Annales Univ. Sci. Budapest., Sect. Comp. 31 (2009) 235-258

LEARNING OF CONSTRAINT LOGIC PROGRAMS BY COMBINING UNFOLDING AND SLICING TECHNIQUES [1]

Gy. Szilágyi Kocsisné

(Budapest, Hungary)

Abstract. This paper discusses learning of Constraint Logic Programs using unfolding and slicing technique. The transformation rule for unfolding together with clause removal is a method for specialization of Logic Programs. Slicing is a program analysis technique originally developed for imperative languages. It facilitates the understanding of data flow and debugging.
This paper formulates the semantics of a learning method of CLP programs, proves that the unfolding transformation preserves the operational and logical semantics, and combines the defined unfolding technique by applying slicing. A prototype learner of CLP programs which implements the above ideas is briefly described.

1. Introduction

Inductive Constraint Logic Programming (ICLP) is a research topic on the intersection of Constraint Logic Programming (CLP) [7, 9] and Inductive Logic Programming (ILP) [13]. In this paper we present an ICLP algorithm based on the specialization of Constraint Logic Programs. This paper formulates the semantics of a specialization method of CLP programs, proves that the unfolding transformation preserves the operational and logical semantics. An other result is that an improved interactive version of the learning algorithm has been

[1]This work was supported by GVOP-3.2.2.-2004-07-0005/3.0 ELTE IKKK

defined integrating an algorithmic debugging algorithm and the slicing method with the specialization algorithm for CLP programs. A prototype tool has been implemented for both algorithms.

An ILP method 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 [3] specializes clauses defining a target predicate by applying different strategies for selecting the literal to apply unfolding upon (e.g. taking the left-most literal, selecting randomly or using the impurity measure [2]). The main idea behind an improved method 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 [1] (have been worked out for Logic Programs) uses a debugging algorithm to identify a buggy clause instance which is then unfolded and partially removed from the initial program. This solution improves the original learning algorithm but it has one major drawback, namely that an oracle has to answer membership questions to identify a buggy clause instance. In our algorithm 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 negative example, 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 paper is organized as follows. Section 2 outlines and formalizes some basic concepts which are then used to formulate and analyze the specialization methods. Section 3 presents the CLP_SPEC algorithm, the associated definitions (specialization, unfolding) and theorems (operational and logical semantics preservation, correctness analysis). Section 4 discusses our main results 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 5. Section 6 provides a comparison with other works and suggestions for future work.

2. Preliminaries

The cornerstone of **Constraint Logic Programming (CLP)** [7, 9] is the notion of a constraint. **Constraint atoms** are formulae constructed with some *constraint predicates* with a predefined interpretation. A **clause** is a formula of the form $h \leftarrow b_1, ..., b_n,\ n \geq 0$, where $h, b_1, \ldots, b_n$ are *atomic formulae*. The predicates used to construct $b_1, \ldots, b_n$ 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 $h \leftarrow c_1, \ldots, c_n$ where $c_1, \ldots, c_n$ are constraints. A **constraint logic program** is a set of *clauses*. The detailed definitions of the valuation, D-interpretation, and D-model can be found in [7]. The least D-model of a set of formula P is denoted by $lm(P, \Im)$. A solution to a query G is a valuation φ such that $\varphi(G) \subseteq lm(P, \Im)$.

The **top-down operational semantics** of constraint logic programs P can be seen as a transition system on states, tuples $\langle A, C, S \rangle$ where A is a multiset of atoms and constraints, and C and S are multisets of constraints [7]. 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 (or are awake), and S is a collection of constraints playing a passive role (or 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 $R_j : h_j \leftarrow b_{j_1}, \ldots, b_{j_{m_j}}, c_j$ $(j = 1, \ldots, n)$, where $h_j, b_{j_1}, \ldots, b_{j_{m_j}}$ 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:

