The Specialization Problem and the Completeness of Unfolding

Shan-Hwei Nienhuys-Cheng cheng@cs.few.eur.nl
Ronald de Wolf bidewolf@cs.few.eur.nl
Department of Computer Science, H4-19
Erasmus University of Rotterdam
P.O. Box 1738, 3000 DR Rotterdam, the Netherlands

January 11, 1996

Abstract

We discuss the problem of specializing a definite program with respect to sets of positive and negative examples, following [BI94]. This problem is very relevant in the field of inductive learning. First we show that there exist sets of examples that have no correct program, i.e., no program which implies all positive and no negative examples. Hence it only makes sense to talk about specialization problems for which a solution (a correct program) exists.

To solve such problems, we first introduce UD_1-specialization, based upon the transformation rule *unfolding*. We show UD_1-specialization is incomplete—some solvable specialization problems do not have a UD_1-specialization as solution—and generalize it to the stronger UD_2-specialization. UD_2 also turns out to be incomplete. An analysis of program specialization, using the subsumption theorem for SLD-resolution, shows the reason for this incompleteness. Based on that analysis, we then define UDS-specialization (a generalization of UD_2-specialization), and prove that any specialization problem has a UDS-specialization as a solution. We also discuss the relationship between this specialization technique, and the generalization technique based on inverse resolution. Finally, we go into several more implementational matters, which outline an interesting topic for future research.

Keywords: logic, resolution, SLD-resolution, Horn clauses, unfolding, program specialization, logic programming and inductive logic programming.

1 Introduction

This article discusses the *specialization problem*, which concerns the specialization of a definite program with respect to sets of positive and negative *examples*. These examples are usually expressed as ground atoms. Suppose E^+ is a set of positive examples, and E^- a set of negative examples. These sets may be infinite. One of the problems researchers in Inductive Logic Programming (ILP)

are interested in, is to find a definite program which is *correct* with respect to the examples. That is, we want to find a program that implies all members of E^+, and none of the members of E^-. Our hope is that such a program will have captured some of the structure and relationships among the examples. If so, the program can be used for predicting truth-values of ground atoms outside the set of examples. Thus finding a correct program can be seen as a form of inductive learning.

Often, it is the case that we initially have a program T that is sufficiently strong with respect to the examples (i.e., $T \models E^+$ and possibly $T \models A$ for some $A \in E^-$). The problem is then to construct a new program T' from T, such that T' is correct w.r.t. the examples. T' will be called a *specialization* of T. Finding a correct specialization corresponds to "shrinking" the least Herbrand model of the program, such that it no longer contains any of the negative examples.

A natural way to specialize T is, first, to replace a clause in T by all its resolvents upon some body-atom in this clause. Constructing these resolvents is called *unfolding*. Currently, there seems to be an increasing interest in unfolding as a specialization method, see for instance [BI94, TS84, Bos95b, Bos95a, AGB95]. The new program obtained after unfolding a clause in T, is clearly implied by T. The function of the replaced clause is taken over by the set of resolvents produced by unfolding. We can then, secondly, delete some new clauses from the program that have to do with the negative examples, thus specializing the program. These two steps probably have to be repeated many times to get rid of all negative examples. This method derives from [BI94], and is the basis of the first specialization technique—UD_1-specialization—that we will define here.

For simplicity, let all examples be ground instances of the atom $P(x_1, \ldots, x_n)$, for some predicate P. The motivation for the method described above, is the fact that it can be used to prune negative examples from the SLD-tree for $T \cup \{\leftarrow P(x_1, \ldots, x_n)\}$.[1] We will illustrate this by an example. Consider the program T, consisting of the following clauses:

$$C_1 = P(x, y) \leftarrow Q(x, y)$$
$$C_2 = Q(b, b) \leftarrow Q(a, a)$$
$$C_3 = Q(a, a)$$

and $E^+ = \{P(b, b)\}$, $E^- = \{P(a, a)\}$. The SLD-tree for $T \cup \{\leftarrow P(x, y)\}$ is shown on the left of figure 1. The labels on the branches correspond to the input clauses used, the computed answer of each success branch is shown below that branch. For instance, in the rightmost branch, first C_1 and then C_3 are used as input clause, which leads to the computed answer $\{x/a, y/a\}$, meaning that $T \models P(a, a)$. We have indicated the computed answer corresponding to the positive example with a '+', for the negative example with a '$\Leftrightarrow$'.

$P(a, a)$ is a negative example, so we would like to remove this by weakening the program. This could be done by deleting C_1 or C_3 from T. However, this

[1] In this paper, we use the letter T to denote definite programs (rather than the usual P), to avoid confusion between the predicate P and a definite program. See [Llo87] for the definition of an SLD-tree.

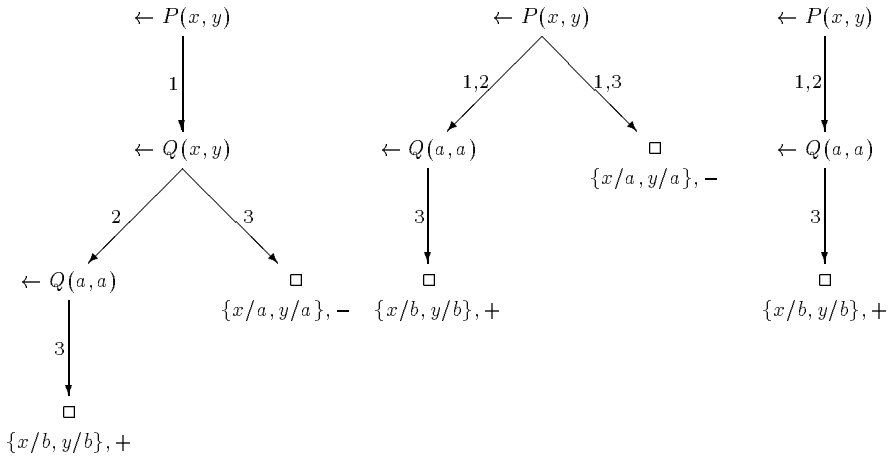


Figure 1: From left to right: the SLD-trees for T, T', and T''

would also make the positive example $P(b,b)$ no longer derivable. Another way to specialize is, first, to unfold C_1 upon $Q(x,y)$. The following $C_{1,2}$ and $C_{1,3}$ are the two clauses produced by unfolding C_1.

$C_{1,2} = P(b,b) \leftarrow Q(a,a)$ (resolvent of C_1 and C_2)
$C_{1,3} = P(a,a)$ (resolvent of C_1 and C_3)

Now we replace the unfolded clause C_1 by its resolvents $C_{1,2}$ and $C_{1,3}$. This results in $T' = \{C_2, C_3, C_{1,2}, C_{1,3}\}$. The SLD-tree for $T' \cup \{\leftarrow P(x,y)\}$ is shown in the middle of figure 1. In this tree, the negative example is directly connected to the root, via the branch that uses $C_{1,3}$. Now the negative example can be pruned from the tree by deleting $C_{1,3}$ from T', which does not affect the positive example. Then we obtain $T'' = \{C_2, C_3, C_{1,2}\}$, which is correct w.r.t. E^+ and E^-. See the right of figure 1 for the SLD-tree for $T'' \cup \{\leftarrow P(x,y)\}$. The idea behind this method is the following:

1. Unfolding removes some internal nodes from the SLD-tree, for instance, the internal node $\leftarrow Q(x,y)$ in the tree on the left of figure 1. This tends to seperate the positive from the negative examples, and also brings them closer to the root of the tree.
2. If a negative example hangs directly from the root, and its input clause C is not used elsewhere in the tree for a positive examples, then the program can be specialized by deleting C.

