note that dual positions are not strictly classified as input or output ones.*

Alternatively, if we say that a position is annotated as an output we mean that it is annotated as inherited provided it is a position in a body atom, or annotated as synthesized if it is a position of the head of a clause.

Now we are ready to define $Skeleton(n)$.

Definition 35 ***Skeleton(n)***
Let T be a SLD-tree, $< n, (G', R), pred\_def\_ref(G') > \in nodes(T)$ a leaf node identified by n. Consider the path W in T from the root to n, which identifies a derivation for the root goal.
Then, $Skeleton(n)$ is a labeled ordered tree such that

1. *Every node is labeled by a double $< Mark, \nu(c) >$, where Mark is an unique identification, and $\nu(c)$ is the annotated clause instance.*

2. *The root node is labeled by $< 1, \nu(c) >$, where the root goal was unified with c during the given derivation.*

3. *Let k be a node labeled by*
   $< Mark, p([X_1, \nu(X_1)], \cdots, [X_{k_0}, \nu(X_{k_0}])) :-$
   $a_1([Y_1, \nu(Y_1)], \cdots, [Y_{k_1}, \nu(Y_{k_1})]), \cdots, a_m([V_1, \nu(V_1)], \cdots, [V_{k_m}, \nu(V_{k_m}])) >$.
   Then k has m children, for $i = 1, \cdots, m$, the i-th child of k being labeled $< Mark, \nu(c'_i) >$, where c'_i is a clause whose head was unified with a_i during the given derivation.

Figure 3 shows Skeleton(5) of Example 4. The variable Y of $a(Y)$ in node 1 is annotated as output, since it would be ground at success of $a(Y)$, and $a(Y)$ is a head atom. For the same reason X in node 2, Y in node 3 and 4 are annotated as output. The variable Y in node 5 is ground at call, so it is annotated as input.

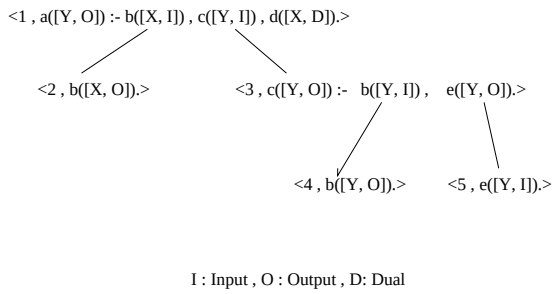

Figure 3: Skeleton(5) for Example 4.

Definition 36 ***Argument Position***
We can refer to the k-th ***argument position*** *in each node of the skeleton by the tuple* $(Mark, i, k, V)$*, where* $Mark$ *is the identity number of the node,* i *is the number of the predicate in the clause (from 0 to m),* k *is the argument position of* a_i *and* V *is the set of variables at this position. If* V *also contains Input and Output variables it is annotated as Dual.*
Let $Pos(S)$ *denote the set of argument positions of the Skeleton(n)* S*.*

As can be seen from the definition of the Augmented SLD-tree and $Skeleton(n)$, there is an one to one correspondence between the nodes belonging to one branch of the SLD-tree (identified by n) and the nodes of the corresponding $Skeleton(n)$. This correspondence is based on the fact that both structures describe the same derivation for a goal step by step. In our formalism the Mark of a node highlights this correspondence.

Definition 37 ***Map from the nodes of*** S ***to the nodes of*** T
Let T *be an SLD-tree for the goal* g*,* $n \in nodes(T)$ *a leaf of* T*, and* S *the* $Skeleton(n)$*. Then, there is a map from the nodes of* S *to the nodes of* T *such that:*

$$\begin{aligned} \phi : \quad & nodes(S) \rightarrow nodes(T) \\ & < Mark, \nu(p :- a_1, \cdots, a_m) > \rightarrow < Mark, (p, R), pred\_def\_ref(p) >. \end{aligned}$$

If $S' \subseteq nodes(S)$ *then let* $\phi(S')$ *denote the corresponding subset of* $nodes(T)$ *such that*

$$\phi(S') = \{\phi(n) | n \in S'\}.$$

For $n \in nodes(T)$ *let* $\phi^{-1}(n) = m \in Pos(S)$*, such that* $\phi(m) = n$*.*

Now we are ready to define the Proof Tree Dependence Graph.

6.2.3 Proof Tree Dependence Graph

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 [10, 11]. The following definition reflects these conditions.

Definition 38 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.
Figure 4 shows the PTDG for $Skeleton(5)$.

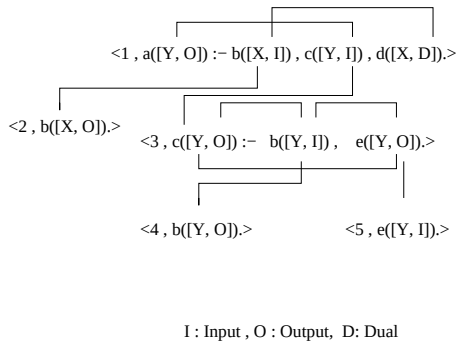

Figure 4: The Proof Tree Dependence Graph for $Skeleton(5)$

6.2.4 Directed Proof Tree Dependence Graph

As mentioned earlier we would like to better approximate the implicit data flow by introducing directionality using an annotation technique. The annotations can be collected during the execution of the program. Based on this annotation, the Proof Tree Dependence Graph can be directed because the data flows from an Input position to an Output one via a local edge, and from an Output position to an Input one via an transition edge. This can be expressed more precisely in the following definition.

Definition 39 ***Directed Proof Tree Dependence Graph***
Let $T_g = (Pos(S), \sim_T)$ be a proof tree dependence graph, $\alpha, \beta \in Pos(S)$. Then the **directed proof tree dependence graph** *is $\vec{T}_{g,n} = (Pos(S), \rightarrow_T)$, where*

1. $\alpha \rightarrow_T \beta$ *if* $\alpha \sim_T \beta$, $\sim_T$ *is a local edge and one of the following conditions holds:*
   - α *is an Input position and* β *is an Output position*
   - α *is a Dual position and* β *is an Output position*
   - α *is an Input and* β *is a Dual position*
   - α *is a Dual and* β *is a Dual position (in this case* $\alpha \rightarrow_T \beta$ *and* $\beta \rightarrow_T \alpha$*)*
2. $\alpha \rightarrow_T \beta$ *if* $\alpha \sim_T \beta$*, the positions being connected by a transition edge and satisfying one of the following conditions:*
   - α *is an Output position and* β *is an Input position*
   - α *is a Dual and* β *is a Dual position (in this case* $\alpha \rightarrow_T \beta$ *and* $\beta \rightarrow_T \alpha$*)*

It is quite easy to check the validity of these rules. It is possible to define more precise conditions to direct the edges (referring to the textual occurrences of the positions), but it would be too complicated to present them here. Our experience shows that the use of these rules (which permit in some cases non-realisable data flow) gives good results. Our slicing algorithm applies to this Directed Proof Tree Dependence Graph.

The Directed Proof Tree Dependence Graph for Skeleton(5) is depicted in Figure 5.

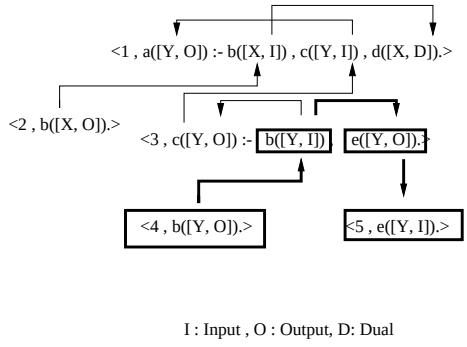

Figure 5: The Directed Proof Tree Dependence Graph and the data flow slice with respect to $(5, 0, 1, \{Y\})$

6.2.5 General Data Flow Slice

In this section a *general slice definition* is given which shows a given argument position of Skeleton(n) which other argument positions depends on (from the aspect of data flow).

Definition 40 ***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.

Figure 5 shows $slice(T_{a(y),5}, (5, 0, 1, \{Y\}))$. The argument of $e(Y)$ in node 5 ($(5, 0, 1, \{Y\})$) is an Input position, the slice being constructed with respect to this position. The set of all positions from which there is a directed path to this argument position (this is the slice with respect to $(5, 0, 1, \{Y\})$) contains the argument position of $e(Y)$ and $b(Y)$ in node 3, and $b(Y)$ in node 4.

In [23] a data flow slice was defined for the success branch of the SLD-tree (Trace-tree). We extended the data flow slice definition to the full SLD-tree (Trace-tree), based on the Directed Proof Tree Dependence Graph.
This general slice definition may not only used for constructing the Debug Slice, but the data flow analysis of the failure branches to provide a good basis for program maintenance. It is useful in incremental testing [5] and can help one detect dead code or find parallelism in programs [29, 56, 73].

6.3 Debug Slice of Logic Programs

A **bug manifestation** is undesired program behavior, i.e. an undesired sequence of solutions computed by the program for a goal. A **bug instance**, which is a predicate instance, is a cause for a top goal bug manifestation. Our algorithm (**Debug Slice**) identifies a set of predicates of a program which can cause a bug manifestation (i.e. an undesired solution with respect to a variable of the top goal).
Analyzing just the data flow of a program is not always enough to find the source of a bug manifestation (as is shown by Example 3). It is necessary to deal with control dependences as well.
The idea behind the definition of the Debug Slice comes from imperative languages, which there is called the ”relevant slice” [21]. The relevant slice could be used to find the kind of 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.
Similarly, with Prolog we deal with those predicates which could affect the control of the search of the Interpreter in such a way that this effect could be responsible for a bug manifestation. In Example 3 the predicate $A > 0$ failed for the goal $p(0, X)$. The Prolog interpreter made a backtrack, so this failure give rise to a bug manifested at the value of X (the given solution was $X = 3$, while we expected $X = 2$). The bug is included in the predicate instance $A > 0$.
We note that our philosophy does not strictly separate the bugs into different sets such as ”wrong solution” or ”missing solution” suspect sets, as is usual.
Our aim was to provide a slicing tool for programmers to help them in debugging (extending incrementally the set of predicates which may be responsible for a bug manifestation,

complementing it with knowledge of the data flow of the failure branches), program maintenance, general analysis and so on.

As we mentioned earlier the **Debug slice** of a Trace-tree (T) deals with those leftmost (selected) predicates in a node of T, that do not directly affect the value (V) of an argument of a predicate in the success branch of T, but could affect the control of the search of the Interpreter in such a way that this effect could create a bug manifestation at V.

So, the question is how the searching strategy of the Interpreter is affected by the predicate instances of the program.

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.

To create the Debug slice, firstly we specify the ***Potentially Dependent Predicates Set*** *(PDPS)* which contains those predicates that actually did not affect the selected argument, but could have done so had they been evaluated differently (i.e. had they succeeded or failed). This set covers case **(1)** above (see Section 6.3.1).
In Subsection 6.3.2 the **Debug slice** is defined on the Augmented SLD-tree (Trace tree) which includes the Potentially Dependent Predicates (case **(1)**), their associated data dependences (case **(3)**) and the predicates affected by some cut (case **(2)**).

6.3.1 Potentially Dependent Predicates

Sometimes we cannot find the source of a bug just by analyzing the data flow for the success branch of the Trace-tree. So we have to examine which predicates might cause a branch of the Trace-tree to fail, or what would have happened if a predicate had succeeded but actually failed, or if it should have failed but actually succeeded. We concentrated on the leftmost (goal) predicate of an SLD node so the slice is defined for these predicates. The following definition covers these cases.

Definition 41 ***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.*

In the following we try to identify those predicates which satisfy this condition.

Theorem 7 ***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* $T\}$.

Proof

To prove the validity of this theorem we have to demonstrate that these predicates indead satisfy the condition in Definition 41 while the other predicates of T do not. To achieve this, we classify the predicates of T in such a way that the categories cover all predicates belonging to T. Notice that we use "the selected variable" expression but it could have been any variables of the program whose values do not meet our expectations (i.e. where a bug was manifested). The PDPS is the same for every selected variable, so it is created for a Trace tree built up for a given goal.

- **If a predicate should have failed but actually succeeded**

  1. If this predicate (selected goal) belongs to the success branch of the Trace-tree, then its boolean outcome could have affected the value of the selected variable (argument), so it could have been the source of the bug. We note here that this situation caused the modification of the structure of the Trace-tree.
  2. If this predicate belongs to a failure branch of the Trace-tree, then its boolean outcome could not have affected the value of the selected variable because, if had it failed it would then have caused the pruning of the subtree below this predicate. But this would have not modified the structure of the other parts of the Trace-tree.

- **If a predicate should have succeeded but it failed**

  Then this predicate is a leaf of a failed branch of the Trace-tree (because these are the only failed predicates). Its boolean outcome could have affected the value of the selected variable because it might have modified the structure of the Trace-tree.

To extend this theorem we recognize that if the user had found a bug in a success branch of the SLD-tree which was not the first one, then the predicates of the previous success branches did not belong to the PDPS because if they had failed the result would not have affected the structure of the later branches of the SLD-tree.

Example 6 *The PDPS of the Trace-tree in Figure 1 is the following (The nodes are identified with their marks):*
PDPS=$\{1, 2, 5, 6, 8, 9, 10, 11, 12, 13, 14\}$.

6.3.2 Debug Slice

In this section our main result *the Debug slice* is specified based on the definitions and theorem of the previous sections. The Potentially Dependent Predicate Set of a Trace-tree (for a logic program P and goal g) includes all those predicates whose boolean outcome may affect the value of an argument in a success branch's predicate. The Debug slice deals with control dependences as well. The Debug slice is a set of predicates which contains the Potentially Dependent Predicates [Definition 41], their data dependences [Definition 40] and the predicates affected by some cut.
Since the Debug slice contains all predicates of the success branch of T, the Debug slice is the same with respect to every selected argument. Hence the Debug slice is defined for a logic program P and goal g.
An interesting question is the effect of cuts. If there is a node in the Trace-tree whose leftmost goal is cut($Mark$) we remove all descendants of this node to the right of the path W up to the node marked $Mark$. We denote this kind of path by cut(W_{Mark}).

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

In the following we examine the contents of these sets in turn.

1. The predicates of the Potentially Dependent Predicate Set (PDPS)

The Potentially Dependent Predicate Set of a Trace-tree includes all predicates whose boolean outcome may affect the value of an argument in a success branch's predicate (see Theorem 7). The PDPS can affect the control of the search.

PDPS= $\{$The predicates of the success branch of $T\}$
$\cup \{$The predicates of the failed leaves of $T\}$.

2. The predicates specified by the data flow of the predicates of PDPS

Since the PDPS consists of two subsets, the first point is dealt with by examining two cases in turn.

- **The predicates that belong to the success branch of T**

  Here the data flow does not introduce new predicates into the Debug slice as the data flow slice is valid for one given branch of the Trace-tree (T), and all predicates of the

success branch of T are in the Debug slice.

- **The predicates of the failed leaves of** T

  Let $n \in PDPS$ such that n is a leaf node of a failed branch of T, S the Skeleton(n), $\vec{T}_{g,n}$ the corresponding directed Proof Tree Dependence Graph (see Section 6.2.4), and $\phi^{-1}(n)$ the corresponding node of S (see Section 6.2.2). Suppose that $\phi^{-1}(n)$ is labeled by $< n, p : -a_1, \cdots, a_m >$.
  Next, construct $slice(\vec{T}_{g,n}, \alpha)$ for every $\alpha \in Pos(S)$ such that α is an argument position belonging to the head predicate p.
  Let

  $H = \cup_{n,\alpha}\{k \in nodes(S) \mid k$ has at least one head argument position in $slice(T_{g,n}, \alpha)$, α is an argument position of p, n is a failed leaf of $T\}$.

  Afterwards, map H back to the Trace tree ($\phi(H)$). So $\phi(H)$ contains those predicates which are specified by the data flow of the failed leaves of T.

  Let the set of these predicates be denoted by $Data\_Flow\_of\_PDPS$.

  Definition 42 Data_Flow_of_PDPS

  $Data\_Flow\_of\_PDPS :$ = $\phi(H) =$
  $= \phi(\cup_{n,\alpha}\{k \in nodes(S) \mid k$ *has at least one head argument position in* $slice(T_{g,n}, \alpha)$, α *is an argument position of* p, n *is a failed leaf of* $T\})$.

  For example, one can see in Figure 5 that $n = 5$ is a failed leaf of the Trace tree (Figure 1), so $\phi^{-1}(5)$ is $< 5, (e(Y, I) >$. As the clause contained in this node is a fact, the head predicate is $e(Y)$, which has one argument position Y. Constructing the slice with respect to this argument position it contains a head argument position from node 4 and 5. So $Data\_Flow\_of\_PDPS$ in this case contains nodes 4 and 5 of the Trace tree in Figure 1.

We will now address the third point.

3. The predicates that belong to some cut(W) of T

If there is a node in the Trace-tree whose **leftmost goal is cut(Mark)** we remove all descendants of this node to the right of the path W up to the node marked $Mark$. We denote this kind of path by $cut(W_{Mark})$. Hence a cut may affect the control dependences and those nodes that belong to W have to be added to the Debug slice.

Definition 43 cut(W_{Mark})
Let T be a Trace-tree, $Mark$ a node identification of T.

$cut(W_{Mark})$ *: =* $\{p \in nodes(T) \mid p$ *is a leftmost predicate on the path from the node whose leftmost goal is* $cut(Mark)$ *up to the node identified by* $Mark\}$

Definition 44 Cut(W)_Set

$Cut(W)\_Set$ *: =* $\bigcup_{Mark} cut(W_{Mark})$

Example 7 *In Example 5, if we had not had a cut in clause 3, we would have got $X = 1$, so $< 5, e(X) >$ and $< 6, h(X) >$ would have affected the value of the variable X in $a(X)$, but this would not have shown up in data flow analysis.*

The following definition formalizes our main result the Debug slice according to Definitions 41, 42, 43, 44 and Theorem 7.

Definition 45 Debug Slice

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

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

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

Example 8 *In Example 3 the PDPS contains the buggy predicate $A > 0$, so $A > 0$ is in the Debug slice, but the data flow slice does not have it because this predicate belongs to a failed branch of the SLD-tree for the goal $p(0, X)$.*

Example 9 *We will now go on with Example 4.*

In order to construct the Debug slice we furnish the sets PDPS, $Data\_Flow\_of\_PDPS$ and $Cut(W)\_Set$.

1. We know from Example 6 that the PDPS of the Trace-tree in Figure 1 is the following (where the nodes are identified by their marks):
   PDPS=$\{1, 2, 5, 6, 8, 9, 10, 11, 12, 13, 14\}$.

2. To get $Data\_Flow\_of\_PDPS$ we have to construct a Skeleton(n) for every $n \in \{5, 6, 8, 11, 12\}$ and the corresponding Proof Tree Dependence Graphs. We also have to specify a $slice(\vec{T}_{g,n}, \alpha)$ for every α argument position of n in the Proof Tree Dependence Graph and to state every node of T that belongs to these slices.
   Figure 5 shows the Proof Tree Dependence Graph for Skeleton(5) and $slice(\vec{T}_{g,n}, (5, 0, 1, \{Y\}))$. We urge the reader to construct all the slices for each argument position of$\{5, 6, 8, 11, 12\}$. In our case the only node in T specified by these slices is node 4.
   So $Data\_Flow\_of\_PDPS = \{4\}$.

3. We had no cut in this example, so $Cut(W)\_Set$ is empty.

Then, **Debug slice**= $\{\{1, 2, 5, 6, 8, 9, 10, 11, 12, 13, 14\} \cup \{4\} \cup \{\}\}$ =
$\{1, 2, 4, 5, 6, 8, 9, 10, 11, 12, 13, 14\}$

In this example the only nodes that are not in the Debug slice are 3 and 7.
The Debug slice of the Trace-tree built for Example 4 is drawn and highlighted in Figure 1 .

6.4 Prototype Implementation

We implemented a prototype in Sicstus Prolog using the complete framework described for slicing Prolog programs. The implementation handles specific programming techniques (cut, if-then, OR). The Trace-tree is constructed from the call ports of the trace given by the Interpreter. The slice with respect to a node of the Trace-tree is created. The slicing technique, if desired, can be combined with Shapiro's debugging method [39, 69]. To first approximation the slice is built up for the success branch of the Trace-tree. The slice is then constructed with respect to the given argument position, and during a debugging session the system asks for the validity of just those nodes that are in the slice. If it is unsuccessful in locating the source of the bug, the Debug slice is constructed and the user can then request data flow information as well for any leaf node predicate of the Debug slice. A graphical interface draws the Trace-tree and highlights those nodes that are included in the data flow slice and in the Debug slice.
We tested our Debug Slice algorithm on several small Prolog programs. These programs can produce big fail branches for some inputs. The test results are shown in the Table 1 below. We examined the number of nodes and arguments in the whole Trace tree, in the success and failed branch of the Trace tree, and lastly in the Debug slice.

The test results demonstrate that if the number of the failed branch' nodes is high and the data flow slices for the failed branch's leaf predicates do not contain too many predicates, then the Debug slice is significantly smaller than the whole Trace tree. The test results of course depend on the size and type of input, as well.
The Debug slice method handles types of bugs which the conventional data-flow slice technique overlooks. Certain types of bugs were found during testing which were overlooked 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 misplaced.

	Complete Tree Nodes	Arg.	Success Branch Nodes	Arg.	Failed Branch Nodes	Arg.	Debug Slice Nodes	Arg.
1.	14	14	6	6	8	8	13	13
2.	15	26	6	9	9	17	7	10
3.	19	39	7	11	12	28	11	20
4.	34	64	15	15	19	49	23	39
5.	267	618	11	13	256	605	16	23
6.	316	769	15	15	301	754	23	39
7.	520	1393	24	49	496	1344	181	423
8.	622	1240	6	9	616	1231	7	10
9.	1142	2687	5	9	1137	2678	7	13

Table 1: Test Results of the Debug Slice Algorithm

- a failed predicate is mis-printed (its name or arity) or a condition ($<, >, =$) has failed.
- 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.

We note that the system finds only those misprinted predicates of the failure branches of the SLD-tree which occur in the data-flow of a failed predicate, or are affected by a cut.

6.5 Related Work and Discussion

While program slicing has been widely studied for imperative programs [81], relatively few papers have dealt with the problem of slicing logic programs [23, 66, 76, 93].
Gyimóthy and Paaki presented in [23] a specific slicing algorithm for sequential logic programs in order to reduce the number of user queries of an algorithmic debugger. But they only analyzed the data dependences for the success branch of the SLD-Tree (Trace-tree). Sometimes it is insufficient to locate the source of a wrong solution because the cause of the erroneous result may also be an invalid proof tree structure. We solved this problem by dealing with control dependences as well as extending the data flow analysis to the failure branches. So the data flow slice given in [23] is a special case of our approach.
Schoening and Ducasé have proposed a backward slicing algorithm for Prolog which produces executable slices [66]. An executable slice is usually less precise than a general slice [81], and their algorithm is only applicable to a limited subset of Prolog programs. Our aim was to develop a tool for debugging Prolog programs that also handles specific programming techniques.
In [93] Zhao at al. presented a new program representation called the argument dependence net for concurrent logic programs in order to produce static slices at the argument level. Dynamic slicing usually produces more precise slices than static ones because it only considers a particular execution of a program. We chose the dynamic version because our application focuses on debugging .
In [61] Pereira and Calejo examined the wrong solution suspect set (WSS) and the missing solution suspect set (MSS). It is possible to refine WSS using our general slice definition.

In [76] a dynamic slicing method was presented for constraint logic programs based on variable sharing and groundness analysis. In the paper the declarative formulation of the slicing problem for constraint logic programs was also described.
The Debug slicing method for Prolog programs has been introduced in this part of the dissertation. This slicing technique is very appropriate for debugging because it deals with control dependences as well. This slicing method will be integrated into the IDTS interactive algorithmic debugging tool [39]. This tool employs an improved version of Shapiro's debugging method [69] for identifying a buggy clause. By using slicer modules the number of user interactions can be reduced during the debugging process. The data flow slice is usually much smaller than the Debug slice, so in the first step we can try to locate the bug in the data flow slice. However it may happen that the data flow slice does not contain the buggy clause. In this case the debugging process has to be extended to the nodes of the Debug slice. We tested our slicing tool on small Prolog programs.

Part II
DATA FLOW ANALYSIS OF CONSTRAINT LOGIC PROGRAMS

The results of this part of the thesis are covered in [74, 75, 76].

7 Constraint Logic Programming

As the name suggests, Constraint Logic Programming (CLP) (see [46]) is descended from logic programming. The commercial acceptance of Prolog was hindered by its relatively poor efficiency compared to procedural languages like C, and its use has declined in recent years. *Prolog*, which is a well-known logic programming language, is based on first-order predicate logic and the objects that it manipulates are pure symbols with no intrinsic meaning: for example in the Prolog proposition *likes(jim, baseball)* the constants *jim* and *baseball* have no deeper interpretation beyond syntactic identity, ie. $jim = jim$.
The execution of a Prolog program proceeds by searching a database of such facts to find those values that will satisfy a user's query (goal), using a process called *unification* based on syntactic identity. Since Prolog tries to find the set of all solutions to a query (goal), during this search many dead-ends may be explored and then abandoned by backtracking to an earlier state and trying a different branch. For complex problems this search process can become very greedy in both space and time, which is the root cause of Prolog's inefficiency. CLP languages make logic programs execute very efficiently by focussing on a particular problem domain. Constraint Logic Programming (CLP) has the power to tackle those difficult combinatorial problems encountered for instance in job scheduling, timetabling, and routing which stretch conventional programming techniques to their limit. CLP is still the subject of intensive research, and is an emerging software technology with a growing number of applications. Logic programs can be treated as a special case of CLP [32].
The cornerstone of *Constraint Logic Programming (CLP)* [32, 46] is the notion of constraint. Constraints 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 equality interpreted over rational numbers, e.g. $X - Y = 1$. The variables of a constraint range over the domain of interpretation. In a CLP language this purely abstract logical framework is supplemented by objects that have meaning in an application domain, hence there isn't a single CLP language, but a whole family of them defined for different application domains. The constraints have to be satisfied for succesful execution of the program.
In such a CLP system the simple unification algorithm that lies at the heart of Prolog must be augmented by a dedicated 'solver' for the particular domain of application, which can decide at any moment whether the remaining constraints are solvable. For efficiency's sake, solvers for CLP systems need to be incremental, so that adding a new constraint to an already solved set does not force them all to be re-solved. For example a useful solver for linear rational constraints is the well-known simplex method.
The constrained search ability makes CLP languages good at precisely those problems that conventional programming techniques find hardest; NP-hard search problems where the time needed for an unconstrained search increases exponentially (or worse) with the problem size.

7.1 Syntax of Constraint Logic Programs

The cornerstone of *Constraint Logic Programming (CLP)* [32, 46] is the notion of constraint. In this section we provide some basic concepts of constraints logic programming.

Definition 46 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).*
In this paper we adopt the Prolog notation for variables *(identifiers starting with a capital) and* function symbols *(identifiers starting with lower case letters).*
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 an n-ary ($n > 0$) function symbol followed by a bracketed n-tuple of terms (The latter is called a compound term).*
Thus $f(g(X), head)$ is a term when f, g and $head$ are function symbols and X is a variable.

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

- *constraint predicates Σ, which are given a fixed interpretation in D. These include the symbol $=$, interpreted as identity.*
- *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.

The definitions above describe the syntax of CLP programs, and the following basic definitions help provide the semantics for them.