1. $\langle A \cup b, C, S \rangle \rightarrow_r \langle A \cup \{b_{j_1}, \ldots, b_{j_{m_j}}, c_j\}, C, S \cup (\overline{b} = \overline{h_j}) \rangle$ if b is a defined atom selected by the computation rule, $h_j \leftarrow b_{j_1}, \ldots, b_{j_{m_j}}, c_j$ 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.
2. $\langle A \cup b, C, S \rangle \rightarrow_r$ *fail* if b is a defined atom selected via the computation rule, and for every rule $h_j \leftarrow b_{j_1}, \ldots, b_{j_{m_j}}, c_j$ of P, h_j and b have different predicate symbols.
3. $\langle A \cup c, C, S \rangle \rightarrow_c \langle A, C, S \cup c \rangle$ if c is selected by the computation rule and c is a constraint.
4. $\langle A, C, S \rangle \rightarrow_i \langle A, C', S' \rangle$ if $(C', S') = \mathit{infer}(C, S)$.
5. $\langle A, C, S \rangle \rightarrow_s \langle A, C, S \rangle$ if $\mathit{consistent}(C)$.
6. $\langle A, C, S \rangle \rightarrow_s \mathit{fail}$ if $\neg \mathit{consistent}(C)$.

The predicate $\mathit{consistent}(C)$ expresses a test for the consistency of C. The function $\mathit{infer}(C, S)$ 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 resolution, $\rightarrow_c$ transitions introduce constraints into the constraint solver, $\rightarrow_s$ transitions test wether the active constraints are consistent, and $\rightarrow_i$ transitions infer more active constraints from the current collection of constraints. 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 $\langle\emptyset, C, S\rangle$. Let G be a goal with free variables $\widetilde{X}$, which initiates a derivation and produces a final state $\langle\emptyset, C, S\rangle$, and denote $\exists_{-\widetilde{X}} Q$ 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. We should note 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 SLD-tree.

Definition 1 (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:*

1. *Each node label is a computational state $\langle A, C, S\rangle$ like that defined above.*
2. *The root node is $\langle G, \emptyset, \emptyset\rangle$.*
3. *Each node has as many children as valid transitions are associated with it.*
4. *Final states have no children.* **5.** *the edges are labeled by the type of the transition (r, c, i, s).*

We note that every branch of the SLD-tree describes one derivation.

Let $\Im$ be a D-interpretation, φ a valuation and $F = \{a \leftarrow c\}$ a set of facts, where a is a defined predicate and c is a conjunction of constraint atoms. Then $[F]_{\Im} := \{\varphi(a) \mid (a \leftarrow c) \in F, \Im \models \varphi(c)\}$. The following theorem [7] describes the connection between the top-down operational and logical semantics of a CLP program, which is then used to prove that the unfolding transformation preserves the logical semantics.

Theorem 1. *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.*

3. Specialization of CLP programs by unfolding

3.1. The unfolding transformation

3.1.1. The definition of the unfolding transformation

The algorithm CLP_SPEC specializes logic programs with respect to positive and negative examples by applying the transformation rule **unfolding** together with **clause removal**. **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 labeled 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 either predicate symbols defined in the background knowledge or constraint symbols.

We deal here with the operational and logical semantics defined by Jaffar and Maher [7], which can be used for combining unfolding and slicing. This is the reason for giving a direct proof of the correctness of the specialization method.

Definition 2 (The specialization problem. 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.

In Figure 1, the skeleton of the SLD-tree of Example 1 is shown whose leaves that correspond to refutations of positive and negative examples are labeled '+' and '-' respectively, and leaves that do not correspond to refutations of any examples are left unlabeled. 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.

Definition 3 (The unfolding transformation). *Let P be a CLP program with the rules $R_1, \ldots, R_n$, such that*

$$R_j : h_j \leftarrow b_{j_1}, \ldots, b_{j_{m_j}}, c_j \quad (j = 1, \ldots, n),$$

where $h_j, b_{j_1}, \ldots, b_{j_{m_j}}$ are defined predicates, c_j denotes the conjunction of the atomic constraints appearing in the body of R_j. Let

$$R : h \leftarrow b_1, \ldots, b_m, \ldots, b_k, c$$

be a program clause in P, and $\overline{R} = \{R_1, \ldots, R_q\}$ be a set of program clauses renamed to new variables such that the head of each $R_i \in \overline{R}$ ($i = 1, \ldots, q$) and

b_m have the same predicate symbol, and b_m is selected by some computation rule. Then the program P' after unfolding is:

$$P' = \mathit{Unf}(P, R, b_m) =$$

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

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

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

$$\mathit{Res}(P, R, b) :=$$

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

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

In Example 1 an unfolding step with respect to $\mathit{main}(M, J)$ in clause 1 is shown.