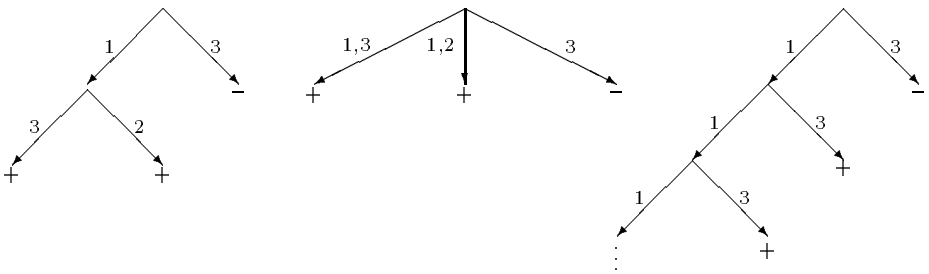
In [BI94], the algorithm SPECTRE is presented, which implements this specialization technique, using an information-based heuristic to guide the search. They also present some experimental results that are very encouraging. SPECTRE generally outperforms an alternative algorithm, which is based on *covering* the examples rather than on seperating the positive from the negative examples. In their experiments, the examples are taken from real-world domains such as data on when to allow the Space Shuttle to land automatically, and Congressional voting records. The programs produced by SPECTRE give a higher accuracy on the test-set, and contain much less clauses than the programs produced by the covering algorithm. Similar and equally encouraging experiments are described

in [Bos95b]. In [Bos95a], the algorithm SPECTRE II is presented, which overcomes some difficulties of SPECTRE concerning recursive clauses, and which can be applied to multiple-predicate learning (i.e., then the members of E^+ and E^- need no longer be ground instances of the same atom $P(x_1, \ldots, x_n)$). [Bos95a] describes a special case in which SPECTRE II is complete, and again contains some experimental results.

However, not all specialization problems are as simple as the example we have presented above. Some essential questions are left unanswered in [BI94]. Let T' be the program resulting from unfolding some clause in T. Then the following points arise:

- In [TS84], it is shown that unfolding does not affect the least Herbrand model of the program: $M_T = M_{T'}$. As [Bos95a] states, unfolding is "meaning preserving". However, despite the fact that unfolding leaves the least Herbrand model invariant, it may still weaken the program in the sense that the new program may not be logically equivalent to the old program. That is, sometimes $T \models T'$, but $T' \not\models T$. As we will explain later on, this is one of the reasons for the incompleteness of the approach towards specialization in [BI94].

- What if after some applications of unfolding, we have a negative example hanging directly from the root of the tree, via input clause C, but C is also used elsewhere for the refutation of a positive example? Deleting C may then specialize the program too much. For instance, on the left of the following figure we cannot delete clause 3. Perhaps some further unfolding



may solve this. For instance, if we unfold clause 1 upon some body atom, we get the tree on the right (here 1,2 denotes a resolvent of clauses 1 and 2, and 1,3 denotes a resolvent of clauses 1 and 3). In this tree, we can safely delete clause 3—it is no longer needed for the refutation of a positive example. This approach may get into trouble in case of infinite E^+ and E^-, or in case there is an infinite number of SLD-refutations of positive or negative examples. The completeness-result proved in [Bos95a] only concerns finite E^+ and E^-, and presupposes that the number of SLD-refutations of positive and negative examples is finite.

Nevertheless, an infinite number of examples might be given. For example, when we want to be sure that our system learns the predicate *Odd* correctly, we could use $\{Odd(s(0)), Odd(s^3(0)), Odd(s^5(0)), \ldots\} \subseteq E^+$ and $\{Odd(0), Odd(s^2(0)), Odd(s^4(0)), \ldots\} \subseteq E^-$. That is, we could include a

complete description of the predicate in the sets of examples. Similarly, an infinite number of refutations of examples may occur in practice, for instance in the tree shown on the right of the previous picture. Will a specialization technique based on unfolding work in this case?

To try to solve these problems, we make our own definition of unfolding, and use this for a second specialization technique, UD_2-specialization, which generalizes UD_1-specialization. However, UD_2-specialization is still not complete—it cannot solve all specialization problems.

For completeness, we need to add the possibility of taking a subsumption step. This gives UDS-specialization (short for **U**nfolding, clause **D**eletion, **S**ubsumption). To analyse and prove the completeness of UDS-specialization, we use the subsumption theorem for unconstrained resolution and for SLD-resolution (see [NW95a, NW95c]). The subsumption theorem allows us to look at unfolding from a higher level, without considering the messy details of SLD-trees. Unfolding turns out to be intimately related to the subsumption theorem. We have divided the article in the following parts:

1. First we consider the specialization problem as stated in [BI94]. We show that there exist sets of examples that have no correct program. Hence it only makes sense to talk about specialization problems for which a solution (a correct program) exists. This leads to a reformulation of the specialization problem in Section 4.
2. Then we formally introduce unfolding in Section 5.
3. In Section 6, we define UD_1-specialization, which is the specialization technique used in [BI94], and show that it is not complete.
4. In Section 7, we define the stronger UD_2-specialization, which also turns out to be incomplete.
5. Then in Section 8, we relate unfolding to the subsumption theorem for resolution. This induces UDS-specialization (a generalization of UD_2-specialization), which we prove to be complete
6. In Section 9, we discuss the relation between program specialization by unfolding, and program generalization by inverse resolution. These two approaches turn out to be more or less duals.
7. This paper discusses specialization and unfolding from a fairly high level. However, in the final section we will devote some attention to more implementational matters, by connecting the previous work with certain properties of SLD-trees. The relations between specialization, unfolding, the subsumption theorem and SLD-trees are very interesting, and suggest some directions for future research.

2 SLD-resolution

In this section, we will briefly describe the definitions we use regarding SLD-resolution. Here a *Horn clause* is either a definite program clause (a clause with one positive, and zero or more negative literals), or a definite goal (a clause containing only negative literals). Our definitions mainly coincide with those of [Llo87], but we use SLD-resolution not only to construct refutations,

but also to derive definite program clauses or definite goals from a set of Horn clauses. Here we follow an idea of [MP94]. Thus our definition of an SLD-derivation generalizes that of [Llo87]. However, with respect to refutations, our definitions are equivalent to those of [Llo87], which allows us to use several results proved in [Llo87].

Definition 1 Let $C = L \leftarrow L_1, \ldots, L_i, \ldots, L_n$ and $D = M \leftarrow M_1, \ldots, M_m$ be two definite program clauses which are standardized apart (i.e., which have no variables in common). Let θ be an mgu (most general unifier) of L_i and M. Then the clause $(L \leftarrow L_1, \ldots, L_{i-1}, M_1, \ldots, M_m, L_{i+1}, \ldots, L_n)\theta$ is called a *binary resolvent* of C and D. C and D are called the *parent clauses* of the binary resolvent, L_i and M are said to be the *literals resolved upon*. $\diamond$

We can similarly define a binary resolvent of C and D in case C is a definite goal (i.e., in case the head L of C is omitted). We might want to obtain a binary resolvent of clauses C and D which are not standardized apart. In this case, we rename D to D', such that C and D' are standardized apart, and then obtain a binary resolvent of C and D'. For simplicity, this is then also called a binary resolvent of C and D itself.