Definition 47 Groundness
A clause or an atom is ground *if it has no variable.*

Definition 48 Valuation
A valuation *of a set S of variables is a mapping φ from S to the interpretation domain D.*

Definition 49 Satisfiability
A set of constraints C is satisfiable *if there exists a valuation φ for the set of variables occurring in C, such that $\varphi(C)$ holds in the constraint domain.*

Definition 50 Clause instance
A substitution *is a finite, possibly empty, set of pairs of the form $X \rightarrow t$, where X is a variable and t is a term and all the variables X are distinct.*
For any substitution $\sigma = \{X_1 \rightarrow t_1, \cdots, X_n \rightarrow t_n\}$ and term s, the term $s\sigma$ denotes the result of replacing each occurrence of the variable X_i by t_i ($i = 1, \cdots, n$). The term $s\sigma$ is called an instance *of s.*
The clause $c\sigma = (h : -b_1, \cdots, b_n)\sigma = h\sigma : -b_1\sigma, \cdots, b_n\sigma$ is called an instance of the clause *c.*

Example 10 *In the following constraint logic program the equality constraint and symbols of arithmetic operations are interpreted over the domain of rational numbers. The constraints are distinguished by the curly brackets $\{\}$.*

Given the definition of a light-meal as consisting of an appetiser, a main meal and a dessert and a database of foods and their calorific values we wish to construct light-meals i.e. meals whose sum of calorific values does not exceed 10.

```
1.  lightmeal(A,M) : - {I + J ≤ 10}, appetiser(A,I),
                          main(M,J).
2.  appetiser(A,I) : - cheese(A,I), {I > 0}.
3.  appetiser(A,I) : - pasta(A,I), {I > 0}.
4.  main(M,J) : - fish(M,J), {J > 0}.
5.  main(M,J) : - meat(M,J), {J > 0}.
6.  fish(sole,2).
7.  fish(tuna,4).
8.  meat(beef,5).
9.  meat(chicken,4).
10.  pasta(general,1).
11.  cheese(camamber,2).
```

7.2 Logical Semantics of Constraint Logic Programs

The previous section introduced the language of formulas of Constraint Logic Programs and some basic definitions. Now we will discuss the meaning of these formulas. In order to give logical semantics of constraint logic programs we have to define the notion of interpretation,

semantics of terms, facts and program clauses. The following definitions have been adapted to our needs based on [32, 50].
Let $\widetilde{X}$ denote a sequence of distinct variables $X_1, \cdots, X_n$ for an appropriate n.

There are two common logical semantics of CLP programs over a constraint domain $\langle L, D \rangle$.

The first interprets a rule

$$h :- b_1, ..., b_n, \; n \geq 0$$

as the logic formula:

(1) $\forall \widetilde{X}, \widetilde{Y} \; h(\widetilde{X}) \vee \neg b_1 \vee \cdots \vee \neg b_n,$

where $\widetilde{X} \cup \widetilde{Y}$ is the set of all free variables in the clause.

The second logical semantics associates a logic formula to each predicate in Π.
If the set of all rules of P with p in the head is

$$\begin{array}{l} p(\widetilde{X}) \leftarrow B_1 \\ p(\widetilde{X}) \leftarrow B_2 \\ \cdots \\ p(\widetilde{X}) \leftarrow B_n \end{array}$$

Then the formula associated with p is

$$\begin{array}{lll} \textbf{(2)} & \forall \widetilde{X} p(\widetilde{X}) \leftrightarrow & \exists \widetilde{Y}_1 B_1 \\ & & \vee \exists \widetilde{Y}_2 B_2 \\ & & \cdots \\ & & \vee \exists \widetilde{Y}_n B_n \end{array}$$

where $\widetilde{Y}_i$ is the set of variables in B_i except for variables in $\widetilde{X}$. If p does not occur in the head of a rule of P then the formula is

$$\forall \widetilde{X} \; \neg p(\widetilde{X})$$

Definition 51 D**-Interpretation**
Let $\langle L, D \rangle$ denote 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 interpretation of constants, functions and predicate symbols provides a basis for assigning truth values to formulas of the language. The meaning of a formula will be defined as a function on meanings of its components. First the meaning of terms will be defined as they are the components of formulas. Since terms may contain variables as well we will make use of the notion of *valuation* defined in the previous section.

Definition 52 Semantics of terms
Let $\Im$ be a D-interpretation, φ a valuation and t a term. Then the meaning $\varphi_\Im(t)$ of t is an element in D defined as follows:

- *if t is a constant c then $\varphi_\Im(t) := c_\Im$*
- *if t is a variable X then $\varphi_\Im(t) := \varphi(X)$*
- *if t is of the form $f(t_1, \cdots, t_n)$ then $\varphi_\Im(t) := f_\Im(\varphi_\Im(t_1), \cdots, \varphi_\Im(t_n))$.*

Note that the meaning of a compound term is obtained by applying the function denoted by its main functor to the meanings of its principal subterms, which are obtained by the recursive application of this definition.

The meaning of a formula Q with respect to the interpretation $\Im$ and valuation φ is a truth value denoted by $\Im \models_\varphi Q$. The meaning of course depends on the components of the formula. So the semantics of clauses (the corresponding logical formulas) is defined based on the semantics of predicates, facts, logical connectives and quantors.

Definition 53 Semantics of formulas
Let $\Im$ be a D-interpretation , φ a valuation and F and G two formulas. Then:

- $\Im \models_\varphi p(t_1, \cdots, t_n)$ *iff* $\langle \varphi_\Im(t_1), \cdots, \varphi_\Im(t_n) \rangle \in p_\Im$
- $\Im \models_\varphi (a \leftarrow c)$ *iff* $\Im \models \varphi(c)$, *where* $a \leftarrow c$ *is a fact*
- $\Im \models_\varphi (\neg F)$ *iff* $\neg(\Im \models_\varphi F)$
- $\Im \models_\varphi (F \wedge G)$ *iff* $\Im \models_\varphi F$ *and* $\Im \models_\varphi G$

- $\Im \models_\varphi (F \vee G)$ *iff* $\Im \models_\varphi F$ *or* $\Im \models_\varphi G$
- $\Im \models_\varphi (F \rightarrow G)$ *iff* $\Im \models_\varphi G$ *whenever* $\Im \models_\varphi F$
- $\Im \models_\varphi (F \leftrightarrow G)$ *iff* $\Im \models_\varphi (F \rightarrow G)$ *and* $\Im \models_\varphi (G \rightarrow F)$
- $\Im \models_\varphi (\forall X F)$ *iff* $\Im \models_{\varphi[X \rightarrow t]} F$ *for every* $t \in D$
- $\Im \models_\varphi (\exists X F)$ *iff* $\Im \models_{\varphi[X \rightarrow t]} F$ *for some* $t \in D$

The logical meaning of a CLP program (with respect to $\Im$ and φ) can be obtained by the recursive application of this definition to the logic formulas (1) or (2).

Definition 54 D**-Model**
A D-interpretation $\Im$ is said to be a D-model of a closed formula Q iff Q is true in $\Im$ ($\Im \models Q$ - the formula is closed so φ can be omitted).
$\Im$ is a D-model of a set of formulas if it is a model of every formula in the set.

The definition of the least D-model will be used to prove the correctness of our unfolding algorithm.

Definition 55 The least D**-model of a formula**
Let $\Im$ denote a D-model of a CLP program with the constraint domain $\langle L, D \rangle$.
The least D-model of a formula Q under the subset ordering will be denoted by $lm(Q, \Im)$, and the least D-model of a set of formula P will be denoted by $lm(P, \Im)$.

The following definition provides the logical semantics of a solution.

Definition 56 Solution
A solution to a query G is a valuation φ such that $\varphi(G) \subseteq lm(P, \Im)$.

7.3 Operational Semantics of Constraint Logic Programs (SLD-trees)

This section presents the operational semantics of constraint logic programs, and the definition of the SLD-tree created for our needs based on this type of operational semantics.

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 [32]. 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 active role (they are awake), and S is a collection of constraints playing a passive role (they are asleep).
There is also another state, denoted by **fail**.
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 $\langle G, \emptyset, \emptyset \rangle$.

Let R_j denote a clause of a P constraint logic program 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.

The transitions in the transition system are:

- $\langle A \cup b, C, S\rangle \rightarrow_r \langle A \cup \{b_{j_1}, \cdots, b_{j_{m_j}}, c_j\}, C, S \cup (\overline{b} = \overline{h_j})\rangle$
  if b is a defined atom selected by the computation rule, $h_j \leftarrow b_{j_1}, \cdots, b_{j_{m_j}}, c_j$ is a rule of P, renamed to new variables, and h_j and b have the same predicate symbol. The expression $\overline{b} = \overline{h_j}$ is an abbrevation for the conjunction of equations between corresponding arguments of b and h_j.

  $\langle A \cup b, C, S\rangle \rightarrow_r fail$
  if b is a defined atom selected via the computation rule and, for every rule $h_j \leftarrow b_{j_1}, \cdots, b_{j_{m_j}}, c_j$ of P, h_j and b have different predicate symbols.

- $\langle A \cup c, C, S\rangle \rightarrow_c \langle A, C, S \cup c\rangle$
  if c is selected by the computation rule and c is a constraint.

- $\langle A, C, S\rangle \rightarrow_i \langle A, C', S'\rangle$
  if $(C', S') = infer(C, S)$.

- $\langle A, C, S\rangle \rightarrow_s \langle A, C, S\rangle$
  if $consistent(C)$.

  $\langle A, C, S\rangle \rightarrow_s fail$
  if $\neg consistent(C)$.

The predicate $consistent(C)$ expresses a test for the consistency of C. The function $infer(C, S)$ computes from the current set of active constraints a new set of active constraints C' and passive constraints S'.
The $\rightarrow_r$ transitions arise from the resolution, $\rightarrow_c$ transitions introduce constraints into the constraint solver, $\rightarrow_s$ transitions test whether or not the active constraints are consistent, and $\rightarrow_i$ transitions infer more active constraints from the current collection of constraints.

Definition 57 Derivation
*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 of $\langle \emptyset, C, S\rangle$.*

Definition 58 *The answer constraint of a derivation*
*Let G be a goal with free variables $\widetilde{X}$ which initiates a derivation and produces a final state $\langle \emptyset, C, S\rangle$. Let $\exists_{-\widetilde{X}} Q$ denote the existential closure of the formula Q except for the variables $\widetilde{X}$, which remain unquantified. Then $\exists_{-\widetilde{X}}\ C \wedge S$ is called **the answer constraint** of the derivation.*

It should noted that the operational semantics we are dealing with can be rewritten as $\rightarrow_r \rightarrow_i \rightarrow_s$ and $\rightarrow_c \rightarrow_i \rightarrow_s$.

The computation for a goal G can be described by a tree called the SLD-tree.

Definition 59 SLD-tree
Let P be a constraint logic program and G a goal. An SLD-tree for $P \cup \{G\}$ is a tree which satisfies the following:

- *each node label is a computational state $\langle A, C, S \rangle$ like that defined above*
- *the root node is $\langle G, \emptyset, \emptyset \rangle$*
- *each node has as many children as valid transitions are associated with it*
- *final states have no children*
- *the edges are labelled by the type of the transition (r, c, i, s)*

We note that every branch of the SLD-tree describes one derivation. A node may have more than one child only if the subsequent transition is an $\rightarrow_r$ transition (since the actual defined predicate could be "unified" with head predicates of different program clauses). A node has children for every clause whose head has the same predicate symbol. An $\rightarrow_r$ transition can be viewed as an operation, if instead of b, the body predicates of the "unified" clause are inserted and the corresponding argument equations are added to the set of constraints.

We will prove later that our specialization algorithm (CLP_SPEC) can be viewed as the problem of pruning an SLD-tree.

Example 11 *An $\rightarrow_r, \rightarrow_i, \rightarrow_s$ transition triple of Example 10 for the goal $lightmeal(A, M)$ is the following:*

$\langle lightmeal(A, M); \emptyset; \emptyset \rangle \rightarrow_r$

$\langle \{\{I_1 + J_1 \leq 10\}, appetiser(A_1, I_1), main(M_1, J_1)\}; \emptyset; \{A = A_1, M = M_1\} \rangle$

$\rightarrow_i \langle \{\{I_1 + J_1 \leq 10\}, appetiser(A_1, I_1), main(M_1, J_1)\}; \{A = A_1, M = M_1\}; \emptyset \rangle$

$\rightarrow_s \langle \{\{I_1 + J_1 \leq 10\}, appetiser(A_1, I_1), main(M_1, J_1)\}; \{A = A_1, M = M_1\}; \emptyset \rangle$

The third state is the same as the second one since C is consistent. This can be viewed as a part of the corresponding SLD-tree.

7.4 Derivation Tree

As slicing concerns computations we now introduce some relevant notions. Abstractly, *a computation of a CLP program* can be regarded as the construction of a tree (skeleton) from renamed instances of clauses. This structure for Logic Programs is discussed formally in [14] we extended it for CLP.

Definition 60 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; 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 61 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).*

Definition 62 Derivation tree and proof tree
*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 nodes) the derivation tree is called a **proof tree**.*

Example 12 *In the following constraint program the equality constraint and symbols of arithmetic operations are interpreted over the domain of rational numbers. A simple example has been chosen so as to simplify the forthcoming illustration of slicing concepts and techniques. The constraints are distinguished by the curly brackets $\{\}$.*

```
p(X,Y,Z):- {X-Y=1}, q(X,Y), r(Z).
q(U,V):- {U+V=3}.
r(42).
```

Figure 6 shows a complete skeleton tree for the program in Example 12.
The set of constraints of this skeleton is:
$C(S) = \{X - Y = 1, X = U, Y = V, U + V = 3, Z = 42\}$ and it is satisfiable. Hence the skeleton is a proof tree.

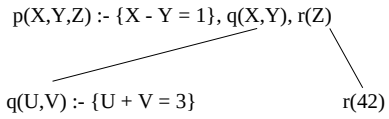

Figure 6: A skeleton for the CLP program of Example 12.

7.5 Program and Proof Tree Positions

In order to properly present the slicing techniques we need to refer to **program positions** and to **derivation tree positions**. The following definitions have been made by the author. 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.
To define the notion of position we assume that some standard way of enumeration of the nodes of any given tree T can be adopted.
Let c be a clause in P CLP program of the form $a_0 : -a_1, \cdots, a_n$ where a_j has the form $p_j(t_1, \cdots, t_k), p_j \in \Pi \cup \Sigma, j = 1, \cdots, n, t_i \in T, i = 1, \cdots, k$.
We can now refer to the A-th argument position in a_j with the tuple (C, J, A), where C is the clause number and J is the number of the predicate. But this is not enough for our task, since we would like to refer to a variable position inside the argument terms.
The recursive definition of terms make possible to view a term as a tree. Figure 7 shows the tree representation of the term $f(t_1, \cdots, t_n)$.

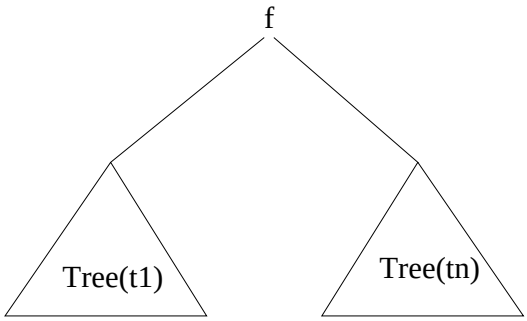

Figure 7: A tree representation of a term

Now each leaf of a term tree t can be represented by the pair (i, X) where X is the variable or the constant labelling a leaf of the tree and i is the number of the leaf in t counting from left to right. Clearly this number itself is sufficient to identify the leaf, and its label but sometimes we include the label for the sake of legibility. For example the occurrence of the variable Z in the term $V - 3 * (Z/2 + x)$ can be identified by the pair $(3, Z)$ or only by 3 as illustrated by Figure 8.

Definition 63 ***Program Positions***
A position of a variable X in the program can be identified by the four-touple (C, P, A, V) containing an identification number of the clause (C), the predicate (P), the argument (A) and the variable position (V) of the given argument .

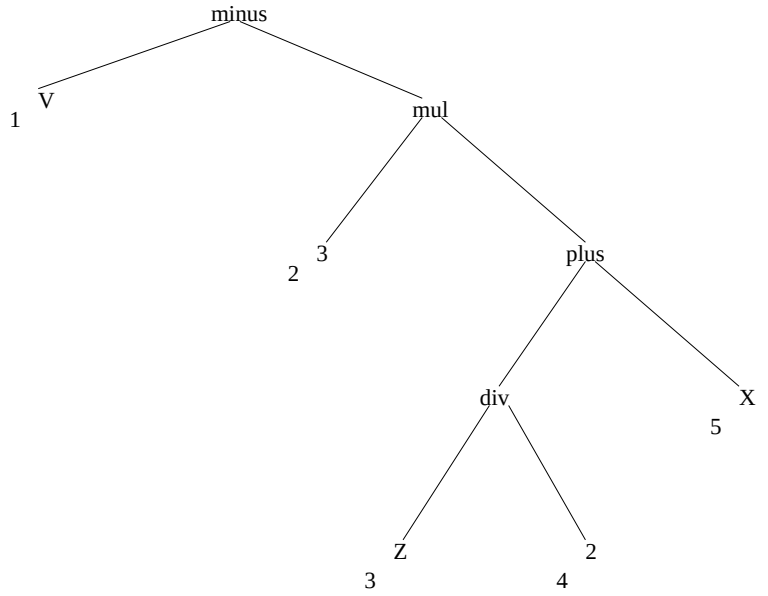

Figure 8: The identification of a "term's leaf"

Definition 64 ***Proof-tree Positions***
A proof tree position can be identified by the tuple $(Mark, P, A, V)$*, where* $Mark$ *is the identity number of the node,* P *is the number of the predicate in the clause (from 0 to m),* A *is the argument position of* P *and* V *is the identification number of the given variable.*

The set of all tree positions of a derivation tree T will be denoted by $Pos(T)$.
Note that each label of a derivation tree T is a variant of a program clause, or of a goal. Therefore the positions of T can be mapped in a natural way into the corresponding program positions.
Similarly, each occurrence of a variable X in $C(T)$ (the constraint set of T) originates from a variable position of X in T. Thus variable positions of T can be associated with the related constraints of $C(T)$.
Let $\mathcal{P}$ be a set of positions of T, and $\Psi_T(\mathcal{P})$ the set of all variables that appear in the terms at positions in $\mathcal{P}$. In this way $\mathcal{P}$ identifies the subset $C_{\mathcal{P}}$ of $C(T)$ consisting of all constraints including variables in $\Psi_T(\mathcal{P})$.

Example 13 *Consider the derivation tree of Figure 1.*
Let $(\mathcal{P}) = \{$ *derivation tree positions of the atom* $q(X, Y)\} \subseteq Pos(T)$.
Then $\Psi_T(\mathcal{P}) = \{X, Y\}$ *and* $C_{\mathcal{P}} = \{X - Y = 1, X = U, Y = V\}$.

8 Slicing of Constraint Logic Programs

Slicing is a program analysis technique originally developed for imperative languages. It facilitates understanding of data flow and debugging.
This section discusses the slicing of Constraint Logic Programs. Here we formulate declarative notions of slice suitable for CLP. They 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 implementing the presented ideas is briefly described together with the results of some slicing experiments.

8.1 Introduction

This part of the thesis discusses the slicing of Constraint Logic Programs. Constraint Logic Programming (CLP) (see e.g. [46]) is an emerging software technology with a growing number of applications. Data flow in constraint 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 behaviour rather difficult. Thus program analysis tools explaining data flow to the user could be of great practical importance. This part of the dissertation formulates declarative notions of a slice suitable for CLP and presents a prototype slicing tool for CLP programs written in SICStus Prolog [72] based on these notions.
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*).
Slicing algorithms can be classified as *static* or *dynamic*. In static slicing the information on the program needed to construct the slice is obtained without executing the program, typically by various static analysis techniques. Thus a static slice concerns all potential executions of the program. In contrast, a dynamic slice is constructed for a particular execution, and relies on the information collected during that execution.
The concept of slice in imperative programming provides a focus for analysis of the origin of the computed values of the variable in question. In the context of CLP the intuition remains the same, but the concept of slice requires a new definition since the nature of CLP computations is different from the nature of imperative computing.
Slicing techniques for logic programs have been discussed in [23, 66, 93]. CLP extends logic programming with constraints. This is a substantial extension and the slicing of CLP programs has, to our knowledge, not yet been addressed by other authors. Novel contributions presented in this work are:

- *A precise declarative formulation of the slicing problem for CLP programs.* 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.

- *Slicing techniques for CLP.* We present slicing techniques that make it possible to construct slices according to the definitions to given later on. The techniques are based on a simple analysis of variable sharing and groundness.

- *Handling calling context problem.* This section shows a way of constructing finer static slices of a program than those obtained by basic slicing.
- *A prototype slicer.* A tool implementing the proposed techniques and experiments (slice distribution) with possible applications is described.

The precisely defined concepts of slice provide a solid foundation for the development of slicing techniques. The presented prototype tool, including some visualisation facilities, assists the user in a better understanding of the program and in the (manual) search for errors. The integration of this tool with more advanced debuggers is a topic of future work. The paper is organized as follows. Section 8.2 formulates the problem of slicing for Constraint Logic Programs. Section 8.3 presents and justifies a declarative formalization of CLP slicing, based on a notion of dependency relation. Section 8.4 discusses a dynamic backward slicing technique, and the use of directionality information for reducing the size of slices. Section 8.5 outlines a way of reducing the size of the static program slice handling the "calling context" problem. Our prototype tool is described in Section 8.6, together with results of some experiments. Section 8.7 discusses related work. Finally in Section 8.8 we present our conclusions and suggestions for future work.

8.2 The Slicing Problem

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. This is rather imprecise, hence our objective is to formalize this intuition. We first define the notion of a slice of a satisfiable set of constraints. A variable X in a derivation tree T has its valuations [32, 46] restricted by the set of constraints of T (which is satisfiable), so our second task will be to define a slice of a derivation tree, and finally a slice of a program (see Figure 9).

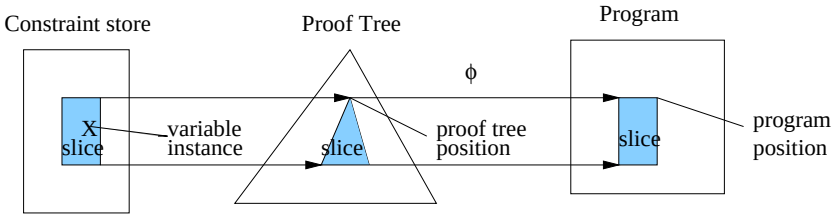

Figure 9: A slice of a constraint set, a proof tree and a program.

Let C be a set of constraints. Intuitively we would like to remove all constraints of the set that do not restrict the valuation of a given variable.

Definition 65 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 $Sol(X, C)$.*

Definition 66 ***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 $Sol(X, S) = Sol(X, C)$.*

In other words the set of all solutions of a slice S of C with respect to X is equal to the set of all solutions of C with respect to X. Definition 66 implicitly gives a notion of **minimal slice** with respect to X.

Definition 67 ***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* $Sol(X, C)$*.*

Note that the whole set C is a slice of itself, and that the definition does not provide any hint about how to find a minimal slice. The minimal slice may not be unique. Consider for example finite domain constraints in $C = \{X < Y, X < Z\}$, where X ranges over $\{0, 1\}$ and both Y and Z range over $\{1, 2\}$. Then each of the two constraints is a minimal slice of C with respect to X.
In general, the problem of finding minimal slices may be undecidable as satisfiability may be undecidable. So reasoning about the minimality of the constructed slices only seems possible in very restricted cases, and for some specific constraint domains. Our general technique is domain independent but we show in Section 5 how the groundness information (which may be provided by a specific constraint solver) can be used to reduce the size of the slices constructed by the general technique.
We now formulate the slicing problem for derivation trees. 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 $C_{\mathcal{P}}$.

Definition 68 ***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*.*

The intuition reflected by this definition is that the constraints associated with those positions of the tree not included in a slice do not influence restrictions on the valuation of X imposed by the tree. We formalize this by referring to the formal notion of the slice of a set of constraints. Note that any superset of a slice is also a slice.
Finally we define the notion of a CLP program slice with respect to a variable position. We note that 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 69 ***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* $\Phi_T(Q) \subseteq S$*.*

This means that for any derivation tree position r such that $\Phi_T(r) = q$ and program slice S with respect to q, the value of the variable in r can only be influenced by variants of the program positions in S.

8.3 Dependency-based slicing

The formal definitions of the previous section make it possible to state precisely our objective, which is the automatic construction of slices.
Our formulation defines the slicing of a CLP program in terms of the slicing of sets of constraints. Generally it is undecidable whether a subset of a set of constraints is a slice. This section presents a simple and sufficient condition for this. We provide here a "syntactic" approach to slicing constraint stores, proof trees and programs.

8.3.1 Slicing sets of constraints

We use variable sharing between constraints as a basis for the slicing of sets of constraints.

Definition 70 ***Explicit Dependence***
Let C be a set of constraints, and $vars(C)$ 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. Notice that the explicit dependency relation is symmetric and reflexive but need not be transitive.*

Definition 71 ***Dependency Relation***
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. (The index C may be omitted if the set C is clearly determined by the context.) Notice 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 $[X]_{dep_C}$. From this definition $vars(C - C_X) \cap vars(C_X) = \emptyset$. With this observation and using the definition of a solution of C with respect to X we obtain the following.

Theorem 8 ***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.

Proof. As C_X is a subset of C then any solution of C with respect to X is also a solution of C_X with respect to X.
Let ν be a valuation that satisfies C_X. Then any valuation that coincides with ν on $vars(C_X)$ also satisfies C_X. Consequently any solution of C_X with respect to X is also a solution of C with respect to X since C is satisfiable and constraints in $C - C_X$ have no common variables with constraints in C_X.

Example 14 *For the set of constraints of Example 12 the dependency relation has two equivalence classes: $\{X, Y, U, V\}$ and $\{Z\}$ and gives the following slice of C with respect to X:*

$$C_X = \{X - Y = 1,\ X = U,\ Y = V,\ U + V = 3\}$$

8.3.2 Slicing of derivation trees

We defined the concept of slice for a derivation tree by referring to the notion of slice of a set of constraints. To construct slices of derivation trees we introduce a dependency relation on the positions of a derivation tree then elucidate how it relates to the dependency relation on the constraints of the tree.
Recall that for a given derivation tree T the set $C(T)$ consists of two kinds of constraints:

- constraints of the clause instances that appear in T,
- the node equations of T.

The tree positions of T correspond to occurrences of variables and constants in both kinds of constraints. We now formally define a direct dependency relation $\sim_T$ on $Pos(T)$. It can be related to the dependency relation on $vars(C(T))$ and thus can be used for slicing T.

Definition 72 ***Direct Dependency Relation of a Derivation tree***
Let T be a derivation tree, $\alpha, \beta \in Pos(T)$. 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).*

Notice that the relation is both reflexive and symmetric. 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. Notice that all positions of $\Psi^{-1}(X)$ are equivalent since they share the common variable X.
The following result relates $\sim_T^*$ to $dep_{C(T)}$.

Theorem 9 ***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 $\Psi([\alpha]_{\sim^*}) = [X]_{dep_{C(T)}}$.

Proof.

1. Assume β is a variable position and $\alpha \sim_T^* \beta$. We then show that $dep(\Psi(\alpha), \Psi(\beta))$.

1.a First, examine the case $\alpha \sim_T \beta$.

From the definition four cases are possible.

1. (α, β) is a constraint edge: the constraint of T is then included in $C(T)$, hence $dep(\Psi(\alpha), \Psi(\beta))$.

2. (α, β) is a transition edge: the node equation involves $\Psi(\alpha)$ and $\Psi(\beta)$, hence $dep(\Psi(\alpha), \Psi(\beta))$.

3. (α, β) is a functor edge. Since all nodes of T are complete, $\Psi(\alpha)$ and $\Psi(\beta)$ must occur in a constraint or in a node equation.
Hence $dep(\Psi(\alpha), \Psi(\beta))$.

4. (α, β) is a local edge. Then $\Psi(\alpha) = \Psi(\beta)$, and there are two different constraints (or node equations) in $C(T)$ sharing this variable. Consequently $dep(\Psi(\alpha), \Psi(\beta))$.

Notice that the first three cases imply an explicit dependency relation between corresponding variables, while the local edges describe its transitive closure.
The transitive closure of $\sim_T$ always involves local edges, that is variable sharing between constraints of $C(T)$. Hence the corresponding variables must also be in the $dep_{C(T)}$ relation. A formal proof can be given by performing a simple induction on the number of local edges.

1.b The induction hypothesis is:

$$\forall \alpha, \beta \in Pos(T) : \alpha \sim_T^n \beta \Rightarrow dep(\Psi(\alpha), \psi(\beta)).$$

1.c We have to prove: $\forall \alpha, \beta \in Pos(T) : \alpha \sim_T^{n+1} \beta \Rightarrow dep(\Psi(\alpha), \psi(\beta))$.

$\alpha \sim_T^{n+1} \beta$ = $\alpha \sim_T^n \gamma \sim_T \beta$ fore some $\gamma \in Pos(T)$.

From the induction hypothesis: $dep(\Psi(\alpha), \psi(\gamma))$ and $dep(\Psi(\gamma), \psi(\beta))$. Since $dep_{C(T)}$ is an equivalence relation $dep(\Psi(\alpha), \psi(\beta))$.

2. To conclude the proof we have to examine $dep_{C(T)}$.

It suffices to show that whenever $dep(X, Y)$ then there exist $\alpha \in \Psi^{-1}(X)$ and $\beta \in \Psi^{-1}(Y)$ such that $\alpha \sim^* \beta$.