3.1.2. The operational semantics preservation of the unfolding transformation

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 4. (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 $\Im \models \exists_{-\widetilde{X}} C' \wedge C'' \Leftrightarrow \Im \models \exists_{-\widetilde{X}} \overline{C}' \wedge \overline{C}''$.*

Theorem 2. *Let P be a CLP program with the rules $R_1, \ldots, R_n$, such that*

$$R_j : h_j \leftarrow b_{j_1}, \ldots, b_{j_{m_j}}, c_j, \quad (j = 1, \ldots, n),$$

where $h_j, b_{j_1}, \ldots, b_{j_{m_j}}$ *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 1). 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.

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

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

$$\begin{array}{rcll} R : h \leftarrow b_1, \ldots, & \underbrace{b_m} & , \ldots b_k, c & \text{was "unified" with} \\ R_j : & \overbrace{h_j} & \leftarrow b_{j_1}, \ldots, b_{j_{m_j}}, c_j & (j = 1, \ldots, n). \end{array}$$

Compare an $\rightarrow_r$ transition (3.1) with an unfolding step (3.2):

$$\langle A \cup b_m, C, S\rangle \rightarrow_r \langle A \cup \{\underbrace{b_{j_1}, \ldots, b_{j_{m_j}}, c_j}_{\text{body}(R_j)}\}, C, S \cup \underbrace{(\overline{b_m} = \overline{h_j})}_{\text{argument equations}}\rangle \tag{3.1}$$

$$R_{new_j} : h \leftarrow \overbrace{(\overline{b}_m = \overline{h}_j)}^{\text{argument equations}}, b_1, \ldots, b_{m-1}, \overbrace{b_{j_1}, \ldots, b_{j_{m_j}}, c_j}^{\text{body}(R_j)}, b_{m+1}, \ldots, b_k, c \tag{3.2}$$

One $\rightarrow_r$ transition can be viewed as an operation when, 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.

Derivation R shows a part of the refutation when R is utilized as an input clause.

Derivation $Res(R)$ shows when $Res(P, R, b_m)$ is used instead of R as an input clause (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. $\ldots \langle b, GRem; C_1\rangle \vdash_r$
2. $\langle b_1\sigma, \ldots, b_{m-1}\sigma, b_m\sigma, b_{m+1}\sigma, \ldots, b_k\sigma, c\sigma, GRem; C_1 \wedge (\overline{b} = \overline{h\sigma})\rangle \vdash_{(r\cup c)^*}$
3. $\langle b_{m-1}\sigma, b_m\sigma, b_{m+1}\sigma, \ldots, b_k\sigma, c\sigma, GRem; C_2\rangle \vdash_{(r\cup c)^*}$
4. $\langle b_m\sigma, b_{m+1}\sigma, \ldots, b_k\sigma, c\sigma, GRem; C_3\rangle \vdash_r$
5. $\langle b_{j_1}\sigma, \ldots, b_{j_{m_j}}\sigma, c_j\sigma, b_{m+1}\sigma, \ldots, b_k\sigma, c\sigma, GRem; C_3 \wedge (\overline{b_m\sigma} = \overline{h_j\sigma})\rangle \vdash_r \ldots$

Derivation Res(R):