Definition 2 Let Σ be a set of Horn clauses, and C be a Horn-clause. An *SLD-derivation* of *length* n of C from Σ is a finite sequence of Horn clauses $H_0, H_1, \ldots, H_n = C$, such that $H_0 \in \Sigma$ and each H_i with $1 \leq i \leq n$ is a binary resolvent of H_{i-1} and a definite program clause $C_i \in \Sigma$, using the head of C_i and a *selected atom* in the body of H_{i-1} as the literals resolved upon.

H_0 is called the *top clause*, and the C_i are called the *input clauses* of this SLD-derivation. If an SLD-derivation of C from Σ exists, we write $\Sigma \vdash_{rs} C$. An SLD-derivation of $\Box$ from Σ is called an *SLD-refutation* of Σ. $\diamond$

We allow the first clause H_0 to be either a definite goal (as in [Llo87]), or a definite program clause. In this way, SLD-resolution can be used to derive not only definite goals, but also definite clauses from the program. Note that either each H_i is a definite goal, or each H_i is a definite program clause.

Definition 3 Let C and D be Horn clauses. We say D *subsumes* (or θ-*subsumes*) C, if there exists a substitution θ such that $D\theta \subseteq C$. $\diamond$

Definition 4 Let Σ be a set of Horn clauses and C a Horn clause. There exists an *SLD-deduction* of C from Σ, written as $\Sigma \vdash_{ds} C$, if C is a tautology, or if there exists a Horn clause D, such that $\Sigma \vdash_{rs} D$ and D subsumes C. $\diamond$

For example, suppose $\Sigma = \{(P(x,y) \leftarrow Q(y,x)), (Q(a,z) \leftarrow R(z)), R(f(y))\}$ and $C = P(f(b),a) \leftarrow R(x)$. Figure 2 shows an SLD-deduction of C from Σ. The sequence H_0, H_1, H_2 forms an SLD-derivation of H_2 from Σ (with input clauses C_1 and C_2), so $\Sigma \vdash_{rs} H_2$. H_2 subsumes C, so $\Sigma \vdash_{ds} C$.

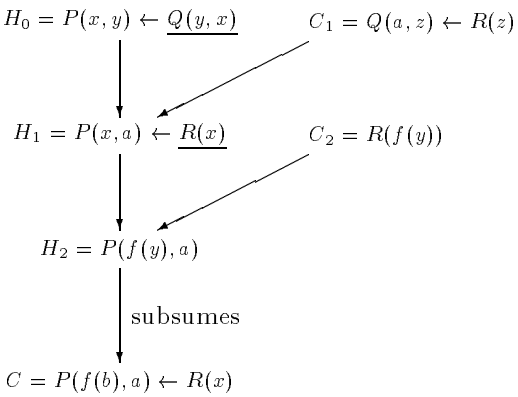


Figure 2: An SLD-deduction of C from Σ

3 Some specialization problems are unsolvable

The following is the statement of the specialization problem in terms of least Herbrand models, as given in [BI94].

> Given: a definite program T and two disjoint sets of ground atoms E^+ and E^- (positive and negative examples).
> Find: a definite program T', called a *specialization* of T w.r.t. E^+ and E^-, such that $M_{T'} \subseteq M_T$, $E^+ \subseteq M_{T'}$ and $M_{T'} \cap E^- = \emptyset$.

Here M_T denotes the least Herbrand model of T. Note that any background knowledge can be included in T, so we do not need to discuss the use of background knowledge separately. It is assumed that E^+ and E^- contain only *ground* instances of atoms. These examples may be instances of different predicates (we could for example have $E^+ = \{P(a), Q(a,b), R(f(b)), \ldots\}$), so our approach is well-suited for multiple-predicate learning. It is also assumed that T is sufficiently strong w.r.t. the examples. That is, $E^+ \subseteq M_T$ and possibly $M_T \cap E^- \neq \emptyset$. Notice that for ground atoms, $A \in M_T$ iff $T \models A$. First we introduce some terminology:

Definition 5 Let T be a definite program, and E^+ and E^- sets of ground atoms. We say T is *sufficiently strong w.r.t. E^+ and E^-* if $T \models A$ for all $A \in E^+$. We say T is *correct w.r.t. E^+ and E^-* if $T \models A$ for all $A \in E^+$, and $T \not\models A$ for all $A \in E^-$. $\diamond$

Note that a correct program is a special case of a sufficiently strong program. The specialization problem starts with a sufficiently strong T, and aims at finding a specialization T' that is correct w.r.t. E^+ and E^-.

A first problem that is raised by this approach, is the following: *does such a program T' always exist?* In other words, suppose we have arbitrary, possibly infinite, sets E^+ and E^- of ground instances of $P(x_1, \ldots, x_n)$—does there always exist a T' that is correct w.r.t. E^+ and E^-? Usually, finding a program for sets of examples is handled in an incremental way, without considering first whether such a program actually exists. However, it is obvious that this question is crucial. If we have a specialization problem that starts with E^+, E^- for

which no correct specialization exists, then we have an *unsolvable* specialization problem.

In this section we will prove that indeed there exist unsolvable specialization problems. We will show that there are sets of examples for which a correct definite program does not exist.

Definition 6 Let L be a clausal language containing an n-ary predicate symbol P, and I be an Herbrand interpretation of L. Then B_L, the *Herbrand base of* L, denotes the set of ground atoms in L, $B_{L,P}$ denotes the set of ground instances of $P(x_1, \ldots, x_n)$ in L, and I_P denotes the set of ground instances of $P(x_1, \ldots, x_n)$ that are true under I (so $I_P = B_{L,P} \cap I$). $\diamond$

Consider a language L containing (possibly among others) an n-ary predicate P, an m-ary function f $(n, m \geq 1)$, and a constant a. The Herbrand base B_L—the set of all ground atoms in L—is countably infinite. Every subset of B_L represents an Herbrand interpretation of L. Since the set of all subsets of a countably infinite set is uncountable, the set of all Herbrand interpretations is uncountable.

On the other hand, the set of definite clauses in L is countably infinite, so the set of all definite programs (i.e., the set of all *finite* sets of definite clauses) is also only countably infinite. A program has only one least Herbrand model, so the set of all programs can only induce a countably infinite number of least Herbrand models. An uncountable set is much larger than a countably infinite sets, hence there is still an uncountable number of Herbrand interpretations I that have no program T with $M_T = I$.

Now consider $\mathcal{I}_P = \{I_P | I \subseteq B_L\}$. By the same argument as before, this is also an uncountable set. Therefore there exists an Herbrand interpretation I for which there is no program T with $M_{T,P} = I_P$. Thus we have the following result:

Theorem 1 *Let L be a clausal language containing (among others) an n-ary predicate symbol P, an m-ary function symbol f $(n, m \geq 1)$, and a constant a. Then there exists an Herbrand interpretation I of L, such that there is no definite program $T \subseteq L$ with $M_{T,P} = I_P$*

From the previous theorem we can infer that any technique that tries to induce a definite program from arbitrary, possibly infinite[2] sets E^+ and E^- of positive and negative examples is necessarily incomplete, even when only a single predicate P is to be learned. Suppose I is one of those Herbrand interpretations for which there is no T with $M_{T,P} = I_P$. Let $E^+ = I_P$ and $E^- = B_{L,P} \backslash I_P$. Then E^+ contains all true ground instances of $P(x_1, \ldots, x_n)$, E^- all false ground instances. There is no program T that is correct w.r.t. these E^+ and E^-, for otherwise we would have $M_{T,P} = I_P$, which is impossible. Thus we have the following corollary:

Corollary 1 *Let L be a clausal language containing (among others) an n-ary predicate symbol P, an m-ary function symbol f $(n, m \geq 1)$, and a constant a.*

[2] If E^+ is finite, setting $T' = E^+$ trivially solves any specialization problem.