2.a If X and Y are in the explicit dependency they appear in a constraint or node equation, hence satisfy the above condition. The transitive closure involves variable sharing between distinct constraints of $C(T)$. A formal proof can be carried out by simple induction on the number of constraints involved.

2.b The induction hypothesis is:

If n constraints are included in the constraint store:

$$dep(X, Y) \Rightarrow \text{there exist } \alpha \in \Psi^{-1}(X) \text{ and } \beta \in \Psi^{-1}(Y) \text{ such that } \alpha \sim^* \beta.$$

2.c We have to prove that adding a new constraint to the constraint store:

$$\forall X, Y\ dep(X, Y) \Rightarrow \text{there exist } \alpha \in \Psi^{-1}(X) \text{ and } \beta \in \Psi^{-1}(Y) \text{ such that } \alpha \sim^* \beta.$$

Suppose that we have added a new constraint c_{new} to the constraint store C, getting C'.
Let $X, Y \in var(C')$ and suppose $dep(X, Y)$. We have three cases:

A) X and $Y \in var(c_{new}) \Rightarrow X$ and Y belong to the same constraint or node equation, so there exists $\alpha \in \Psi^{-1}(X)$ and $\beta \in \Psi^{-1}(Y)$ such that $\alpha \sim_T \beta$ (constraint edge) or $\alpha \sim_T \beta$ (transition edge).

B) $X \in var(c_{new})$ and $\neg(Y \in var(c_{new}))$.

In this case there exists a $c_{new} \neq \overline{c} \in C$ such that X is included in $\overline{c}$ and $dep(X, Y)$.
Then, from the induction hypothesis $\exists \alpha \in \Psi^{-1}(X)$ and $\beta \in \Psi^{-1}(Y)$ such that $\alpha \sim_T^* \beta$.

C) Neither X and nor Y is in $var(c_{new})$.

Then X and $Y \in var(C)$, so from the induction hypothesis $\exists \alpha \in \Psi^{-1}(X)$ and $\beta \in \Psi^{-1}(Y)$ such that $\alpha \sim_T^* \beta$.

As a corollary we immediately obtain the following Theorem.

Theorem 10 ***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.

Proof. $\Psi([\alpha]_{\sim^*})$ is an equivalence class of dep, hence it is a slice of $C(T)$.
So a slice of a proof tree can be obtained by finding the equivalence class of the dependency relation involving a given variable position. This applies to derivation trees as well.

Theorem 10 offers suggestions for a simple slicing technique of derivation trees based on finding equivalence classes of variables in the constraints of a derivation tree. The size of the slices obtained in that way is determined by the nature of the dependency relations on the variables of the computation represented by the tree subject to slicing. If the computation consists of several subcomputations not sharing variables the slice obtained will identify one of them. Usually it is not the case, so that this slicing technique is inadequate in practice. In the sequel we show how to improve it by the analysis of directionality of data flow during the computation.

8.3.3 Slicing of CLP programs

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 $\Phi : Pos(T) \rightarrow Pos(P)$.
Having constructed a slice of a derivation tree we can map it to P using Φ. Program fragments identified in this way can be considered **dynamic slices** of the program. The previously discussed techniques for the slicing of derivation trees thus give a foundation for dynamic slicing of CLP programs.
In this section we show yet another approach for the construction of program slices. This approach will be based on a dependency relation which will be defined on the positions of

the program, without regarded to any particular derivation tree. This can be viewed as static slicing.

Definition 73 ***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, 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).*

The dependency relation of a program can be represented as a graph.

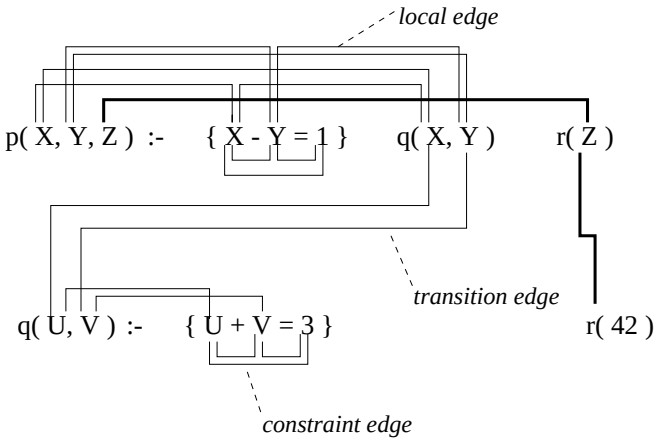

Figure 10: Program dependence represented in graphical form along with the backward slice with respect to Z in p(X,Y,Z))

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 $Pos(P)$. The following result shows how $\sim_P^*$ can be used for the slicing of P.

Theorem 11 ***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.

Proof. We refer to the definition of a program slice. We have to show that for every proof tree T if $\alpha \in \Phi_T^{-1}(\beta)$ there exists a slice Q of T with respect to α such that $\Phi(Q) \subseteq [\beta]_{\sim_P^*}$. From the definition of $\sim_P$ we have that for every T: $\Phi([\alpha]_{\sim_T^*}) \subseteq [\Phi(\alpha)]_{\sim_P^*}$. But $[\alpha]_{\sim_T^*}$ is a slice of T with respect to α. Hence if $\alpha \in \Phi_T^{-1}(\beta)$, then $[\Phi(\alpha)]_{\sim_P^*} = [\beta]_{\sim_P^*}$ is a slice of P.

Figure 10 shows the program dependences and a backward slice of the program in Example 1 with respect to Z in $p(X, Y, Z)$.
For the program in Example 12 $\sim_P^*$ has two equivalence classes. One of them includes all occurrences of Z and the occurrence of the constant 42. The other contains the remaining positions.
Definition 73 with Theorem 11 gives a method for constructing program slices without referring to proof trees. Thus we obtain a **static slicing** technique for CLP. The mapping Φ was introduced to formally describe the correctness of this technique. These results also confirm the correctness of the slicing algorithm in [23] because a logic program can be viewed as a constraint logic program.

8.3.4 Discussion

We have shown that variants of the notion of dependency relation provide sufficient conditions for (static and dynamic) slicing of CLP programs. However, in practice, the slices obtained are often quite large; sometimes they may even include the whole program. We propose two ways of improving the techniques presented in the following two sections. The first way is to take into account groundness information, which makes it possible to introduce directional dependency relations reflecting dataflow during the execution of the program. This approach is applicable both for dynamic slicing and for static slicing. Groundness propagation in a particular derivation tree can be precisely monitored by tracing the execution. In the case of static slicing groundness information can be inferred via static analysis of the program.
For some programs the results of groundness analysis may be insufficient to reduce the size of static slices. This is because a static slice must take into account all possible executions of the program, and static analysis is often imprecise. Therefore, dynamic slicing is preferable for debugging, while static slicing may facilitate a general understanding of the program and assist users in program maintenance.
The second way of improving the techniques to be discussed concerns static slicing. The size of slices may be reduced if we handle the so called "calling context problem"[30], which appears when the same predicate is called from two different clauses. It is possible to adapt to CLP the solution proposed by Horwitz et. al [30] for procedural languages.

8.4 Directional slicing

All the dependency relations discussed so far were symmetric. Intuitively, for a constraint $c(X, Y)$ a restriction imposed on valuations of X usually influences admissible valuations of Y and vice versa. However if $c(X, Y)$ belongs to a satisfiable constraint set C and some other constraints of C make X ground then the slice of C with respect to X need not include $c(X, Y)$. For example, if $C = \{X + 1 = 0, Y > X\}$ is interpreted on the integer domain then $\{X + 1 = 0\}$ is a slice of C with respect to X. This slice can be constructed using information about the groundness of variables occurring in the dependency graph.
This section tells us how to use groundness information in derivation tree slicing, that is in the dynamic slicing of CLP programs. It extends the ideas of dynamic slicing for logic programs presented in [23] to CLP. In our approach groundness is captured by adding directionality information to dependency graphs. The directed graphs show the propagation

of ground data during the execution of CLP programs, and these graphs can then be used to produce more precise slices. Groundness information is collected during computations that construct the derivation tree to be sliced.

8.4.1 Groundness Annotations

Groundness information associated with a derivation tree will be expressed as an annotation of its positions. The annotation classifies the positions of a derivation tree. A position is classified as *inherited* (marked with $\downarrow$), *synthesized* ($\uparrow$) or *dual* ($\updownarrow$). Formally speaking, an annotation is a mapping μ from the positions into the set $\{\downarrow, \uparrow, \updownarrow\}$ [14].
The intended meaning of the annotation is as follows. An inherited position is a position which is ground at time of calling, that is when the equation involving this position is first created during the construction of the derivation tree. A synthesized position is a position which is ground at success, that is when the subtree having the position in its root label is completed in the computation process. The dual positions of a proof tree are those which are ground neither at call nor at success.
The above-mentioned annotations are collected during the execution of the program.
We will now introduce the following auxiliary terminology relevant to the annotated positions of a proof tree. The inherited positions of the head atoms and the synthesized positions of the body atoms are called *an position*. Similarly, the synthesized positions of the head atoms and inherited positions of the body atoms are called *an output position*. Note that dual positions are not strictly classified as input or output ones. Alternatively, if we say that a position is annotated as an output we mean that it is annotated as inherited provided it is a position in a body atom, or annotated as synthesized if it is a position of the head of a clause.

Example 15 *Consider the following CLP program:*

```
1. p(X,Y) :- r(X), q(X,Y).
2. r(3).
3. q(U,V) :- {U+V = 5}.
```

The corresponding annotated proof tree for the goal $p(X, Y)$ is presented in Figure 11, where the actual positions have been replaced by I (the input positions) and by O (the output positions):

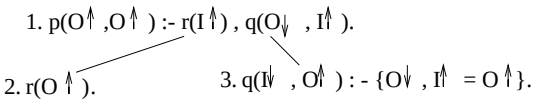

Figure 11: The annotated proof tree for Example 15

As discussed below the annotation reflects groundness propagation during the computation. The variables X and Y of $p(X, Y)$ are annotated as output as they are ground at success of p, and $p(X, Y)$ is a head atom. For the same reason the argument of the fact $r(3)$ in node 2 is annotated as output. The variable U in the first argument of the predicate $q(U, V)$ in node 3 is ground at call so it is annotated as input, while V is ground at success of q so it is annotated as output.

In a CLP program the groundness of a position depends in general on the employed constraint solver.
Monitoring the execution, we are able to annotate certain positions as inputs or outputs. For example the $indomain(X)$ constraint of CHIP instantiates X to a value in its domain, so that X is an input position. Rational solvers can usually solve linear equations. For example in the rational constraint
$Y = 2 * X + Z$, X can be annotated as input if Y and Z are output.

8.4.2 Directional slicing of derivation trees

Having input/output information for the positions of a derivation tree, we then add directions to the corresponding dependence graph. The following definition describes how it can be achieved.

Definition 74 Directed Dependency Graph of a Proof Tree
Let T be a proof tree, $T_G = (Pos(T), \sim_T)$ its proof tree dependence graph, then the directed dependence graph of $T(P)$ can be defined as:
$T_{DG} = (Pos(T), \rightarrow_T)$*, where:*

- $\alpha \rightarrow_{T(G)} \beta$ *if* $\alpha \sim_{T(G)} \beta$ *is a transition edge,* α *is an output position and* β *is an input position*
- $\alpha \rightarrow_{T(G)} \beta$ *if* $\alpha \sim_{T(G)} \beta$ *is a local edge,* α *is an input position and* β *is an output position*
- $\alpha \rightarrow_{T(G)} \beta$ *and* $\beta \rightarrow_{T(G)} \alpha$ *in every other case when* $\alpha \sim_{T(G)} \beta$

From the definition of $\rightarrow_{T(G)}$ assuming correctness of the annotation we see that if α is a position of X in T and it is annotated as input or output then $C(T)$ binds X to a single value. This value is determined by the constraints associated with those positions that are in the set $\{\beta | \beta \rightarrow^*_{T(G)} \alpha\}$. Hence we have:

Theorem 12 ***Directed Slice of a Proof Tree***
$\{\beta | \beta \rightarrow^*_{T(G)} \alpha\}$ *is a slice of T with respect to α.*

Proof
The correctness of this theorem is based on the fact that we did not eliminate edges from the undirected proof tree dependence graph; we only directed them according to the realizable data flow paths.
Formally speaking, Theorem 10 says that $\{\beta | \beta \sim^*_T \alpha\}$ is a slice of T with respect to α. Following the points of Definition 74 :

- there is no realizable data flow along a transition edge from an input position to an output position.
- there is no realizable data flow along a local edge from an output position to an input position.
- in every other case the data can flow in both directions along the directed edges

we get that we did not eliminate edges according to the realizable data flow pathes, so $\{\beta|\beta \rightarrow^*_{T(G)} \alpha\}$ is a slice of T with respect to α.

These slices are usually more precise than in the case when groundness information is not employed.
The concept of a directed dependency graph can be extended to programs and used for static slicing. Good methods are required for static groundness analysis to infer annotations for program positions. One possibility is to rely on groundness information specified by the user in some CLP languages, see for instance the mode declarations of Mercury [71].

8.5 Using the calling context

This section outlines yet another technique for improving the precision of slicing. It is an adaptation of a technique proposed in [30] for imperative programming. The technique has not yet been implemented in our slicer. The objective of this section is only to show that the *calling context problem* known in imperative programming appears also in CLP and to sketch informally an approach to its solution.
First we illustrate in a simple example a common source of imprecision introduced by basic static slicing, which is known as the *calling context problem*. Then we show how to improve slicing precision via a simple analysis. Finally, we illustrate this idea with a CLP example.

Example 16 *Consider the following logic programming example:*

```
1. l(X,Y) :- p(X,Y).
2. p(X,Y) :- r(X,Y), s(X).
3. q(U,W) :- r(U,W).
4. r(X,Y) :- X is Y+1.
5. s(X).
```

The program dependency graph for this program is shown in Figure 12. A slice of the program with respect to the variable in $s(X)$ of clause 5 obtained using the basic dependency-based slicing contains the whole program, so includes the positions of the atom $q(U, W)$ as well. These positions could safely be removed from the slice but the basic method does not do that. The reason is that the predicate r is called in two different bodies but the relation $\sim_P$ does not care about the calling context. So the slice obtained is based on dependencies in nonrealizable paths. This situation can appear quite often in real programs as predicates are often used independently in several parts of the program.
To solve this problem for procedural languages a two-pass slice computation on the program dependency graph was suggested by Horwitz et al [30]. The basic idea is that in the first pass we do not move down into the called procedures, but only up to the callers, then we mark the slice points in the caller procedures. In the second pass we start from these slice points and move down to the called procedures, but do not move up to the callers. In the first pass we also have to consider the effects of procedure calls. To do this we have to compute the so-called *summary edges* in the body atoms of the clauses. In our example in the clause i the body atom $r(X, Y)$ gives rise to the summary edge $Y - X$. This edge is introduced to show

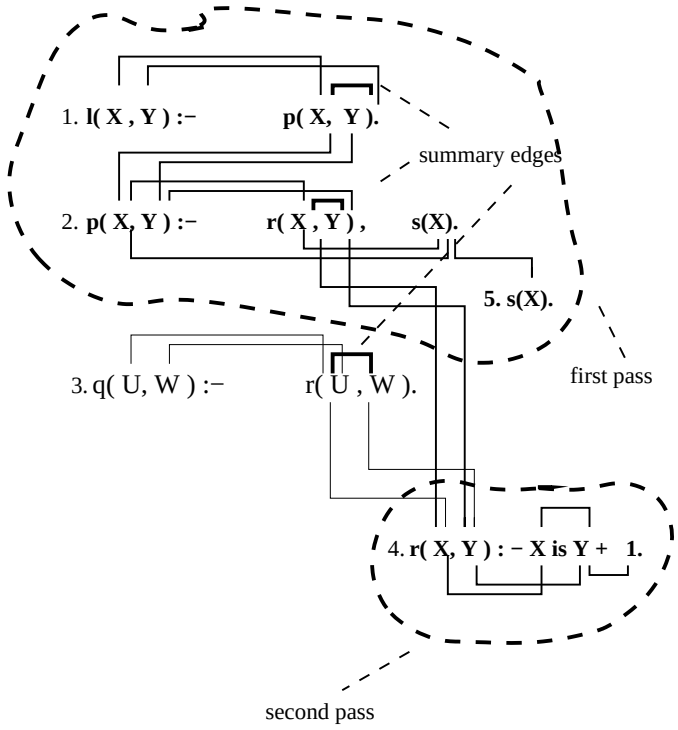

Figure 12: A two-pass static slice computation for the variable s(X)

that calling $r(X, Y)$ leads to a dependency between the variables. For procedural languages efficient algorithms for the computation of summary edges have been suggested in [30] and [20]. Firstly, the interprocedural control and data dependency edges are constructed, then the summary edges can be computed by using these interprocedural dependency graphs.
In CLP programs, predicates correspond to procedures and the intraprocedural dependency graphs can be constructed using the constraint, functor and local edges (Definition 73). On the basis of these dependency graphs and the transition edge the summary edges can be computed.
The implementation of the two-pass static slicing method for CLP programs is one of the tasks of future work.
Figure 12 shows which program positions are added to the slice (with respect to X in $s(X)$ of clause 5) of the example program in the first and second passes (framed by broken lines). The summary edges are also added to the original program dependence graph. We see that the variables in clause 2 are added to the slice in the first pass and the variables of clause 4 in the second pass. The variables of clause 3 do not appear in the slice, which is an improvement on the method previously referred to.
The calling context problem does not appear in slicing of proof trees. The reason for this is that each proof tree describes a real calling sequence with renamed clause instances. This is in contrast with the program dependency graph where calling information is not present and all potential instances are represented by one clause. Overall this means that the two-pass program slicing method is sound since it eliminates, from the slice obtained by the

dependency method only those positions which do not correspond to real proof trees.
We will now illustrate the two-pass slicing method with a simple CLP example.

Example 17 *Consider the following constraint logic program:*

```
1. prg(L1,Sum,L2,DSum) :- p(L1,Sum), q(L2,DSum).
2. p(L1,Sum) :- sum(L1,Sum), control(Sum).
3. control(Sum) :- {Sum <= 500}.
4. q(L2,DSum) :- sum(L2,Sum2), {DSum=2*Sum2}.
5. sum([],0).
6. sum([X|Rem],Sum) :- sum(Rem,Sum1),{Sum=X+Sum1}.
```

Figure 13 shows the modified program dependency graph for this example and the two-pass static slice for the variable $Sum2$ in clause 4. The predicate sum is called by two different predicates (p and q). The original slice with respect to $Sum2$ contains the whole program. For example it contains clause 2, which need not be included in the slice. The two-pass slice, framed by broken lines, does not contain them.

8.6 A Prototype Implementation

We developed a prototype in SICStus Prolog for the backward slicing of constraint logic programs written in SICStus. The tool handles a realistic subset of Prolog. The *inputs* of the slicing system are:

- the source code,
- a test case (a goal)
- (following the execution) the execution traces given by the Prolog runtime system.

From this information the *Directed Proof Tree Dependence Graph* (Definition 39) may be constructed. The following four types of slice algorithms were implemented (see Figure 14):

1. **Proof tree slice**
   In this case the user chooses an argument position of the created proof tree, and the slice is constructed with respect to this proof tree position using the Directed Proof Tree Dependency Graph. This kind of slice is useful when the user is interested in the data dependences of the proof tree.

2. **Dynamic slice**
   This case is very similar to 1, but the constructed slice of the proof tree is mapped back to the program. This is the classic dynamic slice approach [40], as in the case of procedural languages. So this slice provides a slice of the program.

3. **Program position slice**
   In this case the user selects a program position. The system provides all instances of this program position in the proof tree, creates the proof tree slice for every instance, then the union of these slices is constructed and mapped back to the program. So this algorithm also provides a slice of the program which shows all dependences of a program position for a given test case.

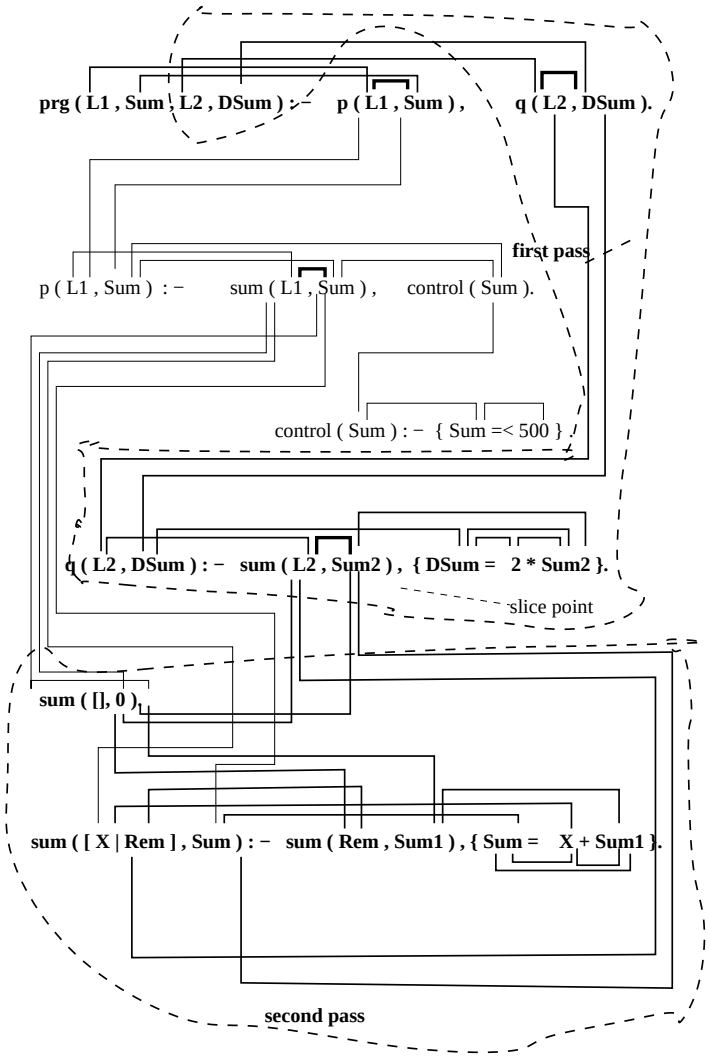

Figure 13: The Program Dependency Graph and two-pass static slice for the variable Sum2

4. **Static slice**
   This kind of slice does not refer to a particular proof tree, but concerns all executions. The user selects a program position as above but, in contrast to the previous case, the slicing is based on a program dependency graph (see Definition 73) and not on the analysis of a particular proof tree.

We can apply the *Proof tree slice* method if we have an efficient tool for the visualization of the proof tree (for example, in the case of debugging). When the *Dynamic slice* algorithm is used no additional tool is needed, since the slice is highlighted in the program text, and can be used for “conventional” debugging.
The *Static slice* incorporates all the possible dependences of a program position, therefore this slice is useful in program understanding, testing and maintenance.
The static slices may be imprecise (too large) in many cases, hence the computation of the *Program Position slice* for several test cases and using this slice to estimate the possible

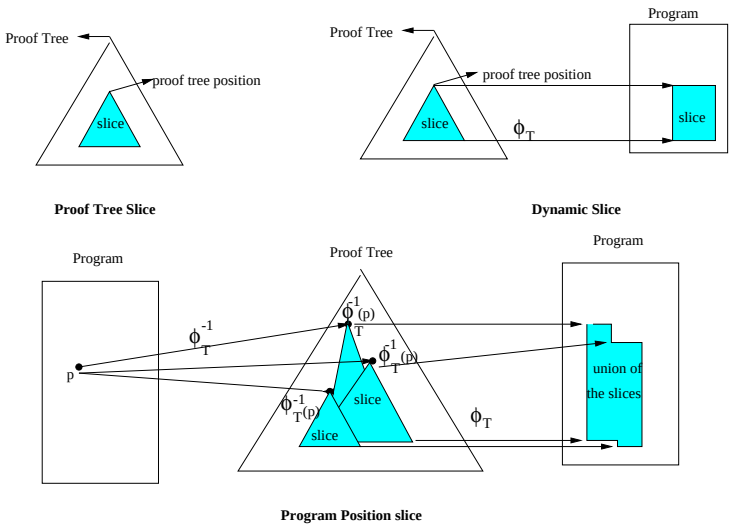

Figure 14: Proof Tree, Static, Dynamic, and Program Position Slice.

dependences may be useful.

Example 18 *This example shows how dynamic slicing can be used to locate a bug in the modified program of Example 17.*
Let suppose that we have an error in clause 6, such that $\{Sum = X - Sum1\}$ *is typed instead of* $\{Sum = X + Sum1\}$.
When running the program we obtain wrong values for Sum *and* $Dsum$. *As the wrong result obtained is related to the variable in the main procedure of the program, which is* p, *a natural impulse is to create a slice of the proof tree of the computation, or dynamic slice with respect to* Sum *in the head of clause 1. This slice is shown in Figure 15. It focuses the search for the bug on the highlighted position including the erroneous constraint* $\{Sum = X - Sum1\}$. *The remaining question, not addressed in this work, is how to organize this search.*

A graphical interface draws the proof tree (see Figure 15), marked with different colored nodes that are in the proof tree slice and for a dynamic slice and program position slice, the corresponding slice of the program is highlighted. The labels of the nodes identify the nodes of the proof tree, including the name of the predicate and annotation of its arguments.
In the implementation we applied a very simple annotation technique: the inherited positions were those which were ground at the time of calling, while synthesized positions were those which were ground at success. This method provides a precise annotation because we continuously extract information from the actual state of the constraint store. However, in the current implementation slicing can only be done on argument positions. If an argument includes several variables it is not possible to distinguish between them, which makes the constructed slice ”less precise” (compared to the minimal slice). So one possible aim of the future work is to improve the existing implementation by extending it to variable positions.
In the present version of the tool the sliced proof tree corresponds to the first success branch of the SLD tree. As the proof tree slice definition is quite general, there is no real difficulty in applying the technique to all success branches of the SLD tree. The extension to failure

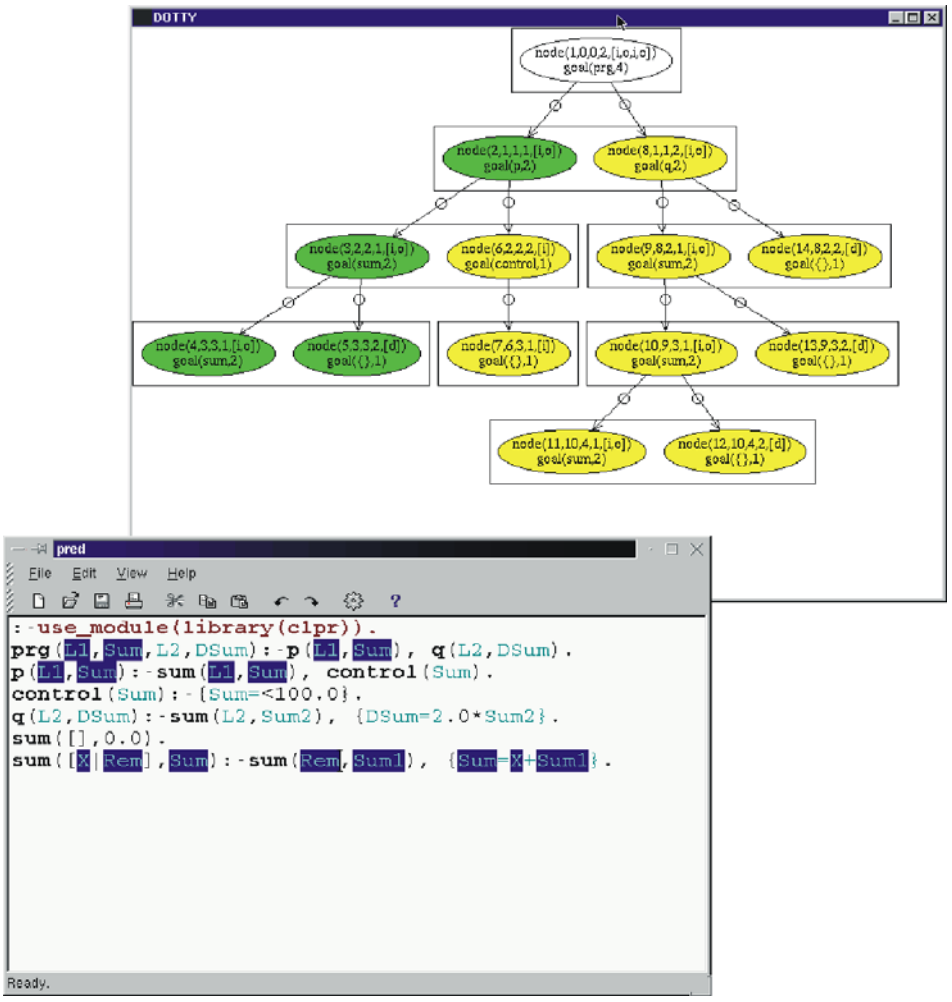

Figure 15: Displayed program and proof tree slices.

branches is discussed in [78] which makes it possible to create a slicer of an unsatisfiable derivation tree (due to incompleteness of the constraint solver) as well.