1. $\ldots \langle b, GRem; C_1\rangle \vdash_r$
2. $\langle b_1\sigma, \ldots, b_{m-1}\sigma, (\overline{b_m} = \overline{h_j})\sigma, b_{j_1}\sigma, \ldots, b_{j_{m_j}}\sigma, c_j\sigma, b_{m+1}\sigma, \ldots, b_k\sigma, c\sigma, GRem;$
$C_1 \wedge (\overline{b} = \overline{h\sigma})\rangle \vdash_{(r\cup c)^*}$
3. $\langle b_{m-1}\sigma, (\overline{b_m} = \overline{h_j})\sigma, b_{j_1}\sigma, \ldots, b_{j_{m_j}}\sigma, c_j\sigma, b_{m+1}\sigma, \ldots, b_k\sigma, c\sigma, GRem; C_2\rangle \vdash_{(r\cup c)^*}$
4. $\langle(\overline{b_m} = \overline{h_j})\sigma, b_{j_1}\sigma, \ldots, b_{j_{m_j}}\sigma, c_j\sigma, b_{m+1}\sigma, \ldots, b_k\sigma, c\sigma, GRem; C_3\rangle \vdash_c$
5. $\langle b_{j_1}\sigma, \ldots, b_{j_{m_j}}\sigma, c_j\sigma, b_{m+1}\sigma, \ldots, b_k\sigma, c\sigma, GRem; C_3 \wedge ((\overline{b_m} = \overline{h_j})\sigma)\rangle \vdash_r \ldots$

For these derivations we employ the following notational conventions:

1. We did not picture the i and s transitions, we only denoted 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 an s transitions are also the same).
2. Every node has the following form: $\langle goals, C\rangle$, where $goals$ contains the actual list of goals and C is the constraint store.
3. R is the clause and b_m is the literal in R to be unfolded upon.
4. 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.

5. 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 $b_1\sigma, \ldots, b_{m-2}\sigma$. 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 were 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 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.

3.1.3. The logical semantics preservation of the unfolding transformation

Theorem 3. *The unfolding transformation preserves the logical semantics (D-semantics) of CLP programs.*

Proof. We use here the notes of Theorem 2. Let $\Im$ be a D-interpretation.
From Theorem 1: $[SS(P)]_\Im = lm(P, \Im)$ and $[SS(P')]_\Im = lm(P', \Im)$.
From Theorem 2: $SS(P) = SS(P')$.
So, $lm(P, \Im) = [SS(P)]_\Im = [SS(P')]_\Im = lm(P', \Im)$.
From which: $lm(P, \Im) = lm(P', \Im)$.

3.2. The CLP_SPEC algorithm

3.2.1. The definition of CLP_SPEC algorithm

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 checked 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.

The **CLP_SPEC 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 [2].

The input of the algorithm: 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).

The output of the algorithm: Series of programs $P'^{(0)}, P'^{(1)}, \ldots, P'^{(n)} (P'^{(0)} = P)$, 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**
6. **if** R does not cover any atom in E^+ **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).

3.2.2. The correctness of the CLP_SPEC algorithm

Theorem 4 (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 2 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 our future work is to combine this algorithm with a constraint inferring method [5, 10, 14].

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 $i = 1, \ldots, n$

2.a. The unfolding step in program line 8 does not change $lm(P'^{(i)}, \Im)$ (see Theorem 3).

2.b. 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 $i = 1, \ldots, n$:

3.a. The unfolding step in program line 8 does not change $lm(P'^{(i)}, \Im)$ (see Theorem 3).

3.b. 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 1). From which,

$$M_{P'(i+1)} \subseteq M_{P'(i)} \text{ for } i = 0, \ldots, n, \text{ so } M_{P'} = M_{P'(n)} \subseteq M_{P'(0)} = M_P.$$

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 iteration should be kept as low as possible. To do this different computation rules can be employed (for more details see [3, 2]).

3.2.3. An example to illustrate the specialization algorithm

Example 1. A simple example has been chosen to simplify the illustration of the specialization algorithm. Given the definition of a fish-meal as consisting of an appetizer, 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.