Then there exist sets E^+ and E^- of ground instances of $P(x_1, \ldots, x_n)$, such that there is no definite program that is correct w.r.t. E^+ and E^-.

In short: some specialization problems are unsolvable, because the specialization T' that we want to find simply does not exist. This shows that any approach towards program-specialization is incomplete if we allow arbitrary infinite sets E^+ and E^-. Hence we should restrict attention to methods that can find a correct specialization *if indeed one exists.*

4 A restatement of the specialization problem

For a solution T' of the earlier formulation of the specialization problem, we mentioned two conditions: $M_{T'} \subseteq M_T$, and T' should be correct w.r.t. E^+ and E^-. From the previous section, it is clear that we should add the existence of such a correct T' as a precondition to the specialization problem.

However, there is a second problem with respect to that formulation of the specialization problem, namely the condition that $M_{T'} \subseteq M_T$. This condition may hold for T' that we would not want to call a specialization of T. For example, suppose we have $T = \{P(a), (P(x) \leftarrow P(f(x)))\}$ and $T' = \{P(x) \leftarrow P(g(x))\}$. Then $M_{T'} = \emptyset \subset \{P(a)\} = M_T$. However, T and T' have not very much to do with each other, since neither $T \models T'$ nor $T' \models T$. It is not very likely that a specialization technique can always construct T' from T as above. Thus the condition that $M_{T'} \subseteq M_T$ is not a very good basis for a specialization method.

Usually, specialization methods—such as Shapiro's refinement operators [Sha81], and also the methods based on unfolding that we will define in the next sections—construct programs T' from T for which $T \models T'$ holds. But inconsistently, many researchers in this field at the same time use $M_{T'} \subseteq M_T$ to formulate their specialization problem. Hence we will replace the condition $M_{T'} \subseteq M_T$ by the stronger, and more natural, requirement $T \models T'$. These arguments suggest a restatement of the specialization problem, as follows:

> Given: two disjoint sets E^+ and E^- of ground instances of the atom $P(x_1, \ldots, x_n)$, a definite program T that is sufficiently strong w.r.t. E^+ and E^-, and there exists a program T' such that $T \models T'$ and T' is correct w.r.t. E^+ and E^-.
> Find: one such a T'.

5 Unfolding

In this section, we introduce the definition of unfolding given in [BI94, TS84]. We use 'u1' in this definition to indicate that this is the first type of program that may result from unfolding. We will define a second type in one of the next sections.

Definition 7 Let T be a definite program, $C = A \leftarrow B_1, \ldots, B_n$ a definite program clause in T, and B_i the i-th atom in the body of C. Let $\{C_1, \ldots, C_m\}$ be the set of clauses in T whose head can be unified with B_i.

Then *unfolding C upon B_i in T* means constructing the set $U_{C,i} = \{D_1, \ldots, D_m\}$, where each D_j is the resolvent of C_j and C, using B_i and the head of C_j as the literals resolved upon. We say the *type 1 program resulting from unfolding C upon B_i in T* is the program $T_{u1,C,i} = \{T \backslash \{C\}\} \cup U_{C,i}$. $\diamond$

Thus the type 1 program is obtained by replacing C by all resolvents upon B_i of C and clauses in T. Now we give an example of unfolding, and the resulting type 1 program. Let the program T consist of the following clauses:

$C_1 = P(f(x)) \leftarrow P(x), Q(x)$
$C_2 = Q(x) \leftarrow R(x, a)$
$C_3 = P(f(a))$
$C_4 = Q(b)$

Suppose we want to unfold C_1 upon $Q(x)$ in the program T. Then we have:

- $\{C_2, C_4\}$ is the set of clauses whose head can be unified with $Q(x)$.
- $U_{C_1,2} = \{(P(f(x)) \leftarrow P(x), R(x, a)), (P(f(b)) \leftarrow P(b))\}$
- $T_{u1,C_1,2} = \{C_2, C_3, C_4\} \cup U_{C_1,2}$

Note that $U_{C,i}$ may be the empty set (this is the case if there is no program clause whose head unifies with the i-th atom in the body of C). In this case, $T_{u1,C,i} = T \backslash \{C\}$. Note also that a unit clause cannot be unfolded, since it has no body-atoms.

The authors of [BI94] call unfolding a "transformation rule", but do not mention any properties of the relation between T and $T_{u1,C,i}$. Here we will investigate this relation. A first result concerning unfolding is the following proposition, which shows that an SLD-refutation using T can be turned into an SLD-refutation using $T_{u1,C,i}$, and vice versa. For the rather technical details of the proof of this proposition, we refer to Appendix A.

Here we only give the idea behind the proof, which shows how a refutation-tree using T can be transformed into a refutation-tree using $T_{u1,C,i}$. If there is a refutation of $T \cup \{G\}$ which doesn't use C, then this is also a refutation of $T_{u1,C,i} \cup \{G\}$. If, on the other hand, C is used somewhere as input clause in a refutation-tree for T, we can replace the usage of this C by using a clause in $U_{C,i}$ instead.

Consider figure 3, the left of which shows part of the refutation-tree for T. Suppose $C_1 = A \leftarrow B_1, \ldots, B_n$ is the unfolded clause, and C_1 is used as input clause in an SLD-refutation of $T \cup \{G\}$, in the step leading from the $(j \Leftrightarrow 1)$-th goal G_{j-1} to the j-th goal G_j. The selected atom is A_k and θ is the mgu of A_k and the head of C_1. We can assume due to the independence of the computation rule (Theorem 9.4 of [Llo87]) that $B_i\theta$ is the selected atom in the next step. Suppose C_2 is the input clause in this next step. Now the two steps using C_1 and C_2 can be replaced by a single step using $C_{1,2}$ (the resolvent upon B_i of C_1 and C_2), which is a member of $U_{C_1,i}$. Furthermore, we can show that G_{j+1} and G'_{j+1} are variants. Thus we can transform an SLD-refutation of $T \cup \{G\}$ into an SLD-refutation of $T_{u1,C_1,i} \cup \{G\}$, each time replacing two derivation steps by a single derivation step to eliminate all usages of C_1 as input clause.

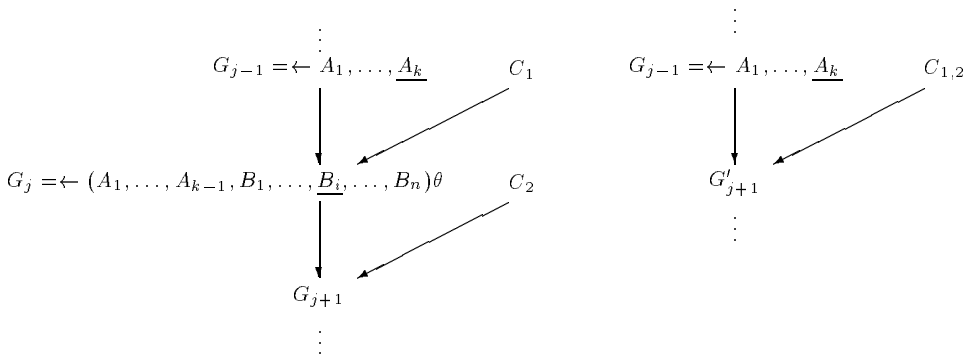


Figure 3: After unfolding, two steps can be replaced by a single step

As illustration, compare figure 3 with the trees on the left and in the middle of figure 1. The leftmost branch of the left tree in figure 1 shows an SLD-refutation of length 3 with computed answer $\{x/b, y/b\}$, which uses C_1 once. In the tree in the middle, for the program T' which is the type 1 program resulting from unfolding C_1, we see that the leftmost branch has the same computed answer, but has length $3 \Leftrightarrow 1 = 2$. The two steps using C_1 and C_2 (left tree of figure 1) are replaced by a single step (middle tree) using $C_{1,2}$ as input clause instead of C_1 and C_2.