Our definition of the derivation tree assumes that the skeleton equations are interpreted over a specific constraint domain, and solved accordingly by a domain specific constraint solver. However, in most CLP languages, including SICStus Prolog, these equations are instead subject to syntactic unification, and a domain-specific constraint solver is applied only to the constraints in clause bodies. This fact was taken into account in our tool, which can only slice the derivation trees constructed via syntactic unification of the skeleton equations.

Systematic slicing experiments were performed on a number of small constraint logic programs (written in SICStus Prolog). Even if the size of the programs is small, the size of the constructed proof trees can be large (up to 10,000 nodes). Our present implementation has not been optimalized yet, so the computation of a proof tree slice (up to 5000 nodes) on a PIII machines needs about 1-2 minutes.

Each of the test programs was executed with a number of test inputs to collect data about the relative size of a slice with respect to the proof tree, depending on the choice of the position.

The selected application programs [32, 46] had different language structures (use of cut, or, if-then, databases, compound constraint).
A summary of the test results obtained from proof tree slicing is listed in *Table 8.6*. A comparison of the four kinds of slices with respect to the number of nodes is shown in Figure 16.

PROGRAM	NUMBER OF CLAUSES	NUMBER OF TEST CASES	NUM. OF COMPUTED SLICES	AVERAGE THE PROOF	SIZE OF TREE	AVERAGE SIZE OF SLICES
				NODE	ARG. POS.	
FIB	4	6	1349	399.57	533.51	7.7 %
CIRC	5	2	139	33.35	54.09	9.41 %
SUM	6	4	626	80.26	120.77	7.89 %
LIGHTMEAL	11	1	17	9	15	29.61 %
SCHEDULING	20	1	575	134	390	51.21 %
PUZZLE	30	2	1363	227.57	607.82	19.05 %

Table 2: Test Results of The Proof Tree Slice Algorithm

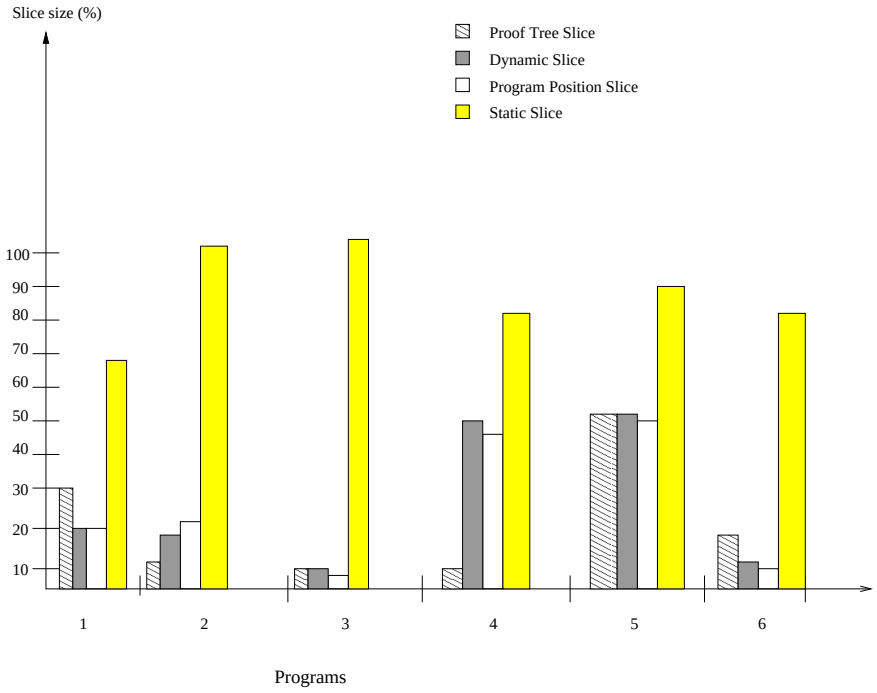

Figure 16: Comparison of the Proof Tree Slice, Dynamic Slice, Program Position Slice and Static Slice.

It should be mentioned that the average slice size (in $\%$) in these experiments had no correlation with the size of the program. The average slice size was 29% of the number of executed nodes. In the case of static program slicing the average slice size was 87% (without using any groundness information, and handling the calling context problem). The reason for this average size of slices was because small-sized programs were used.
Figure 17 shows the slice size distribution of the six examples above. The slice size distribution was counted in the following way:

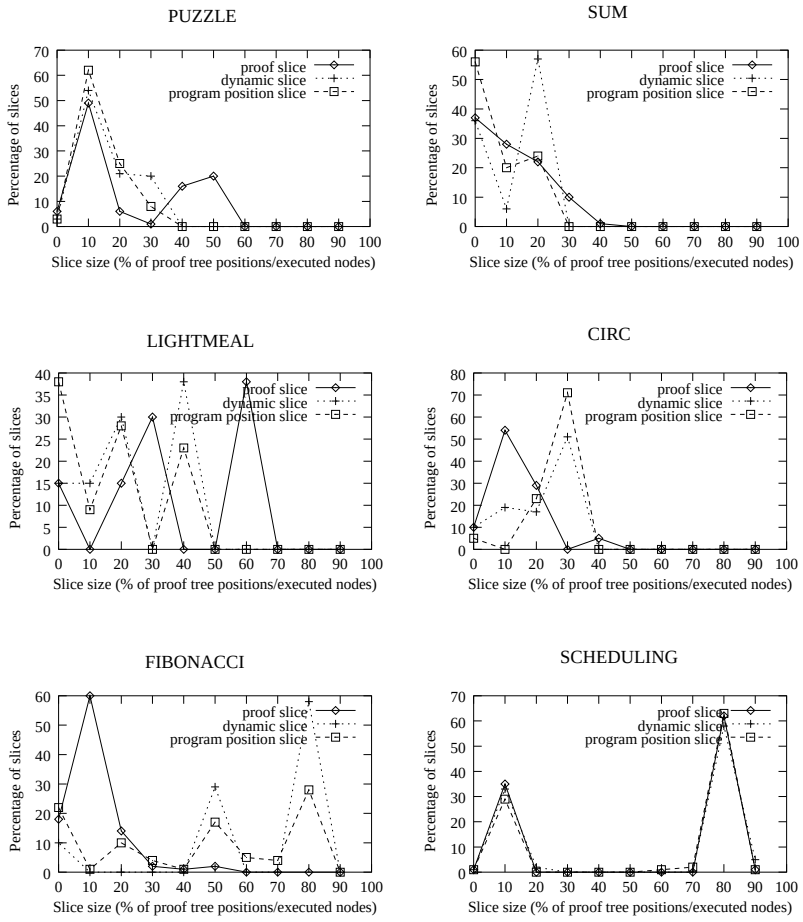

Figure 17: Slice Size distribution.

Let n denote the number of constucted slices and let
- the number of slices whose size is between 0-5 $\%$ be n_1
. . . . . .
- the number of slices whose size is between 95-100 $\%$ be n_{20}
Then the slice size distribution for the i-th class ($i = 1, \cdots 20$) is: $n_i/n * 100$.

The programs were executed for different inputs, and the slices were constructed with respect to different proof tree (program) positions.
The computation time is in ratio to the size of the proof tree or program.
The intended application of our slicing method is to support the debugging of constraint programs. A bug in a program shows up as a symptom during the execution of the program on some input. This means that in some computation step (i.e. in some derivation tree) a variable of the program is bound in a way that does not conform to user expectations. A slice of the derivation tree with respect to this occurrence of the variable can be mapped into the text of the program and will identify a part of the program where the undesired binding was produced. This would provide a focus for debugging, whatever debugging technique is

used. The intersection of the slices produced from different runs of the buggy program may further narrow the focus.
In particular, our slicing technique can be combined with Shapiro's algorithmic debugging [69], designed originally for logic programs.
As shown in [39] the slice of a derivation tree of a logic program may reduce the number of queries of the algorithmic debugger. This technique can also be applied to the algorithmic debugging of CLP programs.

8.7 Related Work

Program slicing has been widely studied for imperative programs [21, 30, 29, 34, 40]. To our knowledge only a few papers have dealt with the problem of slicing logic programs, and the slicing of constraint logic programs has not been investigated till now.
We provided a theoretical basis for the slicing of proof trees and programs starting from a semantic definition of the constraint set dependence. This applies as well to the special case of logic programs (the Herbrand domain). In particular it justifies the slicing technique of Gyimóthy and Paakki [23], developed to reduce the number of queries for an algorithmic debugger and enables us to extend the latter to the general case of CLP. The directional slicing of Section 8.4 is an extension of this technique to the general case of CLP.
Schoening and Ducassé [66] proposed a backward slicing algorithm for Prolog which produces executable slices. For a target Prolog program they defined a slicing criterion which consists of a goal of P and a set of argument positions P_a along with a slice S as a reduced and executable program derived from P. An executable slice is usually less precise but it may be used for additional test runs. Hence the objectives of their work are somewhat different from ours, and their algorithm is only applicable to a limited subset of Prolog programs.
In [93] Zhao et al defined some static and dynamic slices of concurrent logic programs called literal dependence nets. They presented a new program representation called the argument dependence net for concurrent logic programs to produce static slices at the argument level. There are some similarities between our slicing techniques and Zhao's as we also rely on some dependency relations. However, the focus of our work has been on CLP not on concurrent logic programs, and our main aim has been to find a declarative formulation of the slicing problem which provides a clear basis for proving the correctness of the proposed slicing methods.
The results of the ESPRIT Project DiSCiPl [13] show the importance of visualisation in the debugging of constraint programs. Our tool provides a rudimentary visualisation of the sliced proof tree. It was pointed out by Deransart and Aillaud [13] that abstraction techniques are needed in the visualisation of the search space. Program position slicing, if applied to all branches of the SLD-tree, provides yet another abstraction of the search space.

8.8 Conclusions

This part of the thesis offers a precise declarative formulation of the slicing problem for CLP. It also provides a basis for deriving various techniques of slicing CLP programs in general and for logic programs treated as a special case. A data flow based slicing technique [23] was used for the construction of static slices of CLP programs and dynamic slices of proof trees.

Two approaches were proposed for reducing the size of the slices: introducing directionality information (which uses groundness information), and the handling of the calling context problem.
We also presented a prototype slicing tool 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 of the program.
One possible avenue of future research is the application of slicing techniques to constraint programs debugging. Two aspects will need to be considered. Firstly, in the manual debugging of CLP programs it is necessary to support the user with a tool that facilitates the understanding of the program. Secondly, the slicing of a derivation tree can often reduce the number of queries in the algorithmic debugging of logic programs [69].

Part III
LEARNING OF CONSTRAINT LOGIC PROGRAMS

The results of this part of the thesis are covered in [79].

9 The Learning of Constraint Logic Programs

This part of the thesis discusses the learning of Constraint Logic Programs using unfolding and combining unfolding with a slicing technique. Constraint Logic Programming (CLP) is an emerging software technology with a growing number of applications. The transformation rule for unfolding together with clause removal is a method (called SPECTRE) for the specialization of Logic Programs (LP).
Logic programming can be regarded as a special case of constraint logic programming, CLP extends logic programs with constraints. This is a substantial extension, so CLP not only has syntactics distinct from LP but also distinct semantics.
Slicing is a program analysis technique originally developed for imperative languages. It facilitates the understanding of data flow and debugging.
This paper presents a learning method for constraint logic programs by combining unfolding and clause removal, formulates the semantics of this new method, proves that the unfolding transformation preserves the operational and logical semantics, and improves the unfolding technique by applying slicing.
A prototype learner of CLP programs which implements the above ideas is briefly described.

9.1 Introduction

Constraint Logic Programming (CLP) [32, 46] extends logic programs with constraints. It is an emerging software technology with a growing number of applications, it has the power to tackle those difficult combinatorial problems encountered for instance in job scheduling, timetabling, and routing which stretch conventional programming techniques to their limit.

Inductive Logic Programming (ILP) [49] 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 representation language makes ILP systems more powerful and useful than the conventional propositional machine learning systems, but they are weak in handling numerical data. Although some existing ILP systems are capable of handling numerical data, the approach in most cases is ad-hoc.

CLP languages make logic programs perform very efficiently by focussing on a particular problem domain, they have their are good at handling numerical information, as well.

Inductive Constraint Logic Programming (ICLP) is a research topic on the intersection of *Constraint Logic Programming (CLP)* and *Inductive Logic Programming (ILP)*.

In this section we present an inductive learning algorithm (ICLP) based on the specialization of Constraint Logic Programs, formulate the semantics of this method, state the associated definitions and theorems, and prove that the specialization operator preserves the operational and logical semantics of constraint logic programs.

In the case of inductive learning from examples, the learner is given some examples from which general rules or a theory underlying the examples can be derived.

More formally, *the learning setting* in the case of ICLP is the following:
The teacher selects a *target concept* and provides the learner with a finite, nonempty *training set of examples*, each of which is correctly labelled either as *positive* or *negative*. From this training sample and any available background knowledge, the learner constructs a hypothesis of the concept. Examples are ordinary atoms built over a *target* predicate and the background knowledge is a finite set of constrained clauses. A hypothesis is a finite set of nonrecursive constrained clauses whose heads are ordinary atoms built over the target predicate and whose bodies consist of literals built over predicate symbols defined in the background knowledge or constraint symbols.
The algorithm SPECTRE [7] is an ILP method (learning method of Logic Programs) which takes as its input a definite program and two sets of atoms (positive and negative examples). The output of the algorithm is a new definite program that covers all positive examples but no negative ones.
The SPECTRE algorithm specializes clauses defining a target predicate by applying different strategies for selecting the literal to apply unfolding upon (e.g. taking the leftmost, selecting randomly or using the impurity measure).
The main idea behind the IMPUT method [2] is that the identification of a clause to be unfolded plays a crucial role in the effectiveness of the specialization process. If a negative example is covered by the current version of the initial program it is supposed that there is at least one clause that is responsible for this incorrect covering. The IMPUT system uses a debugging algorithm to identify a buggy clause instance which is then unfolded and partially removed from the initial program.
The IMPUT system improves the original SPECTRE algorithm but this solution has one major drawback, namely that an oracle has to answer membership questions to identify a buggy clause instance.
In our prototype the algorithmic debugger is combined with a slicing technique. Slicing makes it possible to reduce the number of user queries during the debugging process. During the slicing a proof tree is produced for a buggy program, then a proof tree dependence graph is constructed and 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 focusing on the suspect part of the program.
The idea of using slicing was introduced in [2], but they did not carry out a safe implementation.
The SPECTRE and IMPUT algorithms have been worked out for Logic Programs. Logic Programming (LP) can be viewed as a special case of Constraint Logic Programming (CLP). CLP extends Logic Programs with constraints. This is a substantial extension, so CLP not only has syntactics distinct from LP but also distinct semantics.
The main contributions of this work are the following.
Firstly, we will formulate the exact definitions of the basic elements of an unfolding learning algorithm such that SLD-tree, specialization and unfolding transformation of CLP programs giving the semantics of them. We will prove that the 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 which integrates an algorithmic debugging algorithm and the slicing method with the specialization algorithm for CLP programs.
A prototype tool has been implemented for both algorithms. The novelity of this implementation is also the combination of the specialization algorithm with a slicing tool for Logic Programs.

This part is organized as follows. Section 9.2 outlines some basic concepts (machine learning, SPECTRE) which are then used to formulate and analyze the specialization methods. Section 9.3.1 presents our main results - the CLP_SPEC algorithm, the associated definitions (specialization, unfolding) and theorems (operational and logical semantics preservation, correctness analysis). Section 9.4 discusses an improved interactive version of the CLP_SPEC algorithm (called CLP_SPEC_SLICE) which combines the unfolding technique with algorithmic debugging and slicing. Our prototype tool is described in Section 9.5. Section 9.6 provides a comparison with other works and suggestions for future work, and finally in Section 9.7 a summary is given.

9.2 Preliminaries

In Section 7 we formalized some basic concepts of Constraint Logic Programming. In the following a short overview of Machine Learning and the SPECTRE algorithm is given.

9.2.1 Machine learning

In the case of inductive learning from examples, the learner is given some examples from which general rules or a theory underlying the examples can be derived. An inductive concept learner is given by a set of training examples, some background knowledge (that needs to be refined), a hypothesis description language, and an oracle willing to answer questions (in the case of interactive learners). The aim is to find a hypothesis such that the hypothesis is complete and consistent with respect to the examples and background knowledge.
To express efficiently the examples, the background knowledge, and the hypothesis to be induced we need to use a language L with sufficient expressive power.

9.2.2 The SPECTRE algorithm

The inductive learning techniques of logic programs (Inductive logic Programming (ILP)) use the first order predicate logic language as the hypothesis and example description language (L). In the case of Constraint Logic Programs(ICLP) the first order language is extended with constraints.

The problem of finding an inductive hypothesis by specializing a logic program with respect to positive and negative examples can be regarded as the problem of pruning an SLD-tree such that all refutations of negative examples and no refutations of positive examples are excluded [7]. It has been shown that the actual pruning can be performed by applying unfolding and clause removal. The algorithm SPECTRE [7] is based on this idea. The algorithm uses

different strategies for the top-down induction of logic programs.
The algorithm SPECTRE accepts a definite program as input, two sets of atoms (positive and negative examples) and a computation rule. The output of the algorithm is a new definite program that covers all positive but no negative examples. The algorithm consists of one main loop that continues until no negative examples are covered. When a clause is found that covers a negative example, and no positive examples, it is then removed. When a clause is found that covers both a negative and a positive example, it is unfolded. The choice of which literal to unfold is decided using the computation rule.
The generality of the resulting specialization depends on the actual computation rule, and hence the choice of computation rule is crucial for the good performance of the algorithm. The computation rule should be formulated such that the number of clauses that are needed to distinguish between positive and negative examples is kept to a minimum. This means that the number of applications of unfolding should be kept as low as possible, since the number of clauses increases when unfolding is applied.
An improvement of the specialization process is achieved when a debugger and slicer with SPECTRE are combined.

9.3 Specialization of CLP Programs by Unfolding

This section presents our main result, that of the CLP_SPEC algorithm, along with associated definitions and theorems.
The algorithm CLP_SPEC specializes logic programs with respect to positive and negative examples by applying the transformation rule **unfolding** together with **clause removal**. After providing the exact definitions of the specialization problem and unfolding transformation we prove that the unfolding transformation of constraint logic programs preserves the operational (Theorem 14) and logical semantics (Theorem 15). It is also shown that pruning an SLD-tree (such that all refutations of negative examples and no refutations of positive examples are excluded) can be performed by combining unfolding and clause removal.
Finally, the algorithm CLP_SPEC is formulated and the correctness of the algorithm is proved.

9.3.1 The specialization problem

Definition 75 The specialization problem
The specialization problem of a Constraint Logic Program with respect to a set of positive (E^+) and a set of negative examples (E^-) can be defined by the following:
Given: *a P Constraint Logic Program and two disjunct 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 note that this definition satisfies the conditions of completeness and consistency of a hypothesis since the specialized program covers all positive examples and does not cover any negative example.

In this work we assume that every positive and negative example is an ground instance of a target (defined) predicate G (goal). SLD-refutations of all the examples are then included in an SLD-tree of $P \cup G$ (every $e \in (E^+ \cup E^-) : e \in lm(P, \Im)$).
This means that for each SLD-refutation of a particular example, there is a branch in the corresponding SLD-tree leading from the root to the empty goal.

Consider the following example.

Example 19 *In the following constraint logic program the equality constraint and symbols of arithmetic operations are interpreted over the domain of rational numbers. A simple example has been chosen to simplify the illustration of the specialization algorithm shown later on. The constraints are distinguished by the use of curly brackets* $\{\}$.

Given the definition of a fish-meal as consisting of an appetiser, a main meal and a dessert and a database of foods and their calorific values we wish to construct light fish-meals i.e. fish-meals whose sum of calorific values does not exceed 10. This program needs to be specialized since it doesn't just cover fish-meals.

```
1.  fishlightmeal(A,M) : - {I + J ≤ 10}, appetiser(A,I),
main(M,J).
2.  appetiser(A,I) : - cheese(A,I), {I > 0}.
3.  appetiser(A,I) : - pasta(A,I), {I > 0}.
4.  main(M,J) : - fish(M,J), {J > 0}.
5.  main(M,J) : - meat(M,J), {J > 0}.
6.  fish(sole,2).
7.  fish(tuna,4).
8.  meat(beef,5).
9.  meat(chicken,4).
10.  pasta(general,1).
11.  cheese(camamber,2).
```

In Figure 18 (see Section 9.3.5), the skeleton of the SLD-tree of Example 19 is shown whose leaves corresponding to refutations of positive and negative examples are labelled '+' and '-' respectively, while leaves that do not correspond to refutations of any examples are left unlabelled (The corresponding set of positive and negative examples are also given in Section 9.3.5). The broken line shows two places where the SLD-tree can be pruned such that all refutations of the negative examples and no refutations of the positive examples are excluded.

9.3.2 Unfolding

Following our formulation of the unfolding transformation, two theorems are proved which state that the defined unfolding transformation preserves the operational and logical semantics of CLP programs.

Definition 76 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 renamed to new variables 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 :

$$P' = Unf(P, R, b_m) = \text{argument equations} \qquad body(R_j)$$
$$= P \setminus \{R\} \cup (\bigcup_{Rj \in \overline{R}} h \leftarrow \overbrace{(\overline{b}_m = \overline{h}_j)}, b_1, \cdots, b_{m-1}, \overbrace{b_{j_1}, \cdots, b_{j_{m_j}}, c_j}, b_{m+1}, \cdots, b_k, c),$$

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 a defined predicate can be unfolded and no constraint predicates.
In the formalization of the algorithm CLP_SPEC we will use the following set:

$$Res(P, R, b) := \bigcup_{Rj \in \overline{R}} (h \leftarrow \overbrace{(\overline{b}_m = \overline{h}_j)}^{\textit{argument equations}}, b_1, \cdots, b_{m-1}, \overbrace{b_{j_1}, \cdots, b_{j_{m_j}}, c_j}^{body(R_j)}, b_{m+1}, \cdots, b_k, c)$$

This set can be viewed as the set of "resolvents" of R.

In Example 20 (see Section 9.3.5) an unfolding step with respect to $main(M, J)$ in clause 1 of Example 19 is shown.

In the following we will show that the unfolding transformation preserves the operational semantics. To do this we first define the notion of operational equivalence.

Definition 77 ***Operational equivalence***
Let P be a CLP program with the constraint domain $\langle D, L \rangle$, $\langle \emptyset, C', C" \rangle$ and $\langle \emptyset, \overline{C}', \overline{C}" \rangle$ two final states of an SLD-tree for $P \cup \{p(\widetilde{X})\}$, where $p(\widetilde{X})$ is a goal with free variables $\widetilde{X}$. Let $\Im$ denote a D-interpretation.
Two states $\langle \emptyset, C', C'' \rangle$ and $\langle \emptyset, \overline{C}', \overline{C}'' \rangle$ are equivalent iff $\Im \models \exists_{-\widetilde{X}} C' \wedge C'' \Leftrightarrow \Im \models \exists_{-\widetilde{X}} \overline{C}' \wedge \overline{C}''$.

The following theorem taken from [32] describes the connection between the top-down operational and logical semantics of a CLP program, which is used to prove here that the unfolding transformation preserves the logical semantics.

Theorem 13 The connection between the top-down operational and logical semantics of a CLP program
Let $\Im$ be a D-interpretation and $\exists_{-\widetilde{X}}Q$ denote the existential closure of the formula Q except for the variables $\widetilde{X}$, which remain unquantified. Consider a P CLP program with the constraint domain $\langle L, D\rangle$. The success set $SS(P)$ collects the answer constraints to simple goals $p(\widetilde{X})$ with free variables $\widetilde{X}$:
$SS(P) = \{p(\widetilde{X}) \leftarrow c \mid \langle p(\widetilde{X}), \emptyset, \emptyset\rangle \rightarrow^* \langle \emptyset, C', C''\rangle, \Im \models c \longleftrightarrow \exists_{-\widetilde{X}} C' \wedge C''\}$.
Then $[SS(P)]_\Im = lm(P, \Im)$ where $lm(P, \Im)$ is the least D-model of P.

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

Theorem 14 The operational semantics preservation of 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 and c_j denotes the conjunction of the atomic constraints appearing in the body of R_j.

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 $SS(P)$ collects the answer constraints to simple goals $p(\widetilde{X})$ (see Theorem 13).
So an unfolding transformation preserves the operational semantics (we deal now only with success refutations).

Proof
Recall that this theorem says that applying an unfolding step on an SLD-tree does not affect the set of answer constraints for a goal. This is important for us since, together with Theorem 13 this means that an unfolding step does not change the least D-model of a CLP program (see Theorem 15).

$SS(P) = \{p(\widetilde{X}) \leftarrow c \mid \langle p(\widetilde{X}), \emptyset, \emptyset\rangle \rightarrow^* \langle \emptyset, C', C''\rangle, \Im \models c \longleftrightarrow \exists_{-\widetilde{X}} C' \wedge C''\}$.

Hence to prove that $SS(P) = SS(P')$ it is enough to show that for every branch of the SLD-tree $P \cup \{p(\widetilde{X})\}$ /i.e. for a given derivation $\langle p(\widetilde{X}), \emptyset, \emptyset\rangle \rightarrow^* \langle \emptyset, C', C''\rangle$ / there exists exactly one branch of the SLD-tree $P' \cup \{p(\widetilde{X})\}$ whose final state is equivalent to $\langle \emptyset, C', C''\rangle$ and $P' \cup \{p(\widetilde{X})\}$ has no additional branches.

To this end let us consider a branch of the SLD-tree $P \cup \{p(\widetilde{X})\}$ and examine the effect of an unfolding step for this branch.

Suppose that b_m in

$$R : h \leftarrow b_1, \cdots, \underbrace{b_m}, \cdots b_k, c$$

was "unified" with

$$R_j : \overbrace{h_j} \leftarrow b_{j_1}, \cdots, b_{j_{m_j}}, c_j \qquad (j = 1, \cdots, n).$$

Compare an $\rightarrow_r$ transition (1) (see Section 7.3) with an unfolding step (2) (see Definition 76):

(1) $\langle A \cup b_m, C, S\rangle \rightarrow_r \langle A \cup \{\underbrace{b_{j_1}, \cdots, b_{j_{m_j}}, c_j}_{body(R_j)}\}, C, S \cup \underbrace{(\overline{b_m} = \overline{h_j})}_{argument\ equations}\rangle$

(2) $R_{new_j} : h \leftarrow \overbrace{(\overline{b}_m = \overline{h}_j)}^{argument\ equations}, b_1, \cdots, b_{m-1}, \overbrace{b_{j_1}, \cdots, b_{j_{m_j}}, c_j}^{body(R_j)}, b_{m+1}, \cdots, b_k, c)$

One $\rightarrow_r$ transition can be viewed as an operation where, instead of b_m, the body predicates of the "unified" clause instance are inserted and the corresponding argument equations are added to the set of constraints.

Applying an unfolding step, the body atoms of the "unified" clause and the argument equations are added to the actual clause. We will show that an unfolding step simulates the $\rightarrow_r$ transition. Since the unfolding transformation involves all "unifiable" clauses, an unfolding step can be viewed as an operation which moves a subtree closer to the root (a node of an SLD-tree may have more then one children only if the following applied transition is an $\rightarrow_r$ transition).

More precisely: If there is a derivation of $P \cup \{G\}$ ($G = p(\widetilde{X})$) in which R (the clause has been unfolded upon) is not used as an input clause, this is then also a refutation in $P' \cup \{G\}$. But suppose R is used as an input clause in a refutation $\langle p(\widetilde{X}), \emptyset, \emptyset\rangle \rightarrow^* \langle \emptyset, C', C"\rangle$ of $P \cup \{G\}$.
We will prove that from such a refutation an $\langle p(\widetilde{X}), \emptyset, \emptyset\rangle \rightarrow^* \langle \emptyset, \overline{C}', \overline{C}"\rangle$ refutation of $P' \cup \{G\}$ can be constructed such that $\langle \emptyset, C', C"\rangle$ and $\langle \emptyset, \overline{C}', \overline{C}"\rangle$ are equivalent (see Definition 23).