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

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

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: $fishlightmeal(A, M)$.
Figure 1 shows 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 (1+4 and 1+5) are the following:
1.4 `fishlightmeal(A,M):-`$\{I + J \leq 10\}$`,appetizer(A,I),fish(M,J),`$\{J > 0\}$.
1.5 `fishlightmeal(A,M):-`$\{I + J \leq 10\}$`,appetizer(A,I),meat(M,J),`$\{J > 0\}$.
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):-`$\{I+J \leq 10\}$`,appetizer(A,I),fish(M,J),`$\{J>0\}$`.`
2. `appetizer(A,I) : - cheese(A,I) ,` $\{I>0\}$`.` 4. `fish(sole,2).`
3. `appetizer(A,I) : - pasta(A,I),` $\{I>0\}$`.` 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.

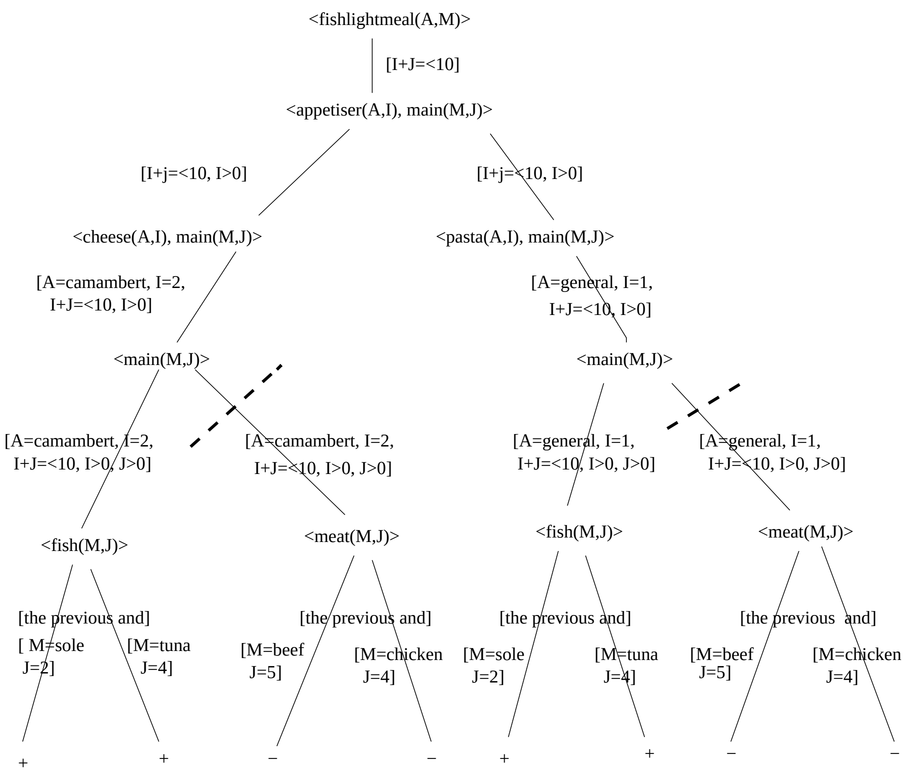

Figure 1. The skeleton of the SLD-tree of $P'^{(0)}$ for the goal $fishlightmeal(A,M)$

4. Improving the CLP_SPEC Algorithm by Slicing

4.1. The skeleton of a CLP program

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 [1]. The number of applications of unfolding should be kept as low as possible, since the number of clauses increases when unfolding is applied. 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 slicing technique is used to find the clause to be unfolded.

Slicing is a program analysis technique originally developed for imperative languages [22]. It facilitates the understanding of data flow and debugging. As slicing concerns computations we now introduce some relevant notions. Abstractly, *a computation of a CLP program* can be seen as the construction of a tree (skeleton) from renamed instances of clauses. We will now briefly explain the idea formally discussed in [21].

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

1. with the root labelled by a goal clause and
2. with the nodes labelled by clause instances of the program; some leaves may instead be labelled "?", in which case they are called *incomplete nodes.*
3. Each non-leaf node has as many children as the non-constraint atoms of its body.
4. 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.

We note that a derivation (proof) tree is a special kind of skeleton. Figure 2 shows a complete skeleton tree for the program in Example 1.

In order to properly present the slicing techniques used here we first need to mention **program positions** and **skeleton positions**. A slice is defined with respect to some particular occurrence of a variable (in a program or skeleton), and positions are used to identify these occurrences [21]. The set of all positions of a skeleton tree T is denoted by $Pos(T)$. Note that each label of a skeleton tree T is a variant of a program clause, or a goal. Thus the positions of T can be mapped in a natural way into the corresponding program positions.