Proposition 1 *Let T be a definite program, G a definite goal, and $T_{u1,C,i}$ the type 1 program resulting from unfolding C upon B_i in T. Then $T \cup \{G\} \vdash_{rs} \square$ iff $T_{u1,C,i} \cup \{G\} \vdash_{rs} \square$.*

A direct consequence of our proof of the previous proposition in Appendix A is the following:

Corollary 2 *Let T be a definite program, G a definite goal, and $T_{u1,C,i}$ the type 1 program resulting from unfolding C upon B_i in T. Suppose there exists an SLD-refutation of length n of $T \cup \{G\}$, which uses C r times as input clause. Then there exists an SLD-refutation of length $n \Leftrightarrow r$ of $T_{u1,C,i} \cup \{G\}$.*

Intuitively, this corollary shows that unfolding makes refutations shorter. So unfolding has the potential of improving the efficiency of an SLD-based theorem prover. Especially unfolding often-used clauses is worthwhile, because then the value r mentioned in the corollary is highest. On the other hand, unfolding usually increases the number of clauses. So what we see here is an interesting trade-off between the number of clauses and the average length of a refutation: unfolding usually decreases the average length of a refutation, but also usually increases the number of clauses in the program.

We now proceed to prove that unfolding preserves the least Herbrand model M_T of the program. This is also proved in [TS84], though differently from our proof (they do not consider SLD-resolution, and hence do not have our Corollary 2). In a way, this shows that unfolding does not make the program "weaker" or "stronger".

Theorem 2 *Let T be a definite program, $C \in T$, and $T_{u1,C,i}$ the type 1 program resulting from unfolding C upon B_i in T. Then $M_T = M_{T_{u1,C,i}}$.*

Proof Let A be some ground atom. Then:
$A \in M_T$ iff (by Theorem 6.2 of [Llo87])
$T \models A$ iff (by Proposition 3.1 of [Llo87])
$T \cup \{\leftarrow A\}$ is unsatisfiable iff (by Corollary 7.2 and Theorem 8.4 of [Llo87])
$T \cup \{\leftarrow A\} \vdash_{rs} \square$ iff (by our Proposition 1)
$T_{u1,C,i} \cup \{\leftarrow A\} \vdash_{rs} \square$ iff (by Corollary 7.2 and Theorem 8.4 of [Llo87])
$T_{u1,C,i} \cup \{\leftarrow A\}$ is unsatisfiable iff (by Proposition 3.1 of [Llo87])
$T_{u1,C,i} \models A$ iff (by Theorem 6.2 of [Llo87])
$A \in M_{T_{u1,C,i}}$.
Hence $M_T = M_{T_{u1,C,i}}$. $\square$

6 UD_1-specialization

Unfolding together with clause deletion can be used to solve some specialization problems. In this section we define UD_1-specialization. We introduce this name as an acronym for **U**nfolding and clause **D**eletion. The '1' indicates that we use the type 1 program resulting from unfolding here. UD_1-specialization corresponds to the approach taken in [BI94].

Definition 8 Let T and T' be definite programs. We say T' is a *UD_1-specialization* of T, if there exists a sequence $T_1 = T, T_2, \ldots, T_n = T'$ $(n \geq 1)$ of definite programs, such that for each $j = 1, \ldots, n \Leftrightarrow 1$, $T_{j+1} = T_{ju1,C,i}$ or $T_{j+1} = T_j \backslash \{C\}$ for some $C \in T_j$. $\diamond$

If $T_{j+1} = T_{ju1,C,i}$, then each clause in T_{j+1} is either in T_j, or a resolvent of two clauses in T_j. Hence $T_j \models T_{j+1}$ in this case. If $T_{j+1} = T_j \backslash \{C\}$, then clearly $T_j \models T_{j+1}$. Thus we have the following:

Proposition 2 *Let T be a definite program, and T' a UD_1-specialization of T. Then $T \models T'$.*

For a solution T' of our restated version of the specialization problem, we mentioned two conditions: $T \models T'$, and T' should be correct w.r.t. E^+ and E^-. The previous proposition shows that a UD_1-specialization of T always satisfies the first condition.

However, the second condition cannot always be satisfied by UD_1-specialization. Two kinds of steps can be taken here: T_{j+1} can be the result of unfolding a clause in T_j, or by deleting a clause from T_j. The first kind of step preserves the least Herbrand model, the second kind possibly reduces it. In fact, not only deleting a clause, but also the unfolding-step may weaken the program. For instance, suppose $T = \{P(a), (P(x) \leftarrow P(f(x)))\}$. Then $T' = \{P(a), (P(x) \leftarrow P(f^2(x)))\}$ is the result of unfolding $P(x) \leftarrow P(f(x))$ in

T. Whereas this unfolding-step has not affected the least Herbrand model—$M_T = M_{T'} = \{P(a)\}$—it has indeed made the program weaker: $T \models T'$, but $T' \not\models T$.

In fact, even if a correct program T' is implied by the original program T, this T' need no longer be implied by a program T'' obtained from T by UD_1-specialization. Since further UD_1-specializations of T'' can only yield programs which are implied by T'' (and hence do not imply the solution T'), UD_1-specialization will not reach a solution of the specialization problem in this case. Consider $T = \{(P(f(x)) \leftarrow P(x)), P(a)\}$. Let $M_T = \{P(a), P(f(a)), P(f^2(a)), P(f^3(a)) \ldots\}$, and let $E^+ = M_T \backslash \{P(f^2(a))\}$ and $E^- = \{P(f^2(a))\}$. See figure 4.