Derivation R shows a part of the refutation when R is utilised as an input clause.
Derivation Res(R) shows when $Res(P, R, b_m)$ is used as an input clause instead of R (we suppose now that $Res(P, R, b_m)$ contains only one element).

The sets of the variables of the clauses ($vars(c)$ where c is a clause) are disjunct. Since the SLD resolution uses clauses instances, in our proof we deal with a substitution, denoted by σ, which creates an instance of R, R_j and R_{new_j}. There exists such a substitution because $vars(R) \cap vars(R_j) = \emptyset$ and $vars(R_{new_j}) = vars(R) \cup vars(R_j)$.

Derivation R:

1. $\cdots \langle b, GRem; C_1\rangle \vdash_r$

2. $\langle b_1\sigma, \cdots, b_{m-1}\sigma, b_m\sigma, b_{m+1}\sigma, \cdots, b_k\sigma, c\sigma, GRem; C_1 \wedge (\overline{b} = \overline{h\sigma})\rangle \vdash_{(r \cup c)^*}$

3. $\langle b_{m-1}\sigma, b_m\sigma, b_{m+1}\sigma, \cdots, b_k\sigma, c\sigma, GRem; C_2\rangle \vdash_{(r \cup c)^*}$

4. $\langle b_m\sigma, b_{m+1}\sigma, \cdots, b_k\sigma, c\sigma, GRem; C_3\rangle \vdash_r$

5. $\langle b_{j_1}\sigma, \cdots, b_{j_{m_j}}\sigma, c_j\sigma, b_{m+1}\sigma, \cdots, b_k\sigma, c\sigma, GRem; C_3 \wedge (\overline{b_m\sigma} = \overline{h_j\sigma})\rangle \vdash_r \cdots$

Derivation Res(R):

1. $\cdots \langle b, GRem; C_1\rangle \vdash_r$

2. $\langle b_1\sigma, \cdots, b_{m-1}\sigma, (\overline{b_m} = \overline{h_j})\sigma, b_{j_1}\sigma, \cdots, b_{j_{m_j}}\sigma, c_j\sigma, b_{m+1}\sigma, \cdots, b_k\sigma, c\sigma, GRem; C_1 \wedge (\overline{b} = \overline{h\sigma})\rangle \vdash_{(r \cup c)^*}$

3. $\langle b_{m-1}\sigma, (\overline{b_m} = \overline{h_j})\sigma, b_{j_1}\sigma, \cdots, b_{j_{m_j}}\sigma, c_j\sigma, b_{m+1}\sigma, \cdots, b_k\sigma, c\sigma, GRem; C_2\rangle \vdash_{(r \cup c)^*}$

4. $\langle (\overline{b_m} = \overline{h_j})\sigma, b_{j_1}\sigma, \cdots, b_{j_{m_j}}\sigma, c_j\sigma, b_{m+1}\sigma, \cdots, b_k\sigma, c\sigma, GRem; C_3\rangle \vdash_c$

5. $\langle b_{j_1}\sigma, \cdots, b_{j_{m_j}}\sigma, c_j\sigma, b_{m+1}\sigma, \cdots, b_k\sigma, c\sigma, GRem; C_3 \wedge ((\overline{b_m} = \overline{h_j})\sigma)\rangle \vdash_r \cdots$

For these derivations we employ the following notational conventions:

- We did not picture the i and s transitions, we only marked that an r or c transition was applied (since the content of the constraint store in the corresponding nodes is the same, the result of the i and s transitions are also the same).
- Every node has the following form: $\langle goals, C\rangle$, where $goals$ contains the actual list of goals and C is the constraint store.
- R is the clause and b_m is the literal in R to be unfolded upon.
- The first node is $\langle b, GRes, C_1\rangle$, where b is "unified" with the head of R in the next derivation step, $GRem$ is the remainder of the actual list of goals, and C_1 is the constraint store.
- We pictured only one refutation, so if b_m can be "unified" with more clauses then this map of nodes can be applied to every other branch of the SLD-tree which correspond to these clauses.

Node 1 in Derivation R and Derivation Res(R) is the first node. The first goal b is "unified" with $R\sigma$ / $R_{new_j}\sigma$.
Node 2 shows when the body predicates of $R\sigma$ (Derivation R) / $R_{new_j}\sigma$ (Derivation Res(R)) are added to the actual list of goals, and the corresponding node equations ($\overline{b} = \overline{h\sigma}$) are

added to the constraint store C_1.
Node 3 shows the state after the derivations of the subgoals $b_1\sigma, \cdots, b_{m-2}\sigma$. The content of the constraint store C_2 is the same in both derivations (SLD-trees) since the content of the constraint store in Node 2 was the same and the same subgoals were derived in both cases.
In Node 4 the set of constraints C_3 is also the same in both derivations since the subgoal $b_{m-1}\sigma$ was derived. But the next step is different: in Derivation R the next actual goal is $b_m\sigma$ while in Derivation Res(R) $(\overline{b_m} = \overline{h_j})\sigma$.
As can be seen in **Node 5**, after applying an r transition for $b_m\sigma$ in Derivation R the resulting node has the same label as when we apply a c transition for $(\overline{b_m} = \overline{h_j})\sigma$ in Derivation Res(R) because
$(\overline{b_m\sigma} = \overline{h_j\sigma})$ = $((\overline{b_m} = \overline{h_j})\sigma)$. Hence from Node 5 the derivation continues with the same list of goals and with the same content of the constraint store (see the comparison of the r transition (1) and unfolding (2)), so finally the result constraint set is the same, which of course means that the final state of the derivations is equivalent.

Theorem 15 The logical semantics preservation

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

Proof

Here we make use of the notes of Theorem 14.

Let $\Im$ be a D-interpretation.

From Theorem 13: $[SS(P)]_\Im = lm(P, \Im)$ and $[SS(P')]_\Im = lm(P', \Im)$.
From Theorem 14: $SS(P) = SS(P')$.
So,

$$lm(P, \Im) = [SS(P)]_\Im = [SS(P')]_\Im = lm(P', \Im).$$

From which: $lm(P, \Im) = lm(P', \Im)$.

9.3.3 Clause Removal

The aim of the learning process is to find a CLP program that does not cover any negative examples. During the learning process it is verified whether or not a clause covers any positive examples. If it covers no positive examples, it is then removed (otherwise it is unfolded). Removing a clause which covers only negative examples corresponds to pruning SLD-trees such that all refutations of negative examples and no refutations of positive examples are excluded.

9.3.4 The CLP_SPEC Algorithm

This section contains one of our key results, namely a specialization algorithm for constraint logic programs. In the CLP_SPEC method, similar to the general ILP approach, background

knowledge can be given that is not modified during the specialization. The algorithm specializes logic programs with respect to positive and negative examples by applying the transformation rule for unfolding together with clause removal. This is done in the following way. The algorithm consists of one main loop that continues until no negative examples are covered. When a clause is found that covers a negative example, and no positive examples, it is removed. When a clause is found that covers both a negative and a positive example, it is unfolded. The choice of which literal to unfold upon is made using the computation rule, which uses different strategies (i.e. it selects the leftmost, in a random way).

Input: An initial constraint logic program program P, background knowledge $B \subseteq P$ (a set of clauses that does not change during the learning process and which clauses does not take part in the refutations of negative examples), sets of ground atoms E^+, E^- (the positive and negative examples which are ground instances of a target predicate).
Output: Series of programs $P'^{(0)}, P'^{(1)}, \cdots, P'^{(n)} (P'^{(0)} = P)$, where
$P'^{(i+1)} = \widetilde{Unf}(P'^{(i)})$ $(0 \geq i \geq n)$, and $\widetilde{Unf}$ is the unfolding operator extended with clause removal.

THE CLP_SPEC ALGORITHM

1. if the program P does not terminate on all $e^+ \in E^+$
2. then stop "Initial program should cover all positive examples."
3. let i=0
4. while there is a clause R in P' that covers an atom in E^-
or (no more unfolding steps can be applied *****) **do**
5. begin
if R does not cover any atom in E^+
6. then
remove R from $P'^{(i)}$
7. else
begin
8. - unfold upon the literal b in R that is selected
by the computation rule
$P'^{(i+1)} := Unf(P'^{(i)}, R, b)$
9. - Let $D\_Res(P'^{(i)}, R, b) := Res(P'^{(i)}, R, b) \setminus \{$*those clauses that do not occur in refutations of positive examples* $\}$
10. - $P'^{(i+1)} := P'^{(i+1)} \setminus D\_Res(P'^{(i)}, R, b)$
/*to find the most specific theory*/
11. **end** /*else*/
12. **let** $i := i + 1$
13. **end** /*while*/

We note that this algorithm produces the most specific theory, removing as many clauses as possible (i.e. it removes all clauses that do not cover positive example - see program lines 9 and 10).

9.3.5 The Correctness of the CLP_SPEC algorithm

Theorem 16 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 of the termination of the algorithm is not (*) above. This also means that $P'^{(n)}$ is complete and consistent (i.e. it covers all positive examples and does not cover any negative examples).

Proof

According to Definition 75 we have to satisfy the following three conditions:

1. $\underline{M_{P'} \cap E^- = \emptyset}$

   We assume that the clauses of the background knowledge B does not take part in the refutations of negative examples.
   If the main loop in program line 4 terminates because there are no more program clauses which cover negative examples, then $P'^{(n)}$ does not cover any negative examples.
   If it terminates because no more unfolding step can be performed (*), and $P'^{(n)}$ still covers negative example(s), then our algorithm could not find a consistent hypothesis (the percentage of the covered negative examples is provided). In this case new and more precise constraints have to be introduced into the program. One aim of future work is to combine this algorithm with a constraint inferring method.

2. $\underline{E^+ \subseteq M_{P'}}$

   We prove this state for $M_{P'} = lm(P, \Im)$.

   From program line 1: $E^+ \subseteq M_{P'^{(0)}}$

   For every $i = 1, \cdots, n$

   - *The unfolding step in program line 8 does not change* $lm(P'^{(i)}, \Im)$ (see Theorem 15).

   - *The clause removal in program line 6 and 10 does not remove clauses that cover positive examples.*

3. $\underline{M_{P'} \subseteq M_P}$

   We prove this state for $M_{P'} = lm(P, \Im)$ in two steps:
   For every $i = 1, \cdots, n$

   - *The unfolding step in program line 8 does not change* $lm(P'^{(i)}, \Im)$ *(see Theorem 15).*

   - *The clause removal in program line 6 and 10 does not extend* $lm(P'^{(i)}, \Im)$.
     The clause removal cuts branches of the SLD-tree, so reduces or does not change the success set of the answer constraints $(SS(P'^{(i)}))$ as well as $lm(P'^{(i)}, \Im)$ (see Theorem 13).
     From which,
     $M_{P'^{(i+1)}} \subseteq M_{P'^{(i)}}$ for $i = 0, \cdots, n$, so $M_{P'} = M_{P'^{(n)}} \subseteq M_{P'^{(0)}} = M_P$.

The complexity of the algorithm depends on the number of iterations i (the number of unfoldings). During the running process the number of the clauses increases when unfolding is applied, so the number of iterations should be kept as low as possible. To do this, different computation rules can be employed (for more details see [7]).

Example 20 *We try out the CLP_SPEC algorithm on Example 19.*
We would like to obtain a CLP program which tells us whether a meal is a light fish-meal, i.e. a fish-meal whose sum of calorific values does not exceed 10. The program in Example 1 needs to be specialized since it does not merely cover fish-meals.

The set of positive examples:
$E^+ := \{fishlightmeal(A, sole), fishlightmeal(A, tuna)\}$

The set of negative examples:
$E^- := \{fishlightmeal(A, beef), fishlightmeal(A, pork)\}$

The goal for the SLD-tree which contains all the examples is: $fishlightmeal(A, M)$.
The unfolding steps and clause removals and the corresponding SLD-trees in each iteration are the following:

$\underline{P'^{(0)}}$ has the following form:

```
1.  fishlightmeal( A, M) : - {I + J  ≤  10}, appetiser(A, I),
main(M, J).
2.  appetiser(A, I) : - cheese(A, I), {I > 0}.
3.  appetiser(A, I) : - pasta(A, I), {I > 0}.
4.  main(M, J) : - fish(M, J), {J > 0}.
5.  main(M, J) : - meat(M, J), {J > 0}.
6.  fish(sole, 2).
7.  fish(tuna, 4).
```

```
8.  meat(beef, 5).
9.  meat(chicken, 4).
10.  pasta(general,1).
11.  cheese(camember,2).
```

Figure 18 shows the skeleton of the SLD-tree of $P'^{(0)}$ for the goal $fishlightmeal(A, M)$.

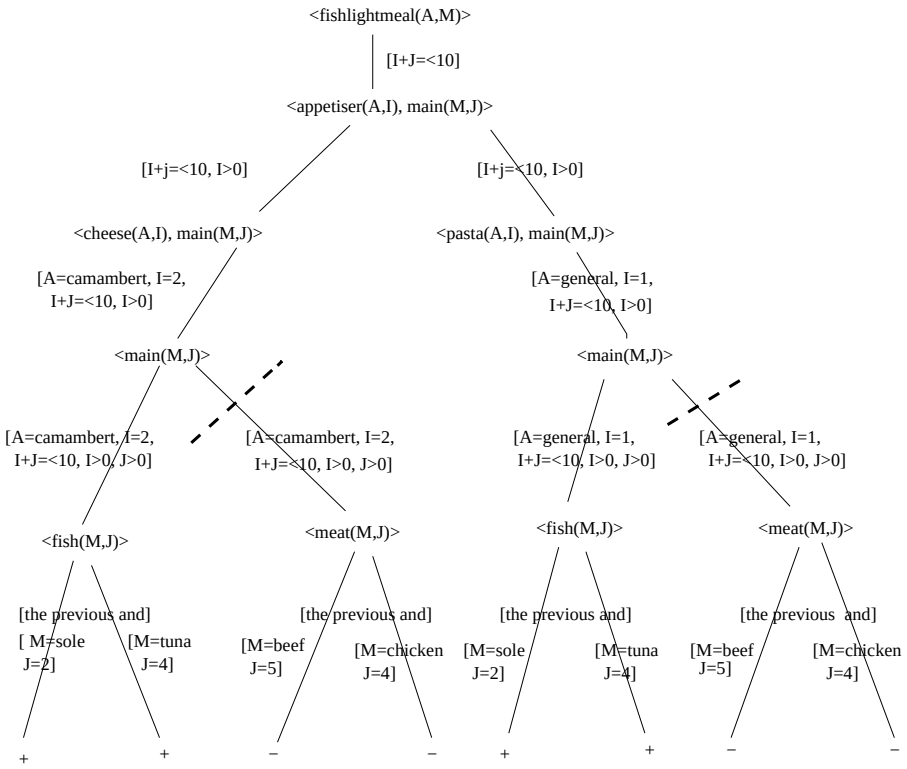

Figure 18: The skeleton of the SLD-tree of $P'^{(0)}$ for the goal $fishlightmeal(A, M)$

Choose the first clause and the $main(M, J)$ predicate for unfolding.

Instead of adding the argument equations we have made use of the same (corresponding) variable names.
The two new clauses are the following:

```
1.  fishlightmeal( A, M) : - {I + J  ≤  10}, appetiser(A, I),
main(M, J).

        4.  main(M, J) : - fish(M, J), {J > 0}.

        5.  main(M, J) : - meat(M, J), {J > 0}.
```

```
1.4 fishlightmeal(A,M) : - {I + J ≤ 10}, appetiser(A,I),
fish(M,J), {J > 0}.

1.5 fishlightmeal(A,M) : - {I + J ≤ 10}, appetiser(A,I),
meat(M,J), {J > 0}.
```

After the clause removal (that of clause 1) $P'^{(1)}$ has the following form:

```
1.4 fishlightmeal(A,M) : - {I + J ≤ 10}, appetiser(A,I),
fish(M,J), {J > 0}.
1.5 fishlightmeal(A,M) : - {I + J ≤ 10}, appetiser(A,I),
meat(M,J), {J > 0}.
2.  appetiser(A, I) : - cheese(A, I) , {I > 0}.
3.  appetiser(A, I) : - pasta(A, I), {I > 0}.
4.  main(M, J) : - fish(M, J), {J > 0}.
5.  main(M, J) : - meat(M, J), {J > 0}.
6.  fish(sole, 2).
7.  fish(tuna, 4).
8.  meat(beef, 5).
9.  meat(chicken, 4).
10.  pasta(general,1).
11.  cheese(camamber,2).
```

Figure 19 shows the skeleton of the SLD-tree of $P'^{(1)}$ for the goal $fishlightmeal(A, M)$.

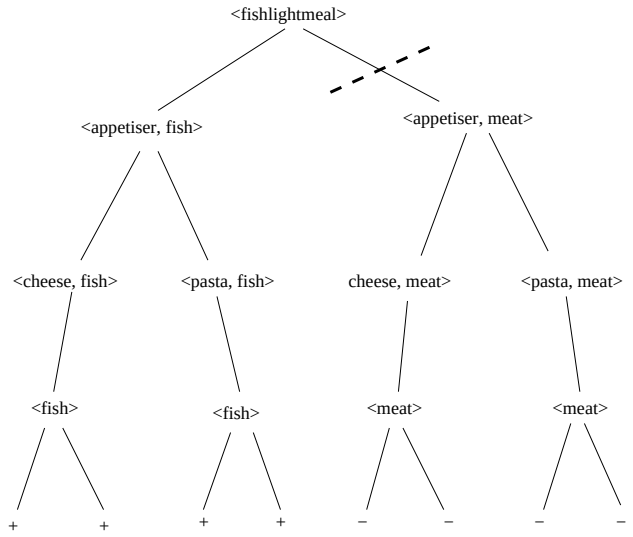

Figure 19: The skeleton of the SLD-tree of $P'^{(1)}$ for the goal $fishlightmeal(A, M)$

As we can see on Figure 19 the clauses 1.5, 4, 5, 8 and 9 do not take part in the refutation of positive examples, so they can be removed. The removing of clause 1.5 cuts the corresponding branch of the SLD-tree (which contains only negative examples).

Finally, $P'^{(1)}$ has the following form:

```
1.4 fishlightmeal( A,M) : - {I + J  ≤  10}, appetiser(A,I),
fish(M,J), {J > 0}.
2.  appetiser(A, I) : - cheese(A, I) , {I > 0}.
3.  appetiser(A, I) : - pasta(A, I), {I > 0}.
4.  fish(sole, 2).
5.  fish(tuna, 4).
6.  pasta(general,1).
7.  cheese(camamber,2).
```

One iteration was enough to obtain the specialized program that covers only positive examples.

9.4 Improving the CLP_SPEC Algorithm by Slicing

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 which is responsible for this incorrect covering. In our algorithm CLP_SPEC_SLICE a debugging system combined with a slicing technique is used to identify a buggy clause instance, then this clause is removed from the initial program.

9.4.1 Combining Algorithmic Debugging and Slicing for Constraint Logic Programs

The algorithmic program debugging method, introduced by Shapiro [69], 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 but it can be also extended to constraint logic programs in a fairly natural way.
A major drawback of this interactive debugging method is the large number of queries made to the user about the correctness of intermediate results of procedure calls.
A major improvement in the localization process is possible by combining the algorithmic debugging with slicing technique [74]. A proof tree is produced for a buggy program, then a proof tree dependence graph is constructed which is sliced and then those parts of the tree that have no influence on the visible symptom of a bug are removed. The algorithmic debugger traverses the sliced proof tree only, thus concentrating on the suspect part of the program. In this paper we refer to this method as the DEB_SLICE debugging method.

9.4.2 The CLP_SPEC_SLICE Algorithm

The main idea behind the CLP_SPEC_SLICE algorithm is that the identification of the next clause to be unfolded plays a crucial role in the effectinevess of the specialization process. It is assumed that when a negative example is covered by the current hypothesis (intermediate program) there is at least one clause which is responsible for this incorrect covering.
CLP_SPEC_SLICE uses the DEB_SLICE algorithm to identify a buggy clause of the program. The clause identified in this process will be unfolded in the next step of the specialization algorithm.

The CLP_SPEC_SLICE algorithm basically consists of the following three steps:

- Finding the clause the unfolding is applied upon (this step is done by the DEB_SLICE algorithm)
- Finding the literal within the clause that will be the basis of the unfolding: The same method is applied here as that used in SPECTRE [7].
- Performing the unfolding on the program.

The CLP_SPEC_SLICE algorithm is the following:

Input: An initial constraint logic program P, background knowledge $B \subseteq P$ (a set of clauses that remains unchanged during the learning process and does not take part in the refutation of negative examples), sets of ground atoms E^+, E^- (the positive and negative examples which are ground instances of a target predicate).

Output: A series of programs $P'^{(0)}, P'^{(1)}, \cdots, P'^{(n)} (P'^{(0)} = P)$, where $P'^{(i+1)} = \widetilde{Unf}(P'^{(i)})$ $(0 \geq i \geq n)$, and $\widetilde{Unf}$ is the unfolding operator extended with clause removal.

THE CLP_SPEC_SLICE ALGORITHM

1. if the program P does not terminate on all $e^+ \in E^+$
2. then stop "Initial program should cover all positive examples."
3. let i=0
4. while there is an $e^- \in E^-$ such that $P'^{(i)}$ does not fail on e^- **or**
(no more unfolding can be applied *****) **do**
begin
5. **find** a buggy clause $R \in P'^{(i)}$ using the DEB_SLICE debugger
(R is not in B)
if R does not cover any atom in E^+
6. **then**
remove R from $P'^{(i)}$

7. **else**
begin
8. - unfold upon the literal b in R that is selected
by the computation rule
$P'^{(i+1)} := Unf(P'^{(i)}, R, b)$
9. - Let $D\_Res(P'^{(i)}, R, b) := Res(P'^{(i)}, R, b) \setminus \{$*those clauses that do not occur in refutations of positive examples* $\}$
10. - $P'^{(i+1)} := P'^{(i+1)} \setminus D\_Res(P'^{(i)}, R, b)$
/*to find the most specific theory*/
11. **end** /*else*/
12. **let** $i := i + 1$
13. **end** /*while*/

We note that this algorithm also produces the most specific theory.

9.4.3 The Correctness of the CLP_SPEC_SLICE Algorithm

It can readily seen be that the difference between this and the CLP_SPEC algorithm is the choice of the clause whose literal is unfolded upon.

Theorem 17 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 (*) above.

Proof:

The correctness of the CLP_SPEC_SLICE algorithm depends on the correctness of the CLP_SPEC algorithm and the correctness of the DEB_SLICE algorithm. The CLP_SPEC algorithm is correct with respect to these conditions (see Theorem 16). There are special cases however when the buggy clause cannot be identified by the DEB_SLICE method. There is a solution to this problem [**?**], but the complexity of this method is too large compared to the complexity of the CLP_SPEC algorithm, so we prefer to apply the CLP_SPEC algorithm in this case.

9.5 Prototype Implementation

Both algorithms (CLP_SPEC and CLP_SPEC_SLICE) were implemented in SICStus Prolog. For slicing we used a previously developed tool [74].
The algorithms were tested on simple examples such as N-Quens, rectangle [2](to recognize a horizontally lying rectangle), horse-jumping, and so on. During the testing we employed the Prolog computation rule (i.e. we chose the leftmost literal).

From the test results it may be concluded that the number of clauses learned by CLP_SPEC_SLICE is less than that learned by the CLP_SPEC algorithm. This means that CLP_SPEC_SLICE can learn more compact theories than CLP_SPEC. However, during the running of the CLP_SPEC_SLICE algorithm an oracle has to answer membership questions to identify a buggy clause instance. Generally, about the 40 % of these user queries could be reduced by applying slicing (in a few cases 100 %). Sometimes it was difficult to answer the user queries, since they were about the correctness of numerical functions (data). A list of the slice points, which may be identified with the help of a graphical user interface, could be given in a list inserted in the goal in the following way:

$fishlightmeal(A, beef), [2].$

In this negative example the second argument is incorrect, so the proof tree is created for the goal $fishlightmeal(A, beef)$, the slice is created with respect to the second argument ($beef$), and the algorithmic debugger asks only about the correctness of those predicates that are included in this slice of the proof tree. If the list is empty then the proof tree is walked by the original algorithmic debugging method.

The following small example shows how the CLP_SPEC_SLICE algorithm learned horse-jumping from an initial theory. As the example is very small, no slice was created and the algorithmic debugger asked only one question at each iteration step.

The initial program describing horse-jumping (which needs to be specialized) was the following:

```
1.  horse(A,B,C,D):-Horiz=abs(A-C),Vert=abs(B-D),horse_step(Horiz,Vert).
2.  horse_step(Horiz,Vert) :- num(Horiz), num(Vert).

background (num(X) :- X=0.0).  background (num(X) :- X=1.0).
background (num(X) :- X=2.0).  background (num(X) :- X=3.0).
background (num(X) :- X=4.0).  background (num(X) :- X=5.0).
background (num(X) :- X=6.0).  background (num(X) :- X=7.0).
background (num(X) :- X=8.0).  background (num(X) :- X=9.0).
```

The set of positive exampes:

```
positive horse(1.0,2.0,3.0,3.0).  positive horse(3.0,6.0,4.0,4.0).
positive horse(4.0,2.0,3.0,4.0).  positive horse(5.0,2.0,3.0,3.0).
positive horse(5.0,6.0,4.0,4.0).  positive horse(4.0,6.0,3.0,4.0).
```

The set of negative examples:

```
negative horse(3.0,2.0,7.0,6.0).  negative horse(2.0,3.0,4.0,8.0).
negative horse(3.0,5.0,7.0,6.0).  negative horse(2.0,3.0,4.0,5.0).
negative horse(3.0,2.0,7.0,6.0).  negative horse(2.0,3.0,4.0,6.0).
negative horse(2.0,3.0,3.0,6.0).
```

The running of the CLP_SPEC_SLICE algorithm:

```
 -? start.

Welcome to CLP_SPEC learning system.

Please enter the filename to be processed:  horse.

The background knowledge is:

3:  num(A):-A=0.0 4:  num(A):-A=1.0 5:  num(A):-A=2.0
6:  num(A):-A=3.0 7:  num(A):-A=4.0 8:  num(A):-A=5.0
9:  num(A):-A=6.0 10:  num(A):-A=7.0 11:  num(A):-A=8.0
12:  num(A):-A=9.0

The theory that needs to be specialized is:

1:  horse(A,B,C,D):-E=abs(A-C),F=abs(B-D),horse_step(E,F)

2:  horse_step(A,B):-num(A),num(B)

The positive examples are:

1013:  horse(1.0,2.0,3.0,3.0) 1014:  horse(3.0,6.0,4.0,4.0)
1015:  horse(4.0,2.0,3.0,4.0) 1016:  horse(5.0,2.0,3.0,3.0)
1017:  horse(5.0,6.0,4.0,4.0) 1018:  horse(4.0,6.0,3.0,4.0)

The negative examples are:

1019:  horse(3.0,2.0,7.0,6.0) 1020:  horse(2.0,3.0,4.0,8.0)
1021:  horse(3.0,5.0,7.0,6.0) 1022:  horse(2.0,3.0,4.0,5.0)
1023:  horse(3.0,2.0,7.0,6.0) 1024:  horse(2.0,3.0,4.0,6.0)
1025:  horse(2.0,3.0,3.0,6.0)

Checking input examples:

The sets of positive and negative examples are distinct.

Checking positive examples:

All positive examples are covered.

Checking negative examples:

1019:  horse(3.0,2.0,7.0,6.0) covered.  1020:
horse(2.0,3.0,4.0,8.0) covered.
1021:  horse(3.0,5.0,7.0,6.0) covered.  1022:
horse(2.0,3.0,4.0,5.0) covered.
```

```
1023:  horse(3.0,2.0,7.0,6.0) covered.  1024:
horse(2.0,3.0,4.0,6.0) covered.
1025:  horse(2.0,3.0,3.0,6.0) covered.

 The fact horse(3.0,2.0,7.0,6.0) is covered by the theory.

Starting the false proc.  algorithm to determine
the basis of the unfolding.

Is it ok [horse_step(4.0,4.0)] (y/n) n

Unfolding at the clause instance:

 2:  horse_step(4.0,4.0):-num(4.0),num(4.0)

- trying resolvent(s):  [2-1]
- trying resolvent(s):  [2-2]

 The result of the unfolding is:

1:  horse(A,B,C,D):-E=abs(A-C),F=abs(B-D),horse_step(E,F).
2:  horse_step(A,B):-num(A),B=1.0.
3:  horse_step(A,B):-num(A),B=2.0.

Checking positive examples:

All positive examples are covered

Checking negative examples:

1021:  horse(3.0,5.0,7.0,6.0) covered.  1022:
horse(2.0,3.0,4.0,5.0) covered.

The above theory:

covers 6 positive samples from 6 (100.00 percent)
fails on 5 negative samples from 7 (71.43 percent.)

The fact horse(3.0,5.0,7.0,6.0) is covered by the theory.
Starting the false proc.  algorithm to determine
the basis of the unfolding.

Is it ok [horse_step(4.0,1.0)] (y/n) n

Unfolding at the clause instance:

2:  horse_step(4.0,1.0):-num(4.0),1.0=1.0
```

```
- trying resolvent(s):  [2-1]

The result of the unfolding is:

1:  horse(A,B,C,D):-E=abs(A-C),F=abs(B-D),horse_step(E,F)
2:  horse_step(A,B):-A=2.0,B=1.0
3:  horse_step(A,B):-num(A),B=2.0

Checking positive examples:

All positive examples are covered

Checking negative examples:

1022:  horse(2.0,3.0,4.0,5.0) covered.

The above theory:

covers 6 positive samples from 6 (100.00 percent) and
fails on 6 negative samples from 7 (85.71 percent).

The fact horse(2.0,3.0,4.0,5.0) is covered by the theory.

Starting the false proc.  algorithm to determine
the basis of the unfolding.

Is it ok [horse_step(2.0,2.0)] (y/n) n

Unfolding at the clause instance:

3:  horse_step(2.0,2.0):-num(2.0),2.0=2.0

- trying resolvent(s):  [3-1]

The result of the unfolding is:

1:  horse(A,B,C,D):-E=abs(A-C),F=abs(B-D),horse_step(E,F)
2:  horse_step(A,B):-A=2.0,B=1.0
3:  horse_step(A,B):-A=1.0,B=2.0

Checking positive examples:

All positive examples are covered

Checking negative examples:

The above theory:
```

```
covers 6 positive samples from 6 (100.00 percent) and
fails on 7 negative samples from 7 (100.00 percent).
```

——————— The final result theory is: ———————

1: horse(A,B,C,D):-E=abs(A-C),F=abs(B-D),horse_step(E,F)
2: horse_step(A,B):-A=2.0,B=1.0
3: horse_step(A,B):-A=1.0,B=2.0

```
Checking positive examples:

All positive examples are covered.

Checking negative examples:

No negative example is covered

The above theory:

covers 6 positive samples from 6 (100.00 percent) and
fails on 7 negative samples from 7 (100.00 percent).

yes - ?
```

As we can see the algorithm has learnt the correct theory for horse-jumping.