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*). The slice then provides a focus for analysis of the origin of the computed values of the variable in question. A precise formulation of the slicing problem for CLP programs and different slicing techniques based on a simple analysis of variable sharing and groundness can be found in [21].

4.2. An overview of slicing and automatic debugging of constraint logic programs

We would now like to provide a brief overview of the slicing of constraint logic programs.
To construct slices of derivation trees we introduce a dependency relation on the positions of a derivation tree (skeleton).

Derinition 5. (The formal definition of the proof tree dependence relation). *Let T be a derivation tree, $\alpha, \beta \in Pos(T)$. Denote the direct dependency relation $\sim_T$ 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. The dependency relation of a proof tree can be represented as a graph called the proof tree dependence graph (PTDG). The nodes of PTDG are the proof tree (skeleton) positions, and there is an edge between two positions if they are directly dependent. This graph represents the data flow of a CLP program. A slice of a proof tree contains a connected part of the PTDG (of the proof tree). Directionality information can be introduced into the PTDG to make the slice more precise [21]. A formal syntactical definition of the slice is the following.

Definition 6 (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 called a slice of T with respect to α.*

A semantical meaning of this slice definition can be given which relates the proof tree dependence relation to the dependencies in the constraint store [21]. Figure 2 shows the program dependencies and a backward slice of the program in Example 1 for the goal $lightmeal(A, M)$ with respect to A.

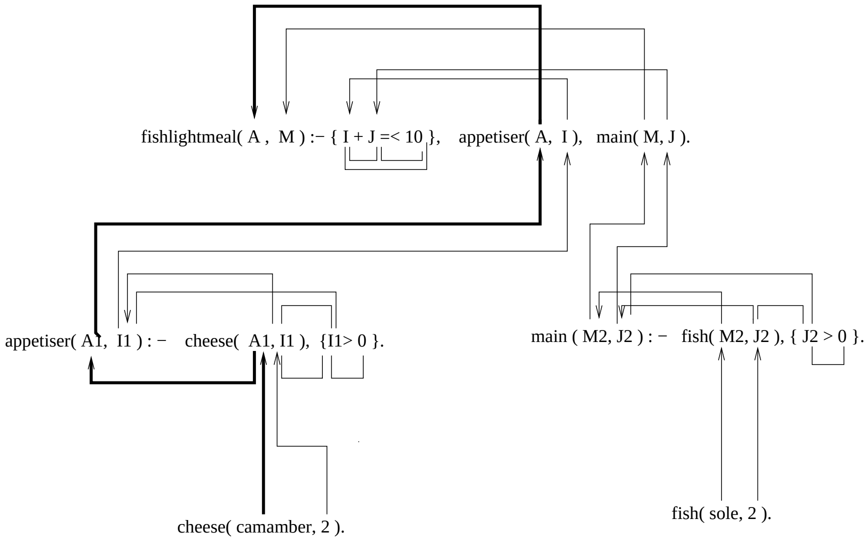

Figure 2. Proof tree dependence represented in graphical form along with the backward slice with respect to A in fishlightmeal(A,M)

The algorithmic program debugging method, introduced by Shapiro [19], 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. Shapiro's algorithm [19] traverses the proof tree of a program in different ways and asks the user about the expected behavior of each resolved goal. A major drawback of this debugging method is the great number of queries made to the user about the correctness of the intermediate result of procedure calls. A major improvement in the localization process is possible by combining the algorithmic debugging with slicing technique [21, 20]. In this paper we refer to this method as the **DEB_SLICE debugging method**, which consists of the following steps:

1. A proof tree is produced for a buggy program (negative example),
2. then a proof tree dependence graph is constructed which is sliced and
3. then those parts of the tree that have no influence on the visible symptom of a bug are removed.
4. The algorithmic debugger traverses the sliced proof tree only, thus concentrating on the suspect part of the program.