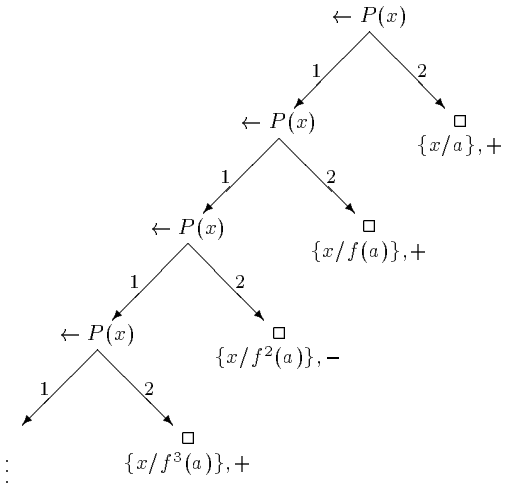


Figure 4: The SLD-tree of $T \cup \{\leftarrow P(x)\}$

Let $T_1 = T$. The only clause that can be unfolded is $P(f(x)) \leftarrow P(x)$. Unfolding this clause results in

$$T_2 = \{(P(f^2(x)) \leftarrow P(x)), P(f(a)), P(a)\}.$$

Then unfolding $P(f^2(x)) \leftarrow P(x)$ gives

$$T_3 = \{(P(f^4(x)) \leftarrow P(x)), P(f^3(a)), P(f^2(a)), P(f(a)), P(a)\}.$$

Notice that $M_{T_1} = M_{T_2} = M_{T_3}$, but unfolding has nevertheless weakened the program: $T_1 \models T_2 \models T_3$, but $T_2 \not\models T_1$ and $T_3 \not\models T_2$. In T_3, $P(f^4(x)) \leftarrow P(x)$ can be unfolded, etc. It is not difficult to see that in general, any UD_1-specialization of T is a subset of

$$\{P(f^{2^n}(x)) \leftarrow P(x)), P(f^{2^n-1}(a)), P(f^{2^n-2}(a)), \ldots, P(f^2(a)), P(f(a)), P(a)\},$$

for some n. To specialize this program such that $P(f^2(a))$ is no longer derivable, we must in any case remove $P(f^2(a))$. However, this would also prune some of the positive examples (such as $P(f^{2^n+2}(a))$) from the program via the clause $P(f^{2^n}(x)) \leftarrow P(x))$. Hence there is no UD_1-specialization that solves this particular specialization problem. Note that

$$T'' = \{(P(f^4(x)) \leftarrow P(x)), (P(f^3(x)) \leftarrow P(x)), P(f(a)), P(a)\}$$

is a solution for this particular specialization problem. $T \models T''$, but the specializations $T_2, T_3, \ldots$ no longer imply this correct program T''. So in this case, UD_1-specialization has "skipped" over the right solution. In the next section, we will show how this can be solved by UD_2-specialization.

7 UD_2-specialization

The previous example showed the incompleteness of UD_1-specialization. But suppose we change the definition of unfolding such that the unfolded clause is not removed immediately from the program. This increases the number of clauses that can later on be used in unfolding. In this case, we *can* find a correct specialization w.r.t. the examples given above in Section 6, as follows. We start with $T'_1 = T$, and unfold $P(f(x)) \leftarrow P(x)$ without removing the unfolded clause. This gives T'_2:

$$T'_2 = \{(P(f^2(x)) \leftarrow P(x)), (P(f(x)) \leftarrow P(x)), P(f(a)), P(a)\}.$$

Now we unfold $P(f^2(x)) \leftarrow P(x)$, again without removing the unfolded clause. This gives T'_3:

$$\begin{aligned} T'_3 = \quad & \{(P(f^4(x)) \leftarrow P(x)), (P(f^3(x)) \leftarrow P(x)), (P(f^2(x)) \leftarrow P(x)), \\ & (P(f(x)) \leftarrow P(x)), P(f^3(a)), P(f^2(a)), P(f(a)), P(a)\}. \end{aligned}$$

If we remove $(P(f^2(x)) \leftarrow P(x))$, $(P(f(x)) \leftarrow P(x))$, $P(f^3(a))$ and $P(f^2(a))$ from T'_3, we obtain T'':

$$T'' = \{(P(f^4(x)) \leftarrow P(x)), (P(f^3(x)) \leftarrow P(x)), P(f(a)), P(a)\}.$$

This is a correct specialization of T w.r.t. E^+ and E^-: $T'' \models E^+$, and $T'' \not\models P(f^2(a))$.

The previous example induces a second type of program resulting from unfolding, which differs from the first only in the fact that the unfolded clause C is not immediately removed from the program, so $T_{u2,C,i} = T \cup U_{C,i} = T_{u1,C,i} \cup \{C\}$. Unfolding itself does not change, but the programs *resulting* from unfolding in the first or second type differ, since in the latter case the unfolded clause is not deleted from the program.

Definition 9 Let T be a definite program, $C \in T$, and B_i the i-th atom in the body of C. Then we say the *type 2 program resulting from unfolding C upon B_i in T* is the program $T_{u2,C,i} = T_{u1,C,i} \cup \{C\}$. $\diamond$

Any clause in $T_{u2,C,i}$ is either in T, or a resolvent of two clauses in T. Hence $T \models T_{u2,C,i}$. Also $T \subseteq T_{u2,C,i}$, so $T_{u2,C,i} \models T$. So while type 1 only preserved the least Herbrand model, type 2 preserves equivalence, which is stronger:

Proposition 3 *Let T be a definite program, $C \in T$, and B_i the i-th atom in the body of C. Then $T \Leftrightarrow T_{u2,C,i}$.*

From the previous proposition and Theorem 2, we immediately have the following:

Corollary 3 $M_T = M_{T_{u1,C,i}} = M_{T_{u2,C,i}}$.

We now define the concept of UD_2-specialization. The only difference with UD_1-specialization is that we now use the type 2 program resulting from unfolding, instead of the type 1 program.[3] Thus of the two possible steps in UD_2-specialization—obtaining the type 2 program resulting from unfolding, and clause deletion—only clause deletion really weakens the program.

Definition 10 Let T and T' be definite programs. We say T' is an *UD_2-specialization* of T, if there exists a sequence $T_1 = T, T_2, \ldots, T_n = T'$ $(n \geq 1)$ of definite programs, such that for each $j = 1, \ldots, n \Leftrightarrow 1$, $T_{j+1} = T_{j_{u2,C,i}}$ or $T_{j+1} = T_j \backslash \{C\}$ for some $C \in T_j$. $\diamond$

Note that any UD_1-specialization is also a UD_2-specialization, since obtaining the type 2 program and then removing the unfolded clause in the next step, is equivalent to obtaining the type 1 program. The following proposition is obvious:

Proposition 4 *Let T be a definite program, and T' a UD_2-specialization of T. Then $T \models T'$.*

For a solution T' of our restatement of the specialization problem, we mentioned two conditions: $T \models T'$, and T' should be correct w.r.t. E^+ and E^-. The previous proposition shows that a UD_2-specialization of T always satisfies the first condition. Since any UD_1-specialization is a UD_2-specialization, while some UD_2-specializations cannot be found with UD_1-specialization (see the example above), UD_2-specialization is "more complete" than UD_1-specialization.

Unfortunately, UD_2-specialization is still not sufficiently strong to provide a solution for all specialization problems. Consider the following: $T = \{P(x)\}$, $E^+ = \{P(f(a)), P(f^2(a)), P(f^3(a)), \ldots\}$, and $E^- = \{P(a)\}$. $T' = \{P(f(x))\}$ is a solution for this specialization problem. However, no solution can be found by UD_2-specialization. Since T contains only a single unit clause, no unfolding can take place here. Hence the only UD_2-specializations of T are T itself and the empty set, neither of which is correct. So some specialization problems do not have a UD_2-specialization as a solution.

8 UDS-specialization

In this section, we will analyse why UD_2-specialization is not complete. Based on that analysis we define UDS-specialization, and prove that any specialization problem has a UDS-specialization as a solution.

[3] Henrik Boström (personal communication) made us aware of the fact that the covering algorithm of [Bos95b], with which his unfolding-algorithm SPECTRE is compared, is in fact equivalent to our UD_2-specialization. He also gave an example of a solution of a specialization problem which could be found by the covering algorithm, though not by SPECTRE, because the hypothesis-space of SPECTRE is a proper subset of the hypothesis-space of the covering-algorithm.