9.6 Related Work

Different learning methods have been introduced for learning constraint logic programs in [35, 67, 47, 58, 48] but to our knowledge only one of them has so far been implemented [67] for a special domain.

Kawamura and Furukawa [35] adopted the dominant paradigm in ILP, namely the paradigm of inverse resolution for generalizing constraints.

Sebag et al. [67] propose a framework for learning clauses which can discriminate between positive and negative examples expressed as constrained clauses. They conjecture that only a subset of the entire set of discriminating clauses need to be determined fully in order to explicitly represent the learned concept. The remaining clauses of the concept can be derived from these basic set of clauses.

Martin and Vrain's [47] idea is that instead of interpreting function symbols in constraints symbolically, if we interpret them by more semantic means, there is scope for development of better algorithms for generalizing and inducing constraint logic programs.

Page and Frisch [58] extend the concepts involved in the generalization of atoms, to more general forms of atoms, especially atoms with constraints associated with them. The constraint is a logical expression built of specialized predicates called constraint predicates and represents a form of

restriction on the atom to which the constraint is involved.

Mizoguchi and Ohwada [48], extend ideas from ILP based on Plotkin's framework [62] of Relative Least General Generalization(RLGG) to induce constraint logic programs.

We have adopted another existing ILP technique for ICLP, namely the specialization method SPECTRE [7].
Our algorithm (CLP_SPEC) is unable to create new (more precise) constraints during the learning process. It applies unfolding on the defined predicates.
One of the aims of future work will be to combine the CLP_SPEC method with another algorithm that is able to infer new constraints as well.

A major area of research motivated by all the ICLP systems involves the question of developing notions of bias restrictions. A reduced size of search space can help to solve the time and complexity problems. In our work we produced a modification of the specialization method which combines the CLP_SPEC algorithm with algorithmic debugging and slicing in order to reduce the bias and to learn more compact theories.

9.7 Summary

The idea of *Logic Programming (LP)* [50] is to use a computer for drawing conclusions from declarative descriptions. Such descriptions -called Logic Programs - consist of finite sets of logic formulas. In order to be able to apply a relatively simple and powerful inference rule (called the SLD-resolution principle), restrictions are introduced on the language of formulas.

Constraint programming [46] is an emergent software technology for declarative description and effective solving of large problems. Not only does it have a strong theoretical foundation but it is attracting widespread commercial interest as well, in particular, in areas of modelling heterogenous optimisation and satisfaction problems.
The cornerstone of *Constraint Logic Programming (CLP)* [46, 32] is the notion of constraint. Constraint Logic programming is a fusion of two declarative paradigms: Constraint Solving and Logic Programming. The framework extends classical logic programming by removing the restriction on programming within the Herbrand universe alone; unification is replaced by the more general notion of constraint satisfaction.

Inductive Constraint Logic Programming (ICLP) refers to a class of machine learning algorithms where the agent learns a theory (a CLP program) from examples and background knowledge.

We presented a learning method for constraint logic programs (ICLP) by combining unfolding and clause removal, formulated the semantics of this new method (CLP_SPEC), proved that the defined operator preserves the operational and logical semantics (for LP a similar theorem was provided in [68]), and improved the unfolding technique by applying slicing (CLP_SPEC_SLICE). A prototype learning tool was then implemented based on these methods.
The specialization algorithm is an extension of the original algorithm SPECTRE [7], which is a learning method for logic programs (SPECTRE 3.0 [6] has some new features for handling numerical values). Logic programming (LP) can be viewed as a special case of constraint logic programming. This is a substantial extension, so CLP not only has distinct syntactics from LP but also distinct semantics.

We formulated the necessary definitions and theorems needed to formalize the CLP_SPEC algorithm and to analyze it. We provided the exact definitions of the specialization problem, unfolding transformation and operational equivalence. We proved that the application of the unfolding operator does not affect the set of the answer constraints of a CLP program for a given goal and preserves the operational semantics as well as the logical semantics. The correctness of the CLP_SPEC algorithm was proved on the basis of these theorems.

In our work we furnished a modification of the specialization method which combines the CLP_SPEC algorithm with algorithmic debugging and slicing to reduce the bias.
IMPUT [2] is a specialization method for LP that also improves the original SPECTRE [7] algorithm using a debugging algorithm to identify a buggy clause instance, but it has the big disadvantage that an oracle has to answer membership questions to identify a buggy clause instance.
In our work the combination of algorithmic debugging with slicing technique was extended to CLP in order to reduce the number of user queries.

Both algorithms were implemented and tested on different examples and, as the results of the implementation show, compact theories can be learned without much difficulty.

Part IV
LEARNING SEMANTIC FUNCTIONS OF ATTRIBUTE GRAMMARS

The results of this part of the thesis are covered in [77].

10 Learning Semantic Functions of Attribute Grammars

Attribute Grammars (AGs)[91, 38, 1, 55] are a generalization of the concept of Context Free Grammars (CFGs). The formalism of AGs has been widely used for the specification and implementation of programming languages. Actually, there is an intimate relationship between AGs and Logic Programming. This part of the dissertation presents a parallel method for learning semantic functions of Attribute Grammars (AGs) based on AGLEARN [22] and PAGE system [88]. The method is more efficient in both execution time and interactions needed than the sequential one. The method presented is adequate for S-attributed grammars and for L-attributed grammars as well.

10.1 Introduction

In the framework of compilation-oriented language implementation, attribute grammars [36] are the most widely applied semantic formalism. The notion of an attribute grammar is an extension of the notion of a context free grammar. The idea is to decorate parse trees of a context free grammar by additional labels which provide a "semantics" for the grammar. Every node of a parse tree labeled by a nonterminal is also decorated by a tuple of semantic values called attribute values. The number of attribute values is fixed for any nonterminal symbol of the grammar. Their names are called attributes of nonterminal. Since the definition of an attribute grammar usually requires much work it would be useful to have a tool for inferring semantic rules in attribute grammars based on examples.
In the case of inductive learning from examples, the learner is given some examples from which general rules or a theory underlying the examples can be derived. An inductive concept learner is given by a set of training examples, some background knowledge, a hypothesis description language, and an oracle willing to answer questions (in the case of interactive learner). The aim is to find a hypothesis such that the hypothesis is complete and consistent with respect to the examples and background knowledge.
To express efficiently the examples, the background knowledge, and the Hypothesis to be induced we need to use a language L with sufficient expressive power. Many systems were developed for learning logic programs, using first order predicate logic language tools (Inductive logic Programming (ILP)). Attribute Grammars merge the declarative power of predicate logic with the flexibility of a predefined interpretation of its terms. Complex objects and relations can be described in the framework of AGs. Introducing an AG-based description language L in ILP implies the definition of an Attribute Grammar learner.
In the following sections we will show how this integration is carried out using the AGLEARN [22] methodology and we will give the description of an innovative technique for a parallel implementation.
AGLEARN is a method for learning semantic functions of attribute grammars, which infers semantic rules of attribute grammars from examples. This is an interactive system, so during the execution the oracle has to answer a lot of questions, which requires a lot of work and much time. The parallel implementation of AGLEARN makes it possible to decrease the number of *oracles* so it is more efficient in both execution time and interactions needed (We rercall that the execution time depends on the number of the user queries). Our method uses the PAGE system, which is a general purpose parallel parser, augmented with a powerful semantic evaluator. Notice that this parallel method can be implemented using any general purpose parallel parser that has the appropriate facilities for storing and handling the necessary information.
The following sections are organized as follows. Section 10.2 gives an introduction to the concepts of Attribute Grammars. Section 10.3 provides a brief introduction to the AGLEARN method, and gives a typical example. Then Section 10.4 presents the computational model of PAGE in order to

represent the necessary facilities for the parallelization. In Section 10.5 the PAGELEARN method is presented. We give a detailed description of the parallel method for S-attributed grammars explaining how can we handle the circuity problem, the we furnish an illustrative example. Finally we show how the parallel method works for L-attributed grammars.

10.2 Preliminaries

10.2.1 Attribute Grammars

Attribute Grammars have been proposed by Knuth [36, 37] as an extension of context-free grammars. The original motivation was to facilitate compiler specification and development procedure.
While compilers was the initial area of research for AGs, they can also be used in a very wide research spectrum, where relations and dependences among structured and interpreted data is very valuable. Areas like software engineering [59, 43], visual programming [83], logic programming [14], distributed programming [92, 64], functional logic programming [45] and signal processing are some notable examples.

Definition 78 Context-free grammar
A context-free grammar G *is a quadruple such that* $G =< N, T, P, D >$*, where* N *is a finite set of* nonterminal symbols, T *is a set of* terminal symbols, P *is a finite set of* productions, *and* $D \in N$ *is the* start *symbol of* G.

An element in $V = N \cup T$ is called *grammar symbol*. The production in P are pairs of the form $X \to \alpha$, where $X \in N$ and $\alpha \in V^*$, i.e. the left hand side symbol (LHSS) X is a nonterminal, and the right hand side symbol (RHSS) α is a string of grammar symbols. An empty RHSS (empty string) will be denoted by ε.

Definition 79 Attribute Grammar
An Attribute Grammar *consists of three elements, a* context-free grammar G*, a finite set of* attributes A *and a finite set of* semantic rules R*. Thus* $AG =< G, A, R >$.

A finite set of attributes $A(X)$ is associated with each symbol $X \in V$. The set $A(X)$ is partitioned into two disjoint subsets, the *inherited attributes* $I(X)$ and the *synthesized attributes* $S(X)$. Thus $A = \cup A(X)$.
The production $p \in P$, $p : X_0 \to X_1 \cdots X_n$ (n $\geq$ 1) has an *attribute occurrence* $X_i.a$, if $a \in A(X_i)$, $0 \leq i \leq n$. A finite set of *semantic rules* R_p is associated with the production p, with exactly one rule for each synthesized attribute occurrence $X_0.a$ and exactly one rule for each inherited attribute occurrence $X_i.a$, $1 \leq i \leq n$.
Thus R_p is a collection of semantic rules of the form $X_i.a = f(y_1, \cdots, y_k)$, $k \geq 1$, where

1. either $i = 0$ and $a \in S(X_i)$, or $1 \leq i \leq n$ and $a \in I(X_i)$
2. each $y_j, 1 \leq j \leq k$, is an attribute occurrence in p and
3. f is a function called *semantic function*, that maps the values of $y_1, \ldots, y_k$ to the value of $X_i.a$. In a semantic rule $X_i.a = f(y_1, \ldots, y_k)$, the occurrence $X_i.a$ *depends* on each occurrence $y_j, 1 \leq j \leq k$.

Thus $R = \cup R_p$. By definition, synthesized attributes are *output* to the LHSS of the productions while inherited attributes are *output* to the RHSS. In other words synthesized attributes move the data flow *upwards* and inherited attributes move the data flow *downwards* in the derivation tree during the attribute evaluation procedure.
Remark: Note that each semantic rule of an attribute grammar can be viewed as a definition of the relation between local attribute values of the neighboring nodes of the parse tree. This relation is defined for a production rule and should hold for every occurrence of this production rule in any parse tree. In the original definition of AG (Definition 79) the definition of the relation takes the form of an equation. It is possible to generalize the concept of semantic rules and allow them to be arbitrary formulae (not necessarily equalities) over a language Ł.

Definition 80 Conditional Attribute Grammar
A Conditional Attribute Grammar *(CAG) is an attribute grammar having the concept of the semantic rules extended. Thus a CAG is a 5-tuple* $< G, S, A, \Im, I >$ *where:*

- G *is the underlying context free grammar*
- S *is a set of sorts (i.e. of domains where the attributes have values)*
- A *is a finite set of attributes. Each attribute* a *has a sort* $s(a)$ *in* S.
- $\Im$ *is a map function of a logic formula* $\Im_p$ *written in terms of an* S*-sorted logic language Ł for each production rule* $p \in P$. *The variables of a formula* $\Im_p$ *include all* output *attribute occurrences of* $A(p) = \cup_{X \in P} A(X)$. *The allowed form of the function* $\Im_p$ *is either a function* f *or a relation* c.
  - *in the case of a function,* f *has the form*
    $f : X_{p,k}.a = f(X_{p,k_1}.a_1, \ldots, X_{p,k_m}.a_m)$
    where $f : I(X_{p,k_1}.a_1) \times \ldots \times I(X_{p,k_m}.a_m) \rightarrow I(X_{p,k}.a)$.
  - *in case of relation,* c *has the form*
    $c : c(X_{p,k_1}.a_1, \ldots, X_{p,k_m}.a_m)$
    where $c : I(X_{p,k_1}.a_1) \times \ldots \times I(X_{p,k_m}.a_m) \rightarrow \{true, false\}$.
- I *is an* interpretation *of Ł in some* S*-sorted algebraic structure* $\mathcal{A}$.

Semantic rules induce dependences between attributes. These dependences can be represented by a *dependency graph*, from which partial ordering relations are implied. From these partial orderings the *evaluation order* of the attribute occurrences can be determined. A *decorated tree* is a derivation tree in which all the attribute occurrences have been evaluated according to their associated semantic rules. The dependency graph characterizes all restrictions on the control of computations. The actual sequence of attribute evaluation must preserve this ordering which is called *attribute evaluation strategy*. Attribute grammars can be classified according to the attribute evaluation strategy used .

Definition 81 S-attributed Grammars
A special class, introduced by Knuth [36], is the $S - attributed$ *grammars where only synthesized attributes are allowed.*

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

Definition 82 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 just on the attributes in the set* $\bigcup_{k \in \{1,\cdots,j-1\}} Inh(X_{p,k}) \cup Syn(X_{p,0})$ *for* $j = 1, \ldots, n_p$.

Example 21 *We now present a typical example for the type checking of arithmetic expressions [91]. Consider the following attribute grammar:*

- **Nonterminals:** $N = \{Expression, Term, Factor, AddOp, MulOp\}$
- **Terminals:** $T = \{Real, Integer, +, -, \times, /, (,)\}$
- **Start Symbol:** $D = Expression$
- **Sorts:**
  $S = \{S_{mode}, S_{operator}\}$ *where*
  - $S_{mode} = \{int, real\}$
  - $S_{operator} = \{add, sub, mul, div\}$

  $\Im = \{add, sub, mul, div, int, real, id, f_1, f_2\}$ *where*
  - add, sub, mul, div, int *and* $real$ *are constants*
  - $id : S_{mode} \rightarrow S_{mode}$ *denotes the identity function*
  - $f_1 : S_{mode} \times S_{mode} \rightarrow S_{mode}$

    $f_1(op_1, op_2) :=$ **if** $op_1 = real$ **or** $op_2 = real$ **then** *real*
    else *int*
  - $f_2 : S_{operator} \times S_{mode} \times S_{mode} \rightarrow S_{mode}$

    $f_2(op_1, op_2, op_3) :=$ **if** $op_1 = mul$ **and** $op_2 = int$ **and** $op_3 = int$
    then *int*
    else *real*
- **Attributes:** $A = Syn = \{mode, operator\}$ *so that*
  $Syn(Expression) = Syn(Term) = Syn(Factor) = mode$
  $Syn(AddOp) = Syn(MulOp) = operator$
- **Production and Semantic Rules:**

  1. $Expression_0 \rightarrow Expression_1\ AddOp\ Term$
     $R(1) : Expression_0.mode := f_1(Expression_1.mode, Term.mode)$
  2. $Expression \rightarrow Term$
     $R(2) : Expression.mode := Term.mode$
  3. $Term_0 \rightarrow Term_1\ MulOp\ Factor$
     $R(3) : Term_0.mode := f_2(MulOp.operator, Term_1.mode, Factor.mode)$
  4. $Term \rightarrow Factor$
     $R(4) : Term.mode := Factor.mode$

5. $Factor \rightarrow Real$
   $R(5) : Factor.mode := real$
6. $Factor \rightarrow Integer$
   $R(6) : Factor.mode := int$
7. $Factor \rightarrow (Expression)$
   $R(7) : Factor.mode := Expression.mode$
8. $AddOp \rightarrow +$
   $R(8) : AddOp.operator := add$
9. $AddOp \rightarrow -$
   $R(9) : AddOp.operator := sub$
10. $MulOp \rightarrow *$
    $R(10) : MulOp.operator := mul$
11. $MulOp \rightarrow /$
    $R(11) : MulOp.operator := div$

10.3 The AGLEARN method

AGLEARN [22] is a 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 together allow one to restrict the space of relations and give a smaller representation of data. The basic idea of this method is that the learning problem of semantic functions is transformed to a propositional form and the hypothesis induced by a propositional learner is transformed back into semantic functions. This approach is motivated by the fact that there is a close relationship between attribute grammars and logic programs [14].
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. An example contains a string which can be derived from the target nonterminal. We suppose 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, the background semantic functions and the examples can be used.
The input :

- The $AG = (G, S, A, \Im, I)$ in which :
  - The unambiguous context-free grammar is $G = (N, T, P, D)$
  - The set of productions P is partitioned into two disjoint sets P_B (The background rules) and P_T (the target rules). The set of semantic functions R is fully defined for the rules belonging to P_B and there are no semantic functions in R associated with the rules belonging to P_T.
  - A partially defined set of semantic conditions C. C is fully defined for P_B, and there is no condition for any rule in P_T.
  - It is supposed that the sorts of attributes(S)and the set of initial semantic functions $\Im_i$ are defined in advance.
- For each $p \in P_T$ and for each synthesized attribute occurrence $X_{p,0}.a$ there is a set of examples $E_p(a)$ (which is partitioned into two disjoint sets) the set of positive examples $E_p^+(a)$ and

negative examples $E_p^-(a)$. Let us suppose that p has the form of $p : X_{p,0} \rightarrow X_{p,1}, \ldots, X_{p,n_p}$. An example $e \in E_p(a)$ is given in the form: $e = (w, (a_1, v_1), ..., (a_m, v_m), (a_{m+1}, v_{m+1}))$, where $w \in T^*$, $X_{p,0} \Rightarrow^* w$, $\{a_1, .., a_m\} = Inh(X_{p,0})$, and $a_{m+1} = a$. A pair $(a_j, v_j)(1 \leq j \leq m+1)$ in the example e denotes the attribute $X_{p,0}.a_j$ that has the value v_j in the evaluated attributed tree built for the word w.

The aim is to find a set of functions $\Im_t$ and for each production $p \in P_T$ a definition of the sets $R(p)$ and $C(p)$ such that the semantic functions and conditions in $R(p)$ and $C(p)$ are defined by the elements of $\Im_i \cup \Im_t$. The resultant attribute grammar must be complete and consistent with the examples.

10.3.1 Learning semantic functions of S-attributed grammar

The method is based on the idea that the semantic functions of the background rules can introduce new columns in the table corresponding to the transformed learning problem.
Consider the production $p : X_{p,0} \rightarrow X_{p,1}, \ldots, X_{p,n_p} \in P_T$, and suppose that the symbols in this rule can possess only synthesized attributes. We would like to learn the semantic function associated with $X_{p,0}.a$. For $X_{p,0}.a$, a $T(a)$ table must be constructed. Each row of this table corresponds to an example from $E_p(a)$. The table has the set of columns

$$\{ \textit{class, word, target} \} \cup U \cup \Im_{UR} \cup \Im_{UCR} \cup \Im_{UF} \cup \Im_{UCF}$$

where, for a given example $e = (w, (a, v))$ these columns are defined as follows:

1. class(e)= + if $e \in E_P^+(a)$
   - otherwise (i.e. $e \in E_P^-(a)$)
2. *word(e)* = *w*
3. *target(e)* is the value of a in e, i.e. $target(e) = v$
4. The key part is the computation of the column U (see Table 3) which contains columns corresponding to the attribute instances $X_{p,j}.b \in Syn(X_{p,j}), j = 1, \ldots, n_p$. The value of this column is computed using the semantic functions in P_B. For the example $e = (w, (a, v))$ this computation is performed as follows:
   - An attributed tree is built on the input string w
   - If the subtree derived from symbol $X_{p,j}$ contains only nodes corresponding to rule instances belonging to P_B, then the attributes of this subtree can be evaluated.
   - If the subtree derived from $X_{p,j}$ contains a node corresponding to a rule instance belonging to P_T then the values of the attributes in $Syn(X_{p,j})$ are asked from the user for the given derivation.
5. Let $X_{p,k_1}.a_1, \cdots, X_{p,k_l}.a_l$ $(0 < k_1, \cdots, k_l \leq n_p)$ be applied attribute occurrences and let $f : \tau_1 \times \cdots \times \tau_i \rightarrow \{true, false\}$ be a Boolean function ($f \in \Im_i$) such that $\tau_i = I(X_{p,k_i}.a_i)$ $(1 \leq i \leq l)$. Then there is a column for the relation $f(X_{p,k_1}.a_1, \cdots, X_{p,k_l}.a_l)$ in $\Im_{UR}$.
6. Let $X_{p,k}.b$ be an applied attribute occurrence $(0 < k \leq n_p)$ and let $\{c_1, \cdots, c_m\}$ be a set of constant values occurring in the column $X_{p,k}.b$ of U. Then for each c_i there is a corresponding column for the relation $X_{p,k}.b = c_i$ in $\Im_{UCR}$ $(1 \leq i \leq m)$.

class	word	target	U			F_{UCR}						F_{UCF}		F_{UF}	
		$T_0.m$	U_1	U_2	U_3	R_1	R_2	R_3	R_4	R_5	R_6	F_1	F_2	C_1	C_2
+	$3*2.5$	real	int*	mul	real	T	F	T	F	F	T	F	**T**	F	T
+	$5*3$	int	int*	mul	int	**T**	F	**T**	F	**T**	F	**T**	F	T	T
+	$1.5*4$	real	real*	mul	int	F	T	T	F	T	F	F	**T**	T	F
+	$2.5/3$	real	real*	div	int	F	T	F	T	T	F	F	**T**	T	F
+	$2/3$	real	int*	div	int	T	F	F	T	T	F	F	**T**	F	F
+	$6/3.4$	real	int*	div	real	T	F	F	T	F	T	F	**T**	F	T
-	$2*3.2$	int	int*	mul	real	T	F	T	F	F	T	T	**F**	T	F
-	$4.3/2$	int	real*	div	int	F	T	F	T	T	F	T	**F**	F	T
-	$8/3$	int	int*	div	int	T	F	F	T	T	F	T	**F**	T	T

Table 3: The generated propositional table

7. Let $X_{p,k_1}.a_1, \cdots, X_{p,k_l}.a_l$ $(0 < k_1, \cdots, k_l \leq n_p)$ be applied attribute occurrences and let $f : \tau_1 \times \cdots \times \tau_l \rightarrow \tau_0$ be a function ($f \in \Im_i$) such that $\tau_i = I(X_{p,k_i}.a_i)$ $(1 \leq i \leq l)$ and $\tau_0 = I(a)$. Then there is a column for the relation $a = f(X_{p,k_1}.a_1, \cdots, X_{p,k_l}.a_l)$ in $\Im_{UF}$.

8. Let $\{c_1, \cdots, c_m\}$ be a set of constant values occurring in U or in the column of $target(e)$. Then for each c_i there is a corresponding column $a = c_i$ in $\Im_{UCF}$ if and only if $I(a) = I(c_i)$ $(1 \leq i \leq m)$.

The if-rules are constructed from the $T(a)$ table, and the semantic functions are generated from a set of accepted if-rules. For a detailed description, see [22].

10.3.2 Learning L-attributed semantic functions

Suppose that the symbols in the rule p can possess synthesized and inherited attributes. The task of an attribute learner is to define semantic functions for the set of defined occurrences $\bigcup_{k\in\{1,\cdots,n_p\}} Inh(X_{p,k}) \cup Syn(X_{p,0})$. The learning process of these attributes can be summarized as follows:

1. **Learning sematic functions for $\mathbf{a} \in \mathbf{Inh}(\mathbf{X_{p,j}}), \mathbf{j} = \mathbf{1}, \ldots, \mathbf{n_p}$.**

   The user is asked to provide examples. For the attribute $X_{p,j}.a$ a $T(a)$ table must be constructed. The columns in the part U of the table are all the attribute occurrences on which the value of $X_{p,j}.a$ depends on $Inh(X_{p,0}) \cup \bigcup_{k\in\{1,\cdots,j-1\}} Syn(X_{p,k})$. Afterwards the propositional part can be constructed in the same way as that presented for S-attributed grammars. When semantic functions for all $a \in Inh(X_{p,j})$ have been learned the attributes of $Syn(X_{p,j})$ can be computed by using the background rules.

2. **Learning semantic functions for $\mathbf{a} \in \mathbf{Syn}(\mathbf{X_{p,0}})$**

   The semantic functions for a can be determined in the same way as that for S-attributed grammars.

Example 22 *An example of AGLEARN method (taken from [22]).*
Applying the above procedure to our example:
$\mathbf{P_T} = \{R(1), R(2), R(3), R(4)\}$
$\mathbf{P_B} = \{R(5), \ldots, R(11)\}$.