4.3. The CLP_SPEC_SLICE algorithm

CLP_SPEC_SLICE algorithm (defined in this paper) 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 consists of the following three steps:

1. finding the clause, the unfolding is applied upon (this step is done by the DEB_SLICE algorithm).
2. finding the literal within the clause, which will be the basis of the unfolding - the same method is applied here as that used in [3].
3. performing the unfolding on the program.

The Input of the algorithm is: An initial constraint logic program 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).

The Output of the algorithm is: A series of programs

$$P'^{(0)}, P'^{(1)}, \ldots, P'^{(n)} (P'^{(0)} = P),$$

where

$$P'^{(i+1)} = \widetilde{Unf}(P'^{(i)}) \qquad (0 \geq i \geq n), \text{ and } \widetilde{Unf}$$

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)
6. **if** the buggy clause can not be identified **then** stop
7. **if** R does not cover any atom in E^+ **then** remove R from $P'^{(i)}$
8. **else**
    begin
9. - unfold upon the literal b in R that is selected by the computation rule
      $P'^{(i+1)} := Unf(P'^{(i)}, R, b)$
10. - Let $D\_Res(P'^{(i)}, R, b) := Res(P'^{(i)}, R, b) \setminus$ {*those clauses that do not occur in refutations of positive examples* }

- $P'^{(i+1)} := P'^{(i+1)} \setminus D\_Res(P'^{(i)}, R, b)$ /*to find the most specific theory*/
end /*else*/
let $i := i + 1$
end /*while*/

We note that this algorithm also produces the most specific theory. It can readily bee seen that the difference between this and the CLP_SPEC algorithm is the choice of the clause whose literal is unfolded upon.

Theorem 2. 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 4). There are special cases however when the buggy clause could not be identified by the DEB_SLICE method. There is a solution to this problem [20], 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.

5. Prototype implementation

Both algorithms (CLP_SPEC and CLP_SPEC_SLICE) have been implemented in SICStus Prolog. For slicing we have used an earlier developed tool [21]. The algorithms have being tested on simple examples such as N-Queens, rectangle [1](to recognize a horizontally lying rectangle), horse-jumping, and so on. During the testing we employed the Prolog computation rule (i.e. we chosed the leftmost literal). From the test results it can be concluded that the number of clauses learned by CLP_SPEC_SLICE is less than that learned by the CLP_SPEC algorithm. It 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 percent of these user queries could be reduced applying slicing. Sometimes it was difficult to answer the user queries, since they were about the correctness of numerical functions (data). The list of the slice points, which can be identified by the help of a graphical user interface, could be given in a list inserted in the goal in the following way: In the

($fishlightmeal(A, beef, < 2 >)$ 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 about only 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.

6. Related work and discussion

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 gave 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. Different learning method have been introduced for learning constraint logic programs in [8, 11, 12, 14, 18]. *Kawamura and Furukawa* [8] adopted the dominant paradigm in ILP, namely the paradigm of inverse resolution for generalizing constraints. *Sebag et all.* [18] 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 [11] 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* [14] extend the concepts involved in the generalization of atoms, to more general forms of atoms, especially atoms with constraints attached to them. *Mizoguchi and Ohwada* [12], extend ideas from ILP based on Plotkin's framework [17] of Relative Least General Generalization(RLGG) to induce constraint logic programs. We have adopted an other existing ILP technique for ICLP, namely a specialization method using [3]. One of the aims of our future work will be to combine the CLP_SPEC_SLICE method with other specialization algorithms [4], [5], [6], [10], [15], [16].

7. Appendix

The following small example shows how the CLP_SPEC_SLICE algorithm learned the 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 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.

References

[1] **Alexin Z., Gyimóthy T. and Boström H.,** Integrating algorithmic debugging and unfolding transformation in an interactive learner, *Proceedings of ECAI'96, 12th European Conference on Artificial Intelligence, Budapest, Hungary, (Ed. W. Wahlster),* John Wiley and Sons Ltd., 1996., 403-408.

[2] **Boström H., Asker L.,** 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 Verlag, 1999.

[3] **Boström, H., 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*, 31-47.