In our restatement of the specialization problem, one of the preconditions is that a program T' exists, such that $T \models T'$ and T' is correct w.r.t. E^+ and E^-. So, why can't UD_2-specialization always find at least one such a T'? To find such a T', we need to produce each member of T' that is not already in T. Let C be an arbitrary clause in T' that is not in T. If UD_2-specialization were complete, then we should be able to construct C from the clauses in T, using only applications of unfolding (with type 2 program, so without deleting the unfolded clauses) and clause deletion. Of these two kinds of operations, only unfolding produces new clauses. Hence if we want to produce C from the clauses in T by UD_2-specialization, C should be a result of a finite number of applications of unfolding.

Now any clause D that can be produced from T using unfolding, is related to T by a binary tree of derivation-steps by resolution, where D is the root of the tree, the leaves of the tree are definite program clauses from T, and each node in the tree is a binary resolvent of its two parents. See the left of figure 5 for an example. Here $R_1, R_2 \in T$, R_3 can be produced by unfolding R_1 or R_2, depending on whether the negative literal resolved upon is in R_1 or R_2. R_4, R_5, R_6 can also be produced by repeatedly applying unfolding. R_7 can be produced by unfolding R_4 or R_5, and finally D can be produced by unfolding R_6 or R_7. Such a tree is called a *derivation* of D from T.

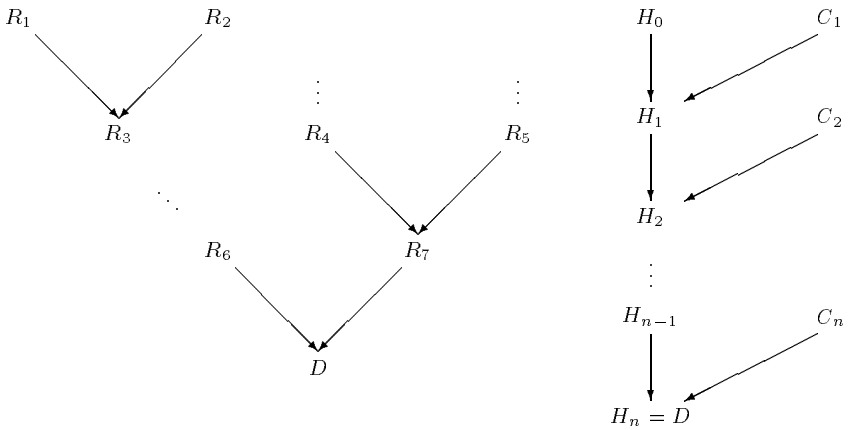


Figure 5: Using unfolding to produce a clause, which corresponds to a derivation with unconstrained resolution (on the left), or an SLD-derivation (on the right)

In [NW95a], we have given a reproof of the *subsumption theorem* of [Lee67] for unconstrained resolution, which says that for a set of clauses Σ and a clause C, $\Sigma \models C$ iff C is a tautology, or there is a D which subsumes C, and which can be derived from Σ using a derivation of unconstrained resolution. See Section 2 for the definition of subsumption.

Now the only thing we know about the definite clause C that we want to produce, is that $C \in T'$, and hence $T \models C$. Assuming C is not a tautology, the subsumption theorem tells us we can always derive by resolution from T a clause D which subsumes C. We cannot always derive C itself, using only resolution-steps. For example, if $T = \{P(x)\}$ and $C = P(f(x))$, then $T \models C$, but there is no derivation of C from T using only resolution. That is, there is

no binary tree of resolution-steps where all leaves of the tree are clauses in T, and the root of the tree is C (in fact, no resolution-steps at all are possible from this particular T).

If we cannot derive C by resolution, we cannot produce C from T using only unfolding, because unfolding can produce only clauses that can be derived as pictured on the left of figure 5. This analysis shows why it is not always possible to get from T to T' using only UD_2-specialization: we lack the possibility of taking a subsumption step. For instance, we cannot get from $T = \{P(x)\}$ to $T' = \{P(f(x))\}$ if we cannot take a subsumption step.

Clearly, the same problem obtains also for UD_1-specialization. Moreover, the clauses R_i on the left of figure 5 may be needed more than once to find D. Hence deleting these clauses (especially clauses from the original T, as we will show later on) after they have been unfolded once, is not advisable. That in the type 1 program resulting from unfolding the unfolded clause is no longer present, is one of the reasons for the fact that UD_1-specialization is less complete than UD_2-specialization.

The previous analysis also suggests a way of patching up UD_2-specialization: add the possibility of a subsumption step. If we allow this, we get UDS-specialization (**U**nfolding, clause **D**eletion, **S**ubsumption):

Definition 11 Let T and T' be definite programs. We say T' is a *UDS-specialization* of T, if there exists a sequence $T_1 = T, T_2, \ldots, T_n = T'$ $(n \geq 1)$ of definite programs, such that for each $j = 1, \ldots, n \Leftrightarrow 1$, $T_{j+1} = T_{ju_{2,C,i}}$, or $T_{j+1} = T_j \backslash \{C\}$ for some $C \in T_j$, or $T_{j+1} = T_j \cup \{C\}$ for some C that is subsumed by a clause in T_j. $\diamond$

UDS-specialization strictly generalizes UD_2-specialization. To prove the completeness of UDS-specialization, we will use the subsumption theorem. It should be noted that the subsumption theorem for unconstrained resolution only guarantees completeness in case so-called *factors* can be used (see [NW95a]). That is, the resolvents used in an unconstrained derivation are not always *binary* resolvents. However, the resolution steps that are implicit in an unfolding-step only use binary resolvents, which means that we cannot use the subsumption theorem for unconstrained resolution here. Fortunately, since we are only concerned with *Horn* clauses here, we can use an easier version of the subsumption theorem, which does not require factors. The following is the subsumption theorem for SLD-resolution, which we have proved in [NW95c]:

Theorem 3 (Subsumption theorem for SLD-resolution) *Let T be a set of Horn clauses, and C be a Horn clause. Then $T \models C$ iff $T \vdash_{ds} C$.*

Using the subsumption theorem for SLD-resolution, we can prove the following powerful result:

Theorem 4 *Let T and T' be definite programs, such that T' contains no tautologies. Then $T \models T'$ iff T' is a UDS-specialization of T.*

Proof

$\Rightarrow$: Given T and non-tautologous T', such that $T \models T'$. Let C be an arbitrary clause in T' that is not a member of T. We will show how several applications of unfolding, and a single subsumption step, can add C to the program. Doing this for all such C, we obtain a UDS-specialization T'' of T, which contains all clauses of T'. Then by deleting some clauses we get T'.

$T \models C$ (because $T \models T'$) and C is not a tautology, so by the subsumption theorem for SLD-resolution, there exists an SLD-derivation $H_0, \ldots, H_n = D$ from T, such that D subsumes C. Let $B_{i_0}, B_{i_1}, \ldots, B_{i_{n-1}}$ be the selected atoms in $H_0, H_1, \ldots, H_{n-1}$, respectively. (Note that these atoms appear in the bodies of their respective clauses, so they are actually negative literals.) See figure 6.