$\Im_i = \{add, sub, mul, div, int, real, id\}$ *(f_1 and f_2 are to be learnt)*

We demonstrate the learning of the semantic function of R(3).
- $\mathbf{E_3^+(mode)} = \{(3{*}2.5, real), (5{*}3, int), (1, 5{*}4, real), (2.5/3, real), (2/3, real), (6/3.4, real)\}$

- $\mathbf{E_3^-(mode)}$ ={$(2 * 3.2, int), (4.3/2, int), (8/3, int)$}

Step 1.
Table 3 shows the generated propositional table+ where:
$*$ *: we have to ask this value from the user*
$U_1 : Term_1.mode$,
$U_2 : MulOp.Operator$,
$U_3 : Factor.mode$,
$R_1 : Term_1.mode = int$,
$R_2 : Term_1.mode = real$,
$R_3 : MulOp.Operator = mul$,
$R_4 : MulOp.Operator = div$,
$R_5 : Factor.mode = int$,
$R_6 : Factor.mode = real$,
$F_1 : Term_0.mode = int$,
$F_2 : Term_0.mode = real$,
$C_1 : Term_0.mode = Term_1.mode$,
$C_2 : Term_0.mode = Factor.mode$,
$T : true$, $F : false$

Step 2.
The if-rule :
if $R_1 = true$ & $R_3 = true$ & $R_5 = true$ & *then* $F_1 = true$ *else* $F_2 = true$

Step 3.
The transformed semantic function $R(3)$ *takes the form of*

if $Term_1.mode = int$ & $MulOp.operator = mul$ & $Factor.mode = int$
  then $Term_0.mode := int$
else $Term_0.mode := real$

10.4 The Computational Model of PAGE

One of the objectives of developing the PAGE system was to implement a tool which is portable and independent of the underlying system architecture. PAGE [85] is built on top of the ORCHID kernel [87, 86], which encapsulates the machine dependencies, providing a layer with parallel programming primitives. In PAGE a *supervisor* process maintains a pool of messages, and is responsible for supplying the processors with a computational load (i.e. processes to execute). Each *slave* process handles a grammar production along with its semantic rules. It collects messages corresponding to the RHSS (body) of the production in which it is the head, and produces new messages which correspond to the head of the production in which it is a RHSS. All the static information of the grammar (grammar productions, semantic rules, atomic productions) is broadcast and kept in the network processors for faster access. All the information generated from the slave processes (i.e. partial solutions) is stored locally in the network processors in a caching hierarchy inducing an *incremental attribute evaluation* and achieving a *controlled grained parallelism*.

Example 23 *Let us assume we have to evaluate the following grammar:*

```
1.   S(x,y,z) :- A(x,y), E(y,z).

2.   A(x,y)   :- B(x), C(y).

3.   E(y,z)   :- B(y), D(z).
```

where S is the start symbol
The Supervisor assigns the evaluation of rule (1)to a new slave (for example, slave s1*). Slave* s1 *asks the Supervisor for solutions for the RHSS A and E along with their corresponding inherited attributes. The supervisor checks if there already exist solutions for $A(x, y)$ and $E(y, z)$. If not, it assigns the evaluation of $A(x, y)$ and $E(y, z)$ to* s1 *two new slaves (say* s2 *and* s3*), respectively. This scheme achieves AND parallelism. If there already exist solutions for some or for all of the RHSSs, then there is no need for new slaves corresponding to these RHSSs. As a result PAGE achieves* controlled grained parallelism, *because no unnecessary slaves are generated. Similarly, slave* s2 *asks the supervisor for solutions for the RHSS B and C along with their inherited attributes (i.e., x and y respectively). The Supervisor checks if there exist already solutions for $B(x)$ and $C(y)$. If not, it assigns the evaluation of $B(x)$ and $C(y)$ to two new slaves (say* s4 *and* s5*), respectively. The procedure goes on in the same manner unfolding a proof tree over the network. If we add one more rule to our example*

```
4.   S(x,y,z) :- A(y,x), E(z,y).
```

then another similar proof sub-tree will be generated, establishing OR parallelism.

Fig. 20 illustrates the above example. Each slave may be located in different processors or in the same processor with another slave.

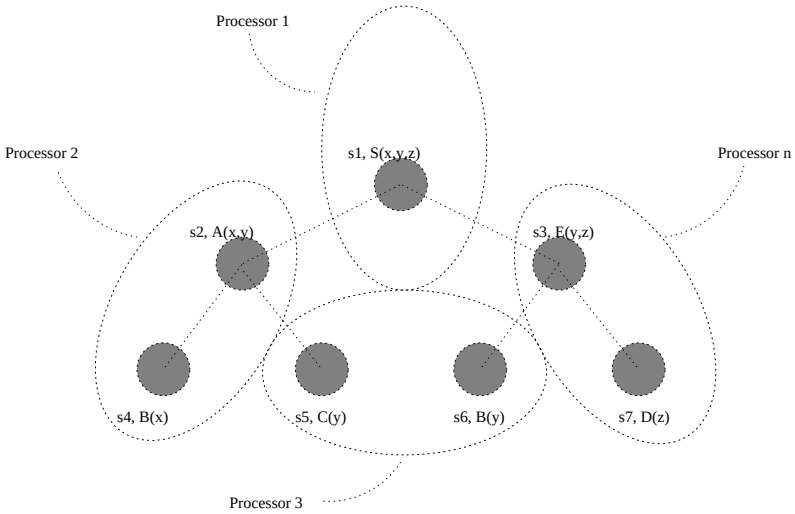

Figure 20: *Slave* Processes generation.

Fig. 21 shows the Network Supervisor processor structure in which the AG is stored. The AG is decomposed and broadcast to the network nodes along with the input string, which is sent to the network nodes. Moreover the network Supervisor process maintains a Process Allocation Table in

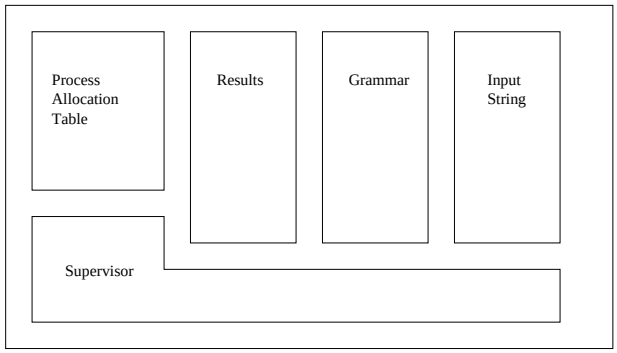

Figure 21: Network Supervisor Structure.

order to prevent an explosion in the number of processes. The results of all the rules evaluated in the network are stored in a special-purpose structure.
Fig. 22 depicts the Node Processor Structure. There is a Node Supervisor Process controlling the underlying slave processes which handles the grammar rules. In addition, the Node Processor keeps a list of the terminals of the AG in order to prevent the overhead communication we would have if this list was maintained by the Network Supervisor Processor. The supervisor process handles local requests for solutions of rules evaluated in remote processes in a special-purpose queue. Besides this every slave process keeps a similar queue of remote requests. The solutions are kept in a caching hierarchy in which every generated or remote accessed solution is stored in a special structure.

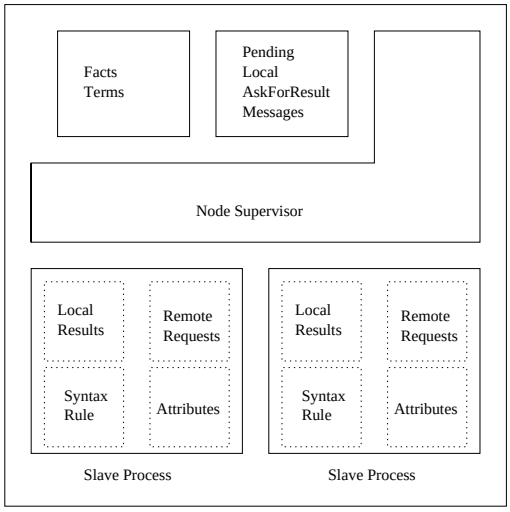

Figure 22: Node Structure.

In general the data-flow style of execution of PAGE assimilate to the computational model of Conery's AND/OR process model, however differs in the following important aspects: We use the above-mentioned structure sharing approach when referring to these structures so there is no need for stack copying which is a major source of overhead in the latter model. Solution caching for incremental attribute evaluation greatly improves our model. Moreover PAGE tries to keep processes belonging to the same OR-branch locally in the same processor. Finally, we use an interleaved unification schema where several solutions of each AND parallel branch are unified at the same time [85].

10.5 PAGELEARN: A parallel approach to AGLEARN

In this section a parallel method is presented for the implementation of AGLEARN using the PAGE general purpose multi-paradigm attribute grammar evaluator. The method is adequate for S-attributed and for L-attributed AGs as well. Parallel learning leads to a more efficient execution time and reduces the oracles that may needed. In the following a description of the method of using S-attribute grammars is supplied. We show how the circuity problem can be handled and give a detailed example. A description of the method using L-attribute grammars is also presented.

10.5.1 Parallel Learning of S-attribute grammars

DESCRIPTION OF THE METHOD Let AG = $< G, S, A, \Im, I >$ the given Attribute Grammar, P_B the set of background rules, P_T the set of target rules, and E the set of the training examples. The target nonterminals posses only synthesized attributes. We would like to learn the semantic functions of each $P \in P_T$ in parallel using PAGE.
In this case the Supervisor processor and the processors have the AG. The rules of P_T are decomposed, and each processor handles one part of the target rules. Every process (slave) learns one semantic function of a rule R. So every rule and every attribute of the rules in P_T is learned in parallel. The process has to build the table T (given in example 22). This means that for every example the corresponding process must compute the columns *class, word, target, U,* F_{UR}, F_{UCR}, F_{UF}, F_{UCF}. It is clear from this procedure that the main issue is the computation of *U*, since the other columns can be derived from the column *U* or target or from the examples.
Let us suppose that R has the form of $R : -X_{p,0} \rightarrow X_{p,1}, \ldots, X_{p,k}$.
To compute *U*, in the previous section, we have to know the value of $X_{p,j}.b \in Syn(X_{p,j}), j = 1, \ldots, k$ (AGLEARN method). For this we build the tree for the actual example, and if the subtree derived from $X_{p,j}$ contains a node corresponding to a rule instance belonging to P_T, then the values of the attributes of $Syn(X_{p,j})$ are asked from the user for the given derivation. Now we have the possibility of asking the other processes if the rule belonging to P_T is already learned or not (because of parallelism). If this is, we can use these semantic functions and we do not need an oracle. If it hasn't been learned yet we have again a possibility of waiting until it is learned (or we can ask the user).
At this point we have to handle a difficult problem. If we decide to wait for these semantic rules to be learned, we may face a circuity situation where two processes wait (perhaps transitively) for one another to produce a semantic rule.
In Fig. 23 we give the general procedure of PAGELEARN

HANDLING THE CIRCUITY PROBLEM The problem can be summarized as follows: Every process learns the semantic rule for an attribute ($R_i.a_j$) in parallel. When the process creates the column *U* for the given training example it has to build the derivation tree for this example, and evaluate this tree. If this tree has a rule that belongs to P_T, then the process has to wait for the semantic rules of this rule. We want to avoid the circuity problem, that is when two or more processes are waiting one for another.

Example 24 *Assume that we have the processes* P_1, P_2, P_3, P_4 *learning the semantic rules* $R1.a, R2.a, R3.a, R4.a$ *respectively. Fig. 24 depicts the circuity problem when the processes are waiting for the learning of the following semantic rules:*

for *each grammatical rule* $p \in P_T$ *and for each synthesized*
occurrence $X_{p,0}.a$ **do** spawn a process in parallel
Every process does the following
for *each example* $e \in E_p(a)$ **do**
build the attributed tree for the actual example on the input string w
using grammar G and semantic functions R
if *the subtree contains only nodes corresponding*
to rule instances R_i *belonging to* P_B **then**
the attributes of this subtree can be evaluated
so the columns of the table T can be computed
else
for *every* $R_i \in P_T$ **do**
the process that evaluates R_i asks the Supervisor if the
unknown rule R has already been learned
if *learned* **then**
the process can evaluate R_i
using the learned semantic rules
else if *there isn't any circuit in the waiting processes* **then**
wait...
else ask the user for an oracle and kill the waiting rules
belonging to this tree. (*These rules are different from the others that are being learned. Moreover, the rules belonging to the tree are used only for their operational semantics, while the rules associated with the other processes may used for inferring the semantics*)
end {for}
end {for example }
As soon as every column and row of table T has been computed,
T is sent to the Supervisor, which then sends it to an attribute value learner
(in our case to C4.5). When the Supervisor gets the learned semantic rules from the
attribute value learner it broadcasts it to the processes.
end {for}

Figure 23: General Algorithm of the PAGELEARN Method.

M	R1	R2	R3	R4
R1	0	0	1	0
R2	0	1	0	1
R3	1	0	0	1
R4	1	0	0	0

Table 4: The circuity problem: Neighbouring Matrix M.

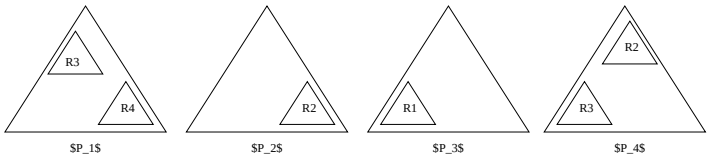

Figure 24: The circuity problem: the waiting processes.

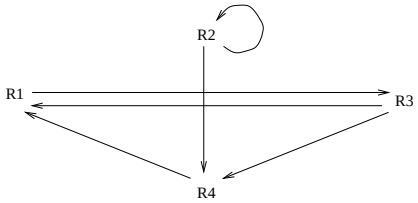

Figure 25: The circuity problem: the dependency graph.

- *P1 is learning $R1.a$ and in the subtree for the given example R3 and R4 are used. (It means that we need the semantic rules R3.a and R4.a to evaluate the value of the attributes of the derivation tree built for the actual example.)*
- *P2 is learning $R2.a$ and in the subtree for the given example R2 is used.*
- *P3 is learning $R3.a$ and in the subtree for the given example R1 is used.*
- *P4 is learning $R4.a$ and in the subtree for the given example R2 and R3 are used.*

In the above example we see that each learning process of a semantic rule may depend on the learning process of another semantic rule of another grammatical rule. These dependences form a dependency graph G. Let $G = (V, E)$ a directed graph, where $V = (R_1, \ldots, R_k)$, $R_j \in P_T$, $j = 1, \ldots, k$,
E = { (R_i, R_j) : There is an edge from R_i to R_j if and only if R_j is waiting for R_i}
In Fig. 25 the dependency graph of the example 24 is shown. In this graph we have three circuits:

- $R1 \rightarrow R3 \rightarrow R4 \rightarrow R1$
- $R2 \rightarrow R2$
- $R1 \rightarrow R3 \rightarrow R1$

In the general algorithm of PAGELEARN (see Fig. 23) we have to check for the circuity problem. For this purpose a neighbouring matrix M is used, situated in the Supervisor, and is similar to that of Table 4. Using this matrix it is easy to detect when a circuity problem is being faced or not. If a circuity problem has come up then it is enough to cut the circuit only at one point (one rule) and ask the user. The checking algorithm is listed in Fig. 26.

Initialize matrix M: fill all elements with 0.
if Pj *(which learns a semantic function of* Rj*) have to wait for* Rk **then**
  Ask the Supervisor if there is a circuit or not
  (check if the addition of the new edge
  (Rk, Rj) in the neighbouring matrix may cause a circuity problem.
  if *there is a circuit* **then**
    the Supervisor sends a message informing that a circuit exists
  else
      Set $M[k, j] = 1$ and the Supervisor sends a message that no circuit exists, so the process can wait for Rk to learn the semantic functions
  if *all semantic rules for* Rk *have been learned* **then**
    they are sent to the waiting processes
    Set $Row(M_k) = 0$

Figure 26: Checking Algorithm for the Circuity Problem

A DETAILED EXAMPLE This section presents a detailed example describing the proposed parallel method using PAGE. Some processes (belonging to a processor) have been assigned the task of learning the semantic rules of a target rule. Other processes belonging to the same processor work on the learning of the semantic rules of another target rule. Every process has to build a table Tij (from which the if-rules are generated). The Supervisor as well as the slave processors know the whole AG specification. To be more precise, each one of the PAGE processes has the following assigned tasks.
The Supervisor:

- Handle the table M that is used for dealing with the circuity problem.
- Decompose the target rules $P \in P_T$.
- Store the learned semantic rules
- Serve the requested messages for the semantic rules or the circuits

The slave processors:

- Spawn and delete processes for the rules and for its attributes
- Forward the requests and the learned semantic functions to the Supervisor
- Handle the requests of the processes for the AGs rules

The processes:

- compute the table T_{ij}.
- Build the derivation tree for the actual example {this is automatic in the PAGE system}

Class	Word	Target	U_1	...	...
+	2.5+3	Real	int	...	...
+				...	...
-				...	...

Table 5: The T(a) table of Example 25

- Ask the Supervisor or other processes or the user for the unknown semantic rules if it is necessary
- Transform the table T_{ij} to the Supervisor

The Supervisor decomposes the target rules and broadcasts them along with the corresponding training examples over the network. The processors spawn processes (slaves) for every semantic rule corresponding to each one of the synthesized attributes of the target rules. The processes start learning the target semantic rules from the given examples. The core tasks of PAGELEARN are: creation of the table $T_{i,j}$, building of the derivation tree for each of the given corresponding examples and, if it is necessary, asking the Supervisor for the target rules for evaluation of the parse tree, or waiting for other processes to produce target semantic rules. When the table $T_{i,j}$ is computed then the process generates the if-rules, transforms them into semantic rules, and sends them to the Supervisor.

Example 25 *We demonstrate how the method works in parallel by continuing the previous example. Recall that* $P_T = \{R(1), R(2), R(3), R(4)\}$.

- *Processor* 2 *handles* $R(1)$ *and* $R(2)$,
- *Processor* 3 *handles* $R(3)$ *and* $R(4)$.
- *Processor* 2 *spawns 2 processes* P_{21} *for* $R(1)$ *and* P_{22} *for* $R(2)$.
- *Processor* 3 *spawns 2 processes* P_{31} *for* $R(3)$ *and* P_{32} *for* $R(4)$.

Fig. 27 shows the mapping of the processes in the processor network.
The rules $R(1)$ - $R(4)$ *are learned in parallel. The positive and negative examples are :*
- $E_1^+(mode) = \{(2.5+3, real), (5+3, int), (1.5-4, real), \ldots\}$
- $E_1^-(mode) = \{2+3.2, int), (4.3-2, int), \ldots\}$
- $E_2^+(mode) = \{(2, int), (3.5, real), \ldots\}$
- $E_2^-(mode) = \{(1.3, int), (2, real), \ldots\}$
- $E_3^+(mode) = \{(3*2.5, real), (5*3, int), (1, 5*4, real), (2.5/3, real), (2/3, real), (6/3.4, real)\}$
- $E_3^-(mode) = \{(2*3.2, int), (4.3/2, int), (8/3, int)\}$
- $E_4^+(mode) = \{(2, int), (3.5, real), \ldots\}$
- $E_4^-(mode) = \{(1.3, int), (2, real), \ldots\}$

Table 5 shows the table $T(mode)$ which is computed in parallel.
To compute the element of the column U an attributed grammar tree is built on the input string w of each example. Fig. 28 shows the attribute tree for the rule R(1). Recall that the rules $R(2)$ and $R(4)$ are evaluated in parallel.

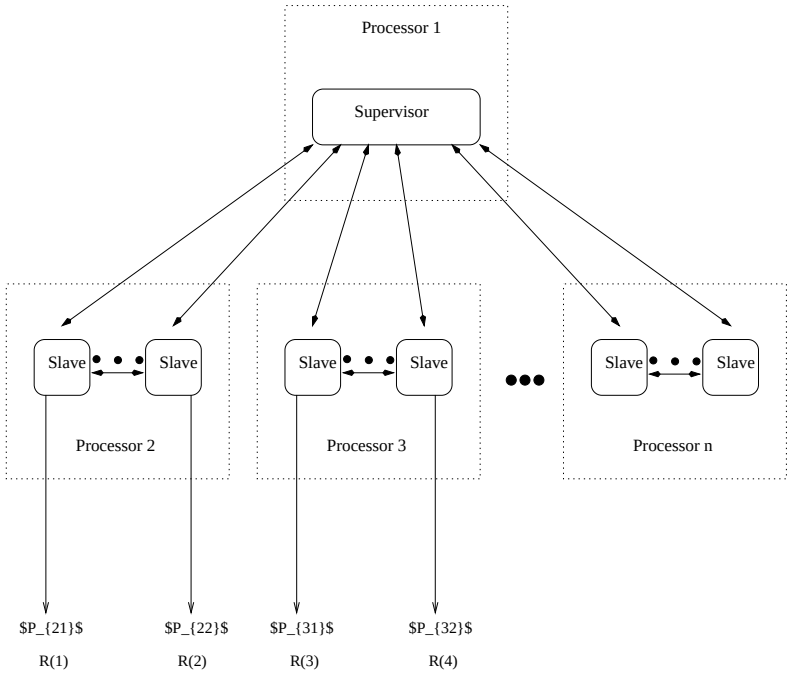

Figure 27: Process mapping for the Example 25

- $R(2)$ and $R(4) \in P_T$ are used in this derivation tree.
  If we build the derivation tree for the given examples we find that
- To learn $R(2)$ we need to use R(4)
- To learn $R(3)$ we need to use R(4)
- To learn $R(4)$ we needn't to use rules from P_T

The corresponding dependency graph for the detection of circuity problem is given in Fig. 29. We do not have a circuit in this graph so P_{22} and P_{31} can wait until $R(4)$ is learned, and P_{21} can wait until R(2) and R(4) are learned. So we needn't ask the user. But when we had to learn sequentially we had to ask the user for every example in R(1), R(2) and R(3).

Example 26 *We give an example to show how there can be a circuit in the dependency graph. Let* $P_T = \{R(3), R(7)\}$. $E_3^+ = \{(3.2 \times (4.1 + 3), real)\}$ *and* $E_7^+ = \{((4.1 * 1.2), real)\}$. *When the process builds the attributed tree for* e_3*, then we have to wait for R(7) to be learned, and when the other process builds the attributed tree for* e_7 *then we have to wait for R(3) to be learned. So we have the circuity problem.*

10.5.2 How the method works for L-attributed grammars

We suppose that a target nonterminal can possess inherited attributes. Consider the production: $p : X_{p,0} \rightarrow X_{p,1} \ldots X_{p,n_p} \in P_T$. The symbols in this rule may possess synthesized and inherited attributes. We have to learn the semantic functions for the set of defined occurrences $\bigcup_{0<k\leq np} Inh(X_{p,k}) \cup Syn(X_{p,0})$.

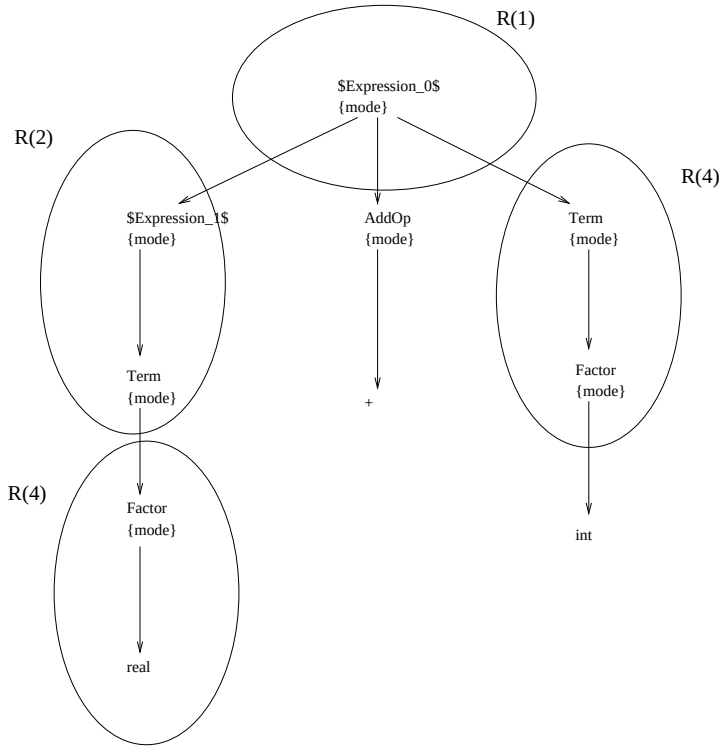

Figure 28: Derivation tree for the Example 25

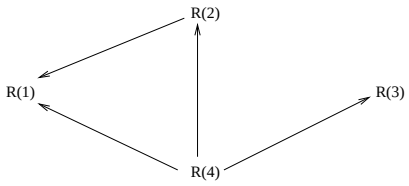

Figure 29: Dependency graph for detecting the circuity problem of Example 25

In our method we learn the semantic functions for every $X_{p,j}$, $j = 1, \ldots, n_p$, and then learn the semantic functions for $X_{p,0}$ sequentially. However, we can learn in parallel the semantic functions for the attributes of a given $X_{p,j}$, $j = 1, \ldots, n_p$, and for $X_{p,0}$ as well since both are independent.

The whole task of process mapping and decomposition of the rules is similar to the parallel learning of the S-attributed grammar. The only difference is that we have to spawn processes for every inherited attribute $\in Inh(X_{p,j}), j = 1, \ldots, n_p$, when their semantic functions are learned, and we have to spawn processes for every synthesized attribute $\in Syn(X_{p,0})$ when we learn the semantic functions of $X_{p,0}$.

The procedure can be summarized like that in Figure 30.

So we learn in parallel the rules in P_T (the set of the target semnatical rules)

For a given rule $p : X_{p,0} \rightarrow X_{p,1} \ldots X_{p,np} \in P_T$ we learn

1. first the semantic functions for the inherited attributes of $X_{p,1}$ in parallel

2. then the semantic functions for the inherited attributes of $X_{p,2}$ in parallel

for *every rule in parallel* **do**
$\{\ p : X_{p,0} \rightarrow X_{p,1} \ldots X_{p,np} \in P_T\ \}$
1. **for** $j = 1, \ldots, n_p$ **do** {sequential }
for *every inherited attribute* $\in Inh(X_{p,j})$ *in parallel* **do** spawn and run Process2
2. **for** *every synthesized attribute* $\in Syn(X_{p,0})$ *in parallel* **do** spawn and run Process1
end {for}

Process1

for *each grammatical rule* $p \in P_T$ *and for each synthesized occurrence* $X_{p,0}.a$ **do**
spawn processes in parallel which do the following
for *each example* $e \in E_p(a)$
build the attributed tree for the actual example on the input string w
using grammar G and semantic functions R
if *the subtree contains only nodes corresponding to rule instances* R_i *belonging to* P_B **then**
the attributes of this subtree can be evaluated,
and the columns of the table T can be computed
else
for *every* $R_i \in P_T$ **do**
the process that evaluates R_i asks the Supervisor if the unknown rule R has already been learned
if *learned* **then** the process can evaluate R_i using the learned semantic rules
else if *there isn't any circuit in the waiting processes* **then** wait...
else ask the user for an oracle and kill the waiting rules belonging to this tree.
end {for}
end{for example }
end {for}

Process2

for *every example* **do**
1. ask the user to give the example for the target attribute $X_{p,j}.a$
2. build the attributed tree for the actual example on the input string w
3. evaluate the part of the tree in order to get the value of
$\bigcup_{k \in \{1, \cdots, j-1\}} Syn(X_{p,k}) \cup Inh(X_{p,0})$
end {for}
5.generate the if rule
6.generate the semantic rule

Figure 30: General algorithm for the L-attributed grammar

.
.
$\mathbf{n_p}$. finally the semantic functions for the inherited attributes of X_{p,n_p} in parallel
$\mathbf{n_p + 1}$. and then the synthesized attributes of $X_{p,0}$ in parallel
For steps $1 - n_p$ to create table T_i , we have to know the value of $\bigcup_{k\in\{1,\cdots,i-1\}} Syn(X_{p,k}) \cup Inh(X_{p,0})$ of which semantic functions are learned in the previous steps.
We learn the rules in parallel, so for a nonterminal both the inherited and the synthesized attributes are learned in parallel because

- If it is in the right hand side its inherited attributes are learned in the actual rule, and its synthesized attributes are learned in another rule of which it is the left hand side symbol (or this rule belongs to P_B).
- If it is in the left hand side, its synthesized attributes are learned in the actual rule, and its inherited attributes are learned in another rule in that it is on the right hand side (or this rule belongs to P_B).

10.5.3 Analysis of the method used

Our method learns semantic functions of Attribute Grammars. During the execution an oracle has to answer questions about the learning problem. So the execution time depends on both the oracles needed in the procedure and the execution time of the program.
Our program is parallelized according to three aspects:

1. Every semantic rule of the target rules is learnt in parallel.
2. The PAGE system 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 offers the possibility of reducing the oracles needed in the procedure. The total elimination of oracles is possible in cases when no circuity situation occurs.
   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 elimination of user queries depends on the training examples, on the number of the circuits appearing among the waiting rules during the learning method.