[4] **De Schreye D., Gluck, R., Jorgensen, J., Leuschel, M., Martens, B. and Sorensen, H. M.,** Conjunctive partial deduction: Foundations, control, algorithms, and experiments. *The Journal of Logic Programming*, **41**(2-3)(1999), 231-277,
Erratum appeared in *JLP*, **43**(3)(2000), 265.

[5] **Fioravanti, F., Pettorossi, A. and Proietti, M.** Automated strategies for specialising constraint logic programs. **Kung-Kiu Lau** (ed.), *10th International Workshop on Logic-based Program Synthesis and Transformation,* LNCS **2042**, Springer-Verlag, 2000., 125146.

[6] **Howe, J.M., King, A.,** Specialising finite domain programs using polyhedra. **A. Bossi** ed.,*Logic-Based Program Synthesis and Transformation (LOPSTR' 99)*, LNCS **1817**, Springer-Verlag, 2000., 118-135.

[7] **Jaffar, J., Maher, M.J.,** Constraint logic programming: A Survey. *The Journal of Logic Programming*, **19/20**(1994), 503-582.
[8] **Kawamura T., Furakawa K.,** Towards inductive generalization in constraint logic programs. *Proceedings of the IJCAI-93 workshop on inductive logic programming, France,* Academic Press, 1993., 93-104.
[9] **Marriott, K., Stuckey, P. J.,** *Programming with constraints. An introduction.* The MIT Press, 1998.
[10] *Marriott M., Stuckey, P.,* The 3 R's of optimizing constraint logic programs: Refinement, removal and reordering. *Proceedings of the Twentieth Symposium on Principles of Programming Languages, Charleston, South Carolina*, ACM Press, 1993, 334-344.
[11] **Martin L., Vrain C.,** Induction of constraint logic programs. *Proceedings of Algorithms and Learning Theory (ALT), 1996, Sydney, Australia*, Springer Verlag, 1996.
[12] **Mizoguchi F., Ohwada H.,** *Constrained relative least general generalization for inducing constraint logic programs. New Generation Computing*, 1995.
[13] **Muggleton S.,** *Inductive logic programming.* The ACM Press, 1994.
[14] **Page C. D., Frisch A. M.,** Generalizing atoms in constraint logic, *Proceedings of Second International Conference on Knowledge Representation and Reasoning*, 1991.
[15] **Peralta, J. C., Gallagher, J. P.,** Imperative program specialisation: An approach using CLP. **Annalisa Bossi** (ed.), *Logic-Based Program Synthesis and Transformation*, LNCS **1817**, Springer Verlag, 1999., 102-117.
[16] **Peralta, J. C., Gallagher, J.P.,** Convex Hull Abstractions in Specialization of CLP Programs. *Proceedings of the 12th International Workshop on Logic Based Program Synthesis and Transformation (LOPSTR 2002), Madrid, Spain*, Springer Verlag, 2002., 90-108.
[17] **Plotkin, G. D.** A note on inductive generalization. *Machine Intelligence*, 1971., 101-124.
[18] **Sebag M., Rouveirol C.,** Constraint inductive logic programming. *Advances in ILP*, IOS Press, 1996., 277-294.
[19] **Shapiro, E.,** *Algorithmic Debugging*, The MIT Press.
[20] **Szilágyi, Gy., Harmath, L. and Gyimóthy, T.,** Debug Slicing of Logic Programs. *Acta Cybernetica*, **15**(2)(2001), 257-278.
[21] **Szilágyi, Gy., Maluszyński, J. and Gyimóthy, T.,** Static and Dynamic Slicing of Constraint Logic Programs. *Journal of Automated Software Engineering*, **9**(1)(2002), 41-65.
[22] **Tip, F.,** A survey of Program Slicing Techniques, *Journal of Programming Languauges*, **3**(3)(1995), 121-189.

Gy. Szilágyi Kocsisné
Department of Programming Languages and Compilers,
Eötvös Loránd University
Pázmány Péter s. 1/C
H-1117 Budapest, Hungary
`szilagyi@aszt.inf.elte.hu`