10.6 Summary

The method presented here is based on the AGLEARN method described in [22]. Parallelism improves the efficiency of the previous method both in execution time and in interaction needed. PAGE is capable of handling the learning of many semantic rules concurrently. OR parallelism PAGE explores the possibility of reducing the oracles needed in the procedure. The total elimination of oracles is possible in cases when no circuity situation occurs.
Moreover, behind parallelism there is a ”dual nature” of the proposed method. The concept of the method described is to give the user the capability of handling smaller specifications

of the grammar describing the problem the user deals with. The natural order to do this is, firstly to learn the semantic rules from the given examples and, secondly, to execute the program. Now we can have both steps interleaving each other in the following fashion: Evaluate the grammar $\rightarrow$ if a semantic rule is needed try to learn it $\rightarrow$ continue the execution. In [3, 94] a learning method called LAG is presented, which infers semantic functions for simple classes of AGs by means of examples and background knowledge. LAG generates the training examples on its own via the effective use of background knowledge. The LAG method makes use of the C4.5 decision tree learner during the learning process. This method has been applied to the Part-of-Speech tagging of Hungarian sentences as well.

11 Summary

See Introduction (Section 3).

The list of the new results (definitions, algorithms and theorems) of this thesis is the following:

I. Data Flow Analysis of Logic Programs:

Definition 28 **Augmented SLD-tree**
Definition 35 **Skeleton(n)**
Definition 36 **Argument Position**
Definition 37 **Map from the nodes of S to the nodes of T**
Definition 38 **Proof Tree Dependence Graph (PTDG: $T_{g,n}$ = ($Pos(S), \sim_T$))**
Definition 39 **Directed Proof Tree Dependence Graph**
Definition 40 **Slice($T_{g,n}, \alpha$)**
Definition 41 **Potentially Dependent Predicate (PDPS)**
Definition 42 **Data_Flow_of_PDPS**
Definition 43 **cut(W_{Mark})**
Definition 44 **Cut(W)_Set**
Definition 45 **Debug Slice**

———————-

Theorem 7 **Theorem: The Potentially Dependent Predicate Set**

II. Data Flow Analysis of Constraint Logic Programs:

Definition 49 **Satisfiability**
Definition 51 **D-Interpretation** (A detailed formalization)
Definition 52 **Semantics of terms** (A detailed formalization)
Definition 53 **Semantics of formulas** (A detailed formalization)
Definition 54 **D-Model** (A detailed formalization)
Definition 55 **The least D-model of a formula** (A detailed formalization)
Definition 56 **Solution** (A detailed formalization)
Definition 59 **SLD-tree** (A detailed formalization)
Definition 60 **Skeleton of a CLP program**
Definition 61 **The set of constraints of a skeleton**
Definition 62 **Derivation tree and proof tree**
Definition 63 **Program Positions**

Definition 64 **Proof-tree Positions**
Definition 66 **Slice of a satisfiable constraint set** C
Definition 67 **A minimal slice of** C
Definition 68 **A slice of a derivation tree**
Definition 69 **A slice of a CLP program**
Definition 70 **Explicit Dependence**
Definition 71 **Dependency Relation**
Definition 73 **Direct Dependency Relation of a Program**
Definition 74 **Directed Dependency Graph of a Proof Tree**

———————-

Theorem 8 **Theorem: Slice of a Constraint Set**
Theorem 9 **Theorem: The relation of** $\sim_T^*$ **to** $dep_{C(T)}$
Theorem 10 **Theorem: A Slice of a Proof Tree**
Theorem 11 **Theorem: A Slice of a Program**
Theorem 12 **Theorem: Directed Slice of a Proof Tree**

III. Learning of Constraint Logic programs:

Definition 75 **The specialization problem**
Definition 76 **The unfolding transformation**
Definition 77 **Operational equivalence**

———————-

Theorem 14 **Theorem: The operational semantics preservation of the unfolding transformation**
Theorem 15 **Theorem: The logical semantics preservation**
Theorem 16 **Theorem: The correctness of the CLP_SPEC algorithm**
Theorem 17 **Theorem: The correctness of the CLP_SPEC_SLICE algorithm**
Section 9.3.4 **Algorithm: The CLP_SPEC algorithm**
Section 9.4.2 **Algorithm: The CLP_SPEC_SLICE algorithm**

IV. Learning Semantic Functions of Attribute Grammars:

Figure 23 **Algorithm: General Algorithm of the PAGELEARN Method**
Figure 26 **Algorithm: Checking Algorithm for the Circuity Problem**
Figure 30 **Algorithm: General algorithm for L-attributed grammar**

12 Összefoglalás

Az imperatív programozási nyelvek mellett egy másik fontos programozási paradigma a deklaratív irány. A deklaratív nyelvek magát a feladatot írják le, azaz azt, hogy ”mit” kell megoldani szemben az imperatív programokkal, melyeknél a hangsúly azon van, hogy ”hogyan” kell megoldani a problémát. Ezért a deklaratív programok sokkal leíróbb jellegűek, közelebb állnak az emberi gondolkodáshoz. A deklaratív nyelvek első implementációja, a LISP programozási nyelv elkészítése McCarthy nevéhez fűződik (az 1960-as évek közepe). Azóta több más programozási nyelv (például: Prolog (1972), Standard ML (1984)) megszületése is segítette a deklaratív nyelvek gyakorlati elterjedését.

Ez a doktori értekezés a deklaratív programozási paradigmákon belül négy fő területtel foglakozik, a **Logikai Programok adatfolyamának analízisével és szeletelésével** [78, 28, 27], **Korlátozásos Logikai Programok adatfolyamának analízisével és szeletelésével** [74, 75, 76], **Korlátozásos Logikai Programok tanulásával** [79], és **Attribútum Nyelvtanok szemantikus függvényeinek tanulásával** [77].

Ezen területek közt szoros kapcsolat áll fenn, a [14] -ben megadott cikk a Logikai Programok Attribútum Nyelvtanokkal történő leírásával foglakozik, a Logikai Programok pedig a Korlátozásos Logikai Programok (CLP) speciális eseteként kezelhetők [32]. Ezen megfeleltetések alapján a Logikai Programok tanulására kidolgozott bizonyos algoritmusok (ILP) alkalmazhatóak Attribútum Nyelvtanok tanulására, illetve némely ILP technikák adoptálhatóak Korlátozásos Logikai Programok tanulására (és viszont).

12.1 Logikai Programok Adatfolyam Analízise

A **Logikai Programozás** alapötlete az volt, hogy deklaratív leírások alapján következményeket lehessen előállítani. Az ilyen leírások (Logikai Programok) logikai formulák véges halmazából épülnek fel. Ahhoz hogy egy egyszerű, de hatékony következtetési szabályt az SLD rezolúciót alkalmazni tudjuk, a logikai nyelven megszorításokat kell bevezetni.
Egy közkedvelt logikai nyelv a *Prolog* [50] az első rendű logikán alapul. Az első rendű formulák egy megszorított formájával, a Horn klózokkal dolgozik.
A **Logikai Programok adatfolyam analízise** fontos szerepet játszik többek közt a nyomkövetésben, tesztelésben és program megértésben.
A **Szeletelés** [29] egy olyan program analízis technika, melyet eredetileg imperatív programokra dolgoztak ki. A szeletelés szintén alkalmazható az előzőekben felsorolt területeken. Egy adott változóra vonatkozó program szelet a program azon részeit tartalmazza, melyek hatással lehetnek a változóra (”visszafele irányuló szeletelés”), illetve amelyekre hatással lehet az adott változó (”előre irányuló szeletelés”).
A Logikai Programok adatfolyama nem explicit, ezért az imperatív programoknál alkalmazott szeletelési módszerek közvetlenül nem adoptálhatóak. Az adatfolyam implicit volta tovább nehezíti a program viselkedésének megértését. Ezért olyan program elemző eszközök, melyek segítik az adatáramlás megértését nagy gyakorlati hasznossággal bírnak. Ebben a témában az egyik kutatási eredmény korábbi Prolog programokra kidolgozott

nyomkövetési módszerek [69] optimalizálása, illetve a program analízis segítése szeletelési technika alkalmazásával. A szeletelési technika egy úgynevezett Program Függőségi Gráfon alapul.
Kidolgozásra került egy olyan **dinamikus szeletelő algoritmus** amely az adatfolyam vizsgálata mellet a kontroll függőségeket is figyelembe veszi, így segítve a program összefüggő komponenseinek megtalálását, illetve a különböző típusú program hibák forrásának feltárását.
Megadunk egy olyan általános szelet definíciót mely az SLD-fa sikeres ágai mellett a sikertelen ágakra is érvényes. Az adatfolyam analízisnek a sikertelen ágakra ilyen módon történő kiterjesztése segíti már létező nyomkövetési technikák hatékonyságának növelését, ”halott” kódrészek megtalálását, a program jobb megértését, párhuzamosítását, stb.
Elkészült egy prototípus is Prolog programok szeletelésére, alkalmazva a fenti meggondolásokat, mely nyomkövetéssel lett kombinálva.
A szerzőnek ezen témához kapcsolódó cikkei a következők: [78, 28, 27].

12.2 Korlátozásos Logikai Programok Adatfolyam Analízise

A doktori értekezésben leírt egy másik eredmény a **Korlátozásos Logikai Programok adatfolyam analízisére** vonatkozik. **A Korlátozásos Logikai Programozás (CLP)** [32, 46] egy egyre jobban fejlődő szoftver technológia igen sokféle alkalmazási lehetősséggel. A hatékonysága abban áll, hogy képes kezelni olyan nehéz kombinatórikus problémákat is, mint amely megjelenik például az ütemezéses, az időosztásos, az útvonal választásos problémáknál, melyek megoldása túlmutat a konvencionális programozási technikák lehetőségein.
A Korlátozásos (Constraint) Logikai Programozás (CLP) két deklaratív programozási paradigma, a constraint megoldás és a Logikai Programozás (LP) fúziójaként jött létre. A klasszikus Logikai Programozás [50, 44] esetén a Herbrand Univerzumban lehet csak mozogni, a CLP esetén a unifikáció egy sokkal általánosabb constraint megoldó mechanizmussal lett helyettesítve. Az imperatív programokra kidolgozott adatfolyam technikák [4, 29, 30, 31] a CLP esetén sem alkalmazhatóak közvetlenül a CLP imperatív jellege miatt. Mivel a Logikai Programozás a CLP egy speciális eseteként kezelhető, ezért az LP -nél alkalmazott szeletelési technikák CLP-re történő adoptálása mélyreható elméleti analízist igényel.
Ezen analízis során formalizálásra került a Korlátozásos Logikai Programok szeletelésének szemantikája. A minimális szelet megtalálása általános esetben az eldönthetetlen problémák kategóriájába tartozik, mivel a CLP szeletelést constraint halmaz kielégíthetőségére vezettük vissza, ami általános esetben eldönthetetlen probléma. A szemantikus definíciók teremtették meg az alapot változó megosztáson alapuló dinamikus, illetve statikus szeletelési technikák kidolgozására. Az algoritmusok további kiterjesztésre kerültek változó lekötési információk figyelembe vételével. A fenti meggondolások implementálásra kerültek egy prototípus szeletelő rendszerben.
Ezen doktori értekezés nem tartalmazza az algoritmusok implementáció szintű formalizálását, a szerző fő célja a pontos elméleti háttér lefektetése volt.
A szerzőnek ezen témához kapcsolódó cikkei a következők: [74, 75, 76].

12.3 Korlátozásos Logikai Programok Tanulása

Az **Induktív Korlátozásos Logika Programozás (ICLP)** egy olyan kutatási irány, mely a Korlátozásos Logikai Programozás (CLP) [32, 46] területén induktív logikai tanulási módszereket alkalmaz (ILP [49, 42]). Az **ILP** esetén a felhasználó az elsőrendű logikai programozás nyelvén megfogalmazott teóriát tanul meg példák alapján, bizonyos háttér tudásra támaszkodva. A logikai programozás nyelvét használva reprezentációs nyelvként az ILP algoritmusok sokkal hatékonyabb tanulást tesznek lehetővé mint a hagyományos elsőrendű logikára kifejlesztett gépi tanulási módszerek, viszont még hátrányuk, hogy nagyon gyengék numerikus információk kezelésében.
A **Korlátozásos Logikai Programozási** nyelvek (CLP) hatékonysága pont abban van, hogy a Logikai Programozási nyelveket kiterjesztik úgy, hogy alkalmasak legyenek numerikus adatok kezelésére is.
A doktori értekezésben ismertetésre kerül egy Korlátozásos Logikai Programok tanulására kidolgozott olyan módszer, mely CLP programok specializálásán alapul. A kidolgozott módszer egy szeleteléssel kombinált változatát is megadjuk. A megadott eljárás lényege, hogy az úgynevezett ”unfolding” (kicsomagolás) transzformációs szabály klóz elhagyással van kombinálva. Hasonló módszer eredetileg Logikai Programokra lett kidolgozva [7]. A Logikai Programok a CLP speciális esetének tekinthetők, a numerikus adatok kezelése nemcsak különböző szintaktikát jelent, hanem más szemantikát is maga után vonz. A doktori értekezésben először is formalizáljuk a specializáláson alapuló tanuló algoritmus alapvető fogalmait CLP-re vonatkozóan, mint például a specializálási problémát, magát az unfolding transzformációt, megadjuk a hozzájuk tartozó szemantikát is. Bebizonyítjuk, hogy az általunk definiált unfolding transzformáció megőrzi a CLP programok műveletei és logikai szemantikáját is. Formalizáljuk magát a CLP_SPEC tanuló algoritmust, és belátjuk annak helyességét. Megadunk egy interaktív javított változatát is a specializáló transzformációnak, mely a CLP_SPEC algoritmust kombinálja algoritmikus nyomkövető technikával és szeleteléssel. Egy olyan prototípus került implementálásra, mely a fenti módszereken alapulva LP és CLP programok tanulására is alkalmas.
A szerzőnek ezen témához kapcsolódó cikke a következő: [79].

12.4 Attribútum nyelvtanok Szemantikus Függvényeinek Tanulása

A Fordító Programok világában az **Attribútum Nyelvtanok (AG)** használata az egyik legszélesebb körben alkalmazott módszer [91, 38, 1]. Az Attribútum Nyelvtanok a Környezet Független Nyelvek [37] általánosításának tekinthetők. Az Attribútum Nyelvtanokat gyakran használják különböző programozási nyelvek specifikálására és implementálására. Mivel az Attribútum Nyelvtanoknak, illetve szemantikus függvényeinek definiálása igen nagy munkával jár, ezért igen hasznos lenne egy olyan eszköz, mely példák alapján képes AG-k szemantikus függvényeinek következtetésére. A doktori értekezésben bemutatunk egy olyan párhuzamos tanuló eljárást, mely az LP és az AG közti kapcsolton alapulva [14] képes AG-k szemantikus függvényeinek tanulására. Egy a Logikai Programokra kidolgozott tanulási módszer lett adoptálva Attribútum Nyelvtanokra a [22] cikkben (Gyimóthy T. és Horváth T.), a doktori értekezésben megadjuk ennek egy párhuzamosított változatát. A párhuzamos tanuló algoritmus kidolgozásra került S-attribútum és L-attribútum nyelvtanokra vonatkozóan is. A párhuzamosság nem csak hatékonyabb végrehajtási időt

eredményez, hanem segít az interaktív algoritmus során a felhasználónak feltett kérdések számát is redukálni.
A szerzőnek ezen témához kapcsolódó cikke a következő: [77].

13 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 in Computer Science 2000, Hungary, pages 43-44.

[6] 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.

[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] Alexin, Z., Zvada Sz. and Gyimóthy, T., 1999. Application of AGLEARN on Hungarian Part-of-Speech Tagging. Proceedings of the Second Workshop on Attribute Grammars and their Applications (WAGA'99), Amsterdam, The Netherlands, pages 133-152.

[4] Atkinson, D.C. and Griswold. W.G., 1996. The Design of Whole-program Analysis Tools. In Proceedings of the 18th International Conference on Software Engineering, Berlin, pages 16-27.

[5] Bates, S. and Horwitz, S., 1993. Incremental Program Testing Using Program dependence Graphs. In Conference Record of the 20th Anual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South California, ACM Press, pages 384-396.

[6] Boström H. and Asker L., 1999. Combining Divide-and-Conquer and Separate-and-Conquer for Efficient and Effective Rule Induction. Proceedings of the Ninth International Workshop on Inductive Logic Programming, LNAI Series 1634, Springer.

[7] 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.

[8] Boye, J. 1991. S-SLD Resolution - An Operational Semantics for Logic Programs with External Procedures. Programming Languange Implementation and Logic Programming, Springer Verlag, LNCS 528, pages 383 - 393.

[9] Boye, J., 1993. Avoiding Dynamic Delays in Functional Logic Programs. Programming Language Implementation and Logic Programming, Springer-Verlag, LNCS 724.

[10] 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.

[11] 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.

[12] Cheng, J., 1997. Dependence Analysis of Parallel and Distributed Programs and its Applications. In Proceedings of International Conference on Advances in Parallel and Distributed Computing (IEEE-CS).

[13] Deransart, P. and Aillaud, C., 2000. Towards a Language for CLP Choice-tree Visualisation. In Deransart, P., Hermenegildo, M. and Małuszyński, J. (editors): *Analysis and Visualization Tools for Constraint Programming,* LNCS. Springer Verlag, pages 209-236.

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

[15] Dershowitz, N. and Jouannaud, J. P., 1990. Rewrite Systems. Handbook of Theoretical Computer Science, Volume B, Elsevier, pages 243 - 320.

[16] Emden, M. H. and Kowalski, R. A., 1976. The Semantics of Predicate Logic as a Program language. Journal of the ACM (1976), 23, pages 733-742.

[17] Farkas, Zs., Futó, I., Langer, T. and Szeredi P., 1989. M-Prolog programozási nyelv. Műszaki Könyvkiadó, Budapest, 1989.

[18] Farkas, Zs., Köves, P. and Szeredi, P., 1993. MProlog: an Implementation Overview. ICLP-Workshops on Implementation of Logic Programming Systems, 1993, pages 103-117.

[19] Ferrante, J., Ottenstein, K.J. and Warren, J. D., 1987. The Program Dependence Graph and its Uses in Optimization. ACM Transactions on Programming Languages and Systems, 9(3), (July 1987), pages 319-349.

[20] Forgács, I. and Gyimóthy, T., 1997. An Efficient Interprocedural Slicing Method for Large Programs. In Proceedings of SEKE'97, Madrid, pages 279-287.

[21] 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.

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

[23] Gyimóthy, T. and Paakki, J.,1995. Static Slicing of Logic Programs. In Proceedings of the Second International Workshop on Automated and Algorithmic Debugging (AADEBUG'95), Saint Malo, France, pages 85-105.

[24] Hanus, Michael, 1994. The Integration of Functions into Logic Programming: from Theory to Practice. The Journal of Logic Programming, volume 19, Number 20, pages 583 - 628.

[25] Hanus, Michael, 1997. A Unified Computation Model for Functional and Logic Programming. In Proceedings of 24th ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages, pages = 80 - 93.

[26] Hanus, Michael, 1997. Parallel Evaluation Strategies for Functional Logic Programming. In Proceedings of the Fourteenth International Conference on Logic Programming, The MIT Press.

[27] 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.

[28] 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.

[29] 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.

[30] Horwitz, S., Reps, T. and Binkley, D., 1990. Interprocedural Slicing Using Dependence Graphs. ACM Transactions on Programming Languages and Systems 12, pages 26-61.

[31] Horwitz, S., Reps, J. T. and Binkeley, D.,1988. Interprocedural Slicing Using Dependence Graphs. Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation, SIGPLAN Notices, 23(7), pages 35-46.

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

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

[34] Kamkar, M. and Fritzson, P., 1995. Evaluation of Program Slicing tools. In Proceedings of the Second International Workshop on Automated and Algorithmic Debugging (AADEBUG'95), Saint Malo, France, pages 51-69.

[35] Kawamura T. and Furakawa K., 1993. Towards inductive generalizations in constraint logic programs. In Proceedings of the IJCAI-93 workshop on inductive logic programming, France, Academic Press, pages 93-104.

[36] Knuth, Donald E., 1968. Semantics of Context-Free Languages. Mathematical Systems Theory, Volume 2, Number 2, pages 127 -145.

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

[38] 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.

[39] Kókai, G., Harmath, L. and Gyimóthy, T., 1997. Algorithmic Debugging and Testing of Prolog Programs. In Proceedings of the Fourteenth International Conference on Logic Programming, Eighth Workshop on Logic Programming Environments (ICLP'97), Leuven, Belgium, pages 14-21.

[40] Korel, B. and Rilling, J., 1997. Application of Dynamic Slicing in Program Debugging. In Proceedings of the Third International Workshop on Automatic Debugging (AADEBUG '97), Linköping, Sweden, pages 43-59.

[41] Kowalski, R.A., 1974. Logic for Problem Solving. DCL Memo 75, Department of Artificial Intelligence, University of Edinburgh, March, 1974.

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

[43] Lewi, J. and De Vlaminck, K. and Steegmans E. and Van Horebeek, J., 1992. Software development by LL(1) syntax description. John Wiley and Sons.

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

[45] Małuszyński, J. and Bonnier, S. and Boye, J. and Kluzniak, F. and Kagedal, A. and Nilsson, U., 1993. Logic Programs with External Procedures. Logic Programming Languages - Constraints, Functions and Objects. The MIT Press, pages 21 - 48.

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

[47] Martin L. and Vrain C., 1996. Induction of constraint logic programs. In Proceedings of Algorithms and Learning Theory (ALT), 1996, Sydney, Australia. Springer Verlag.

[48] Mizoguchi F. and Ohwada H., 1995. Constrained relative least general generalization for inducing constraint logic programs. New Generation Computing.

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

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

[51] Ottenstein, K. J. and Ottenstein, L. M., 1984. The Program Dependence Graph in a Software Development Environment. In Proceedings of the ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments, 1984, SIGPLAN Notices, 19(5), pages 177-184.

[52] Paakki, J., 1994. Multi-Pass Execution of Functional Logic Programs. In Proceedings of 21th ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages ('94), pages 361 - 374.

[53] Paakki, J., 1990. A Logic-Based Modification of Attribute Grammars for Practical Compiler Writing. In the Proceedings of the Seventh International Conference on Logic Programming (D.H.D.Warren, P.Szeredi, eds.), Jerusalem, 1990. The MIT Press, pages 203-217.

[54] Paakki J., 1991. PROFIT: A System Integrating Logic Programming and Attribute Grammars. In the Proceedings of the Third International Symposium on Programming Language Implementation and Logic Programming, (PLILP'91) (J.Maluszynski,

M.Wirsing, eds.), Passau, 1991. Lecture Notes in Computer Science 528, Springer-Verlag, pages 243-254.

[55] Paakki J., 1995. Attribute Grammar Paradigms - A High-Level Methodology in Language Implementation. ACM Computing Surveys 27, 2, 1995, pages 196-255.

[56] Paakki J., Gyimóthy T., Horváth T., 1995. Independent And- Parallelization of Logic Programs Using Static Slicing. In the Proceedings of the 4th Symposium on Programming Languages and Software Tools (L.Varga, ed.), Visegrád, Hungary, 1995, pages 302-311.

[57] Paakki J., Gyimóthy T., Horváth T., 1994. Effective Algorithmic Debugging for Inductive Logic Programming. In the Proceedings of the 4th International Workshop on Inductive Logic Programming (ILP-94) (S.Wrobel, ed.), Bad Honnef/Bonn, 1994, pages 175-194.

[58] Page C. D. and Frisch A.M., 1991. Generalizing atoms in constraint logic. In Proceedings of Second International Conference on Knowledge Representation and Reasoning.

[59] Papakonstantinou, G. and Tsanakas, P., 1988. Attribute grammars and dataflow computing. Information and software technology, Volume 30, Number 5, pages 306 - 313.

[60] Papakonstantinou, G. and Panayiotopoulos, T. and Dimitriou, G., 1992. AGP: a parallel processor for knowledge and software engineering. The Computer Journal, Vol. 35, Num. 2.

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

[62] Plotkin, G.D., 1971. A note on inductive generalization. Machine Intelligence, pages 5:101-124.

[63] Quinlan, J.R., 1993. Programs for Machine Learning. Morgan Kaufmann.

[64] Reps, Thomas, 1993. Scan Grammars: Parallel Attribute Evaluation via Data Parallelism. In Proceedings of the 5th ACM Symposium on Parallel Algorithms and Architectures, ACM, 1993.

[65] Roussel, P., 1975. Prolog : Manuel de Reference et d’Utilisation. Groupe d’Intelligence Artificielle, Marseille-Luminy, 1975.

[66] Schoening, S. and Ducassé, M., 1996. A Backward Slicing Algorithm for Prolog. In Proceedings of the Third International Static Analysis Symposium (SAS’96), LNCS 1145, Springer-Verlag, pages 317-331.

[67] Sebag M. and Rouveriol C., 1994. Induction of maximal general clauses compatible with intergrity constraints. In Proceedings of Fourth International Workshop on Inductive Logic Programming, 1994.

[68] Shan-Hwei Nienhuys Cheng and Ronald de Wolf, 1997. Foundations of Inductive Logic Programming. Springer-Verlag, Berlin, Hiedelberg.

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

[70] Steindl, C., 1998. Intermodular Slicing of Object-oriented Programs. In International Conference on Compiler Construction (CC'98).

[71] Somogyi, Z., Henderson, F., Conway, T., 1996. The Execution Algorithm of Mercury, an Efficient Purely Declarative Logic Programming Language. JLP 29(1-3), pages 17-64.

[72] http://www.sics.se/sicstus.html

[73] Szeredi P., Carlsson M. and Yang R., 1991. Interfacing engines and schedulers in or-parallel Prolog systems. In E.H.L. Aarts, J. van Leeuwen, and M. Rem, editors, PARLE'91, Conference on Parallel Architectures and Languages Europe, Volume 506 of LNCS. Springer-Verlag.

[74] 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.

[75] 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.

[76] 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*.

[77] 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.

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

[79] 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.

[80] Tessier, A. and Fèrrand, G.2000. Declarative Diagnosis in the CLP Scheme. In: P. Deransart, M. Hermenegildo, and J. Małuszyński, (editors), Analysis and Visualization Tools for Constraint Programming, LNCS. Springer Verlag, pages 151-173.

[81] Tip, F., 1995. A survey of Program Slicing Techniques. Journal of Programming Languauges, Vol.3, No.3, pages 121-189.

[82] Trachanias, P. and Skordalakis, E., 1990. Syntactic Pattern Recognition of the ECG. IEEE transactions on Pattern Recognition and Machine Intelligence, Volume 12, Number 7, pages 648 - 657.

[83] Vander Zanden T., Bradley, 1988. Constraint Grammars in user interface management systems. In Proceedings of the Graphics Interface Conference, LNCS, 1988.

[84] Vander Zanden T., Bradley, 1988. Incremental Constraint Satisfaction and its applications to graphical interfaces, PhD Thesis. Department of Computer Science, Cornell University, 1988.

[85] Voliotis, C. and Thanos, A. and Sgouros, N. and Papakonstantinou, G., 1995. Daffodil: A Framework for Integrating AND/OR Parallelism. In Proceedings of 5th Hellenic Conference on Informatics, Athens.

[86] Voliotis, K. and Manis, G. and Thanos, A. and Papakonstantinou, G. and Tsanakas, P., 1995. Facilitating the Development of Portable Parallel Applications on Distributed Memory Systems. In Proceedings of Massively Parallel Programming Models MPPM-95 conference, Berlin, IEEE Computer Society Press.

[87] Voliotis, K. and Manis, G. and Lekatsas, H. and Tsanakas, P. and Papakonstantinou, G. ORCHID: A potrable platform for parallel programming. Euromicro Journal of Systems Architecture.

[88] Voliotis, C. and Sgouros, Nikitas M. and Papakonstantinou, G., 1995. Attribute Grammar Based Modeling for Concurrent Constraint Logic Programming. International Journal on Artificial Intelligence Tools, volume 4, number 3, pages 383 - 411.

[89] Weiser, M., 1982. Programmers use slices when debugging. Communications of the ACM, July 1982, pages 446-452.

[90] Weiser, M., 1984. Program Sicing. In Proceedings of Transactions on Software Engineering (IEEE), July 1984, pages 352-357.

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

[92] Zaring K., Alan, 1990. Parallel Evaluation in Attribute Grammar-Based Systems, PhD Thesis. Department of Computer Science, Cornell University, 1990.

[93] Zhao, J., Cheng, J. and Ushijima, K., 1997. Slicing Concurrent Logic Programs. In Proceedings of the Second Fuji International Workshop on Functional and Logic Programming, pages 143-162.

[94] Zvada Sz. and Gyimóthy, T., 2001. Using Decision Trees to Infer Semantic Functions of Attribute Grammars. Acta Cybernetica, Vol. 15, No.2.