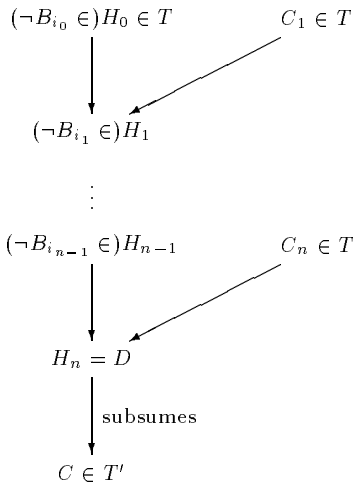


Figure 6: An SLD-derivation of D from T, such that D subsumes $C \in T'$

Since $H_0, C_1 \in T$, unfolding H_0 upon B_{i_0} results in the addition of H_1 to the program (i.e., H_1 is a member of T_{u2,H_0,i_0}). Then unfolding H_1 upon B_{i_1} adds (among others) H_2 to the program. Unfolding H_2 upon B_{i_2} yields H_3, etc. Finally, after unfolding H_{n-1} upon $B_{i_{n-1}}$, $H_n = D$ is a member of the program. Since C is subsumed by D, we can now add C to the program. In this way we can apply unfolding and the addition of a subsumed clause to add C (and a lot of other clauses) to the program.

Do this for any $C \in T'$ which is not in T. Call the program obtained after doing this, T''. Because any $C \in T'$ is now in T'', we have $T' \subseteq T''$. Since T'' is obtained from T by a finite number of applications of unfolding and the addition of subsumed clauses, T'' is a UDS-specialization of T (note that $T \subseteq T''$, but T and T'' are still logically equivalent). Now delete all members of T'' that are not in T'. This way we obtain T' from T'' as a UDS-specialization of T.

$\Leftarrow$: If $T_{j+1} = T_{j_{u2,C,i}}$, then $T_j \models T_{j+1}$. If T_{j+1} is obtained from T_j by adding a clause that is subsumed by a clause in T_j, or if $T_{j+1} = T_j \backslash \{C\}$ for some $C \in T_j$, then we also have $T_j \models T_{j+1}$. Hence $T \models T'$.

□

Suppose we are given T, T', E^+ and E^-, such that $T \models T'$ and T' is correct w.r.t. E^+ and E^-. We can assume T' contains no tautologies. Then it follows

from the previous theorem that T' is a UDS-specialization of T. This shows that UDS-specialization is complete:

Corollary 4 (Completeness of UDS-specialization) *Any specialization problem has a UDS-specialization as solution.*

Now we can also see more clearly the different reasons for the incompleteness of UD_1- and UD_2-specialization. Suppose we start with T, and there is a correct program (a "target program") T' such that $T \models T'$. Type 1 unfolding preserves the least Herbrand model of the program, but need not preserve equivalence. So even if no clauses are deleted explicitly in a deletion step, UD_1-specialization may still produce a T_i which no longer implies T'. Then T' becomes unreachable for UD_1-specialization.

On the other hand, as long as no clauses are deleted from the program, UD_2-specialization preserves equivalence. Thus any program T_i produced by type 2 unfolding, still implies the target program T'. But due to the lack of a subsumption step, this target program cannot always be reached from the original T by UD_2-specialization.

The subsumption theorem explains not only why UD_1 and UD_2 are not complete, but also shows that the type 1 and type 2 programs are inefficient. Namely, if we want to unfold some clause C, we only need to consider the resolvents of C and clauses from the original T. This is clear from figure 6, because to produce H_{i+1}, we only need to resolve H_i against C_{i+1}, and C_{i+1} is a member of the original T. In other words, we only need to add a subset of $U_{C,i}$ to the program. We might define $U'_{C,i}$ as the set of resolvents upon B_i of C and clauses from the original T, and then use $T_{j+1} = T_j \cup U'_{C,i}$ instead of $T_j \cup U_{C,i}$. This reduces the number of clauses that unfolding produces, and hence improves efficiency.

9 Relation with inverse resolution

In ILP, there are basically two possible approaches: the top-down approach (of which UDS-specialization is an example) which starts with a too-general program and specializes this, and the bottom-up approach which starts with a too-specific program and generalizes this. There is an interesting relation between our previous analysis of program specialization on the one hand, and program generalization by *inverse resolution* (see, for instance [MB92, Mug92]) on the other hand. In the case of specialization, we have some negative examples implied by T, and we want to find a T' such that $T \models T'$, and T' no longer implies those negative examples. Now program generalization tries to do the exact opposite: we have a T which does not imply certain positive examples, and we want to generalize this to some T', which implies T, and which also implies those positive examples. Thus the specialization problem and an analogous *generalization problem* can be viewed as dual problems.

In ILP, a well-known approach approach towards generalization is the inversion of resolution. Here the inversion of a resolution-step can be viewed as the dual of unfolding. However, in the same way as specialization is not complete

without subsumption, its dual also needs (the inversion of) subsumption. It can be seen from figure 6 that for completeness[4], the process of program generalization needs two operators: the ability to invert some resolution steps, but also the ability to invert a subsumption step. Most research in inverse resolution has focused on inverting resolution steps, ignoring the inversion of the final subsumption step of the deduction.[5] However, by the previous analysis, inverting a subsumption step will be necessary for completeness. For example, we cannot generalize $T = \{P(f(x))\}$ to $T' = \{P(x)\}$ only by inverting resolution steps.

So while program *specialization* by UDS-specialization corresponds to a top-down approach towards figure 6, program *generalization* by inverse resolution (which should include inverse subsumption for completeness) corresponds to a bottom-up approach. In a way, figure 6 shows a "ladder" which is climbed downward in case of program specialization, and upward in case of generalization.

10 Future work concerning implementation

Theorem 4 tells us that for *given* T and T', T' can be reached from T by a finite number of applications of unfolding, clause deletion, and subsumption. However, the usual practice is that we are given only T, E^+, and E^-, and our task is then to *find* an unknown specialization T' that is correct w.r.t. E^+ and E^-. Given that such a T' exists, how can we find it in a finite number of steps?

A natural way is the following. Define $R_0 = T$, and let R_i be the union all i-step resolvents from T (i.e., all clauses such as H_i in figure 6), and R_{i-1}. Then R_i contains all clauses that can be derived from T by SLD-derivations of length $\leq i$. This means R_i contains all clauses that can be produced by at most i applications of unfolding. Now let S_i be the set of all clauses subsumed by a clause in R_i. Note that $R_i \subseteq S_i$, since any clause subsumes itself. Clearly $S_0 \subseteq S_1 \subseteq S_2 \subseteq \ldots$. It follows from our proof of Theorem 4 that there exists an i such that $T' \subseteq S_i$. However, there is a problem here. Whereas each R_i is finite (up to alphabetical variants), each S_i is infinite, since a clause may subsume an infinite number of other clauses. For instance, $P(x)$ subsumes $P(f(x))$, $P(f^2(x))$, $P(f^3(x))$, etc. Hence you cannot construct S_{i+1} by first constructing the complete set S_i.

To avoid this, we might use some appropriate complexity-measure on clauses (such as *newsize*, which consists of a pair of natural numbers and is defined in [LN93]). For a fixed size m, the set of all clauses bounded by (i.e., "smaller" than or equal to) m will be finite up to alphabetical variants. Let $m_0, m_1, m_2, \ldots$ be a monotonically increasing sequence of bounds of the complexity measure. Define F_i as the set of all clauses in S_i bounded by m_i. Then each F_i is a finite

[4] By the *completeness* of a generalization technique, we mean the ability to find (given T) all T', such that $T' \models T$ and T' is correct w.r.t. the examples.

[5] The only papers we are aware of that try to include full inverse subsumption into inverse resolution, are [Rou92, SA93]. However, the approach to inverse subsumption taken in [Rou92] appears to be incomplete. For instance, it cannot generate $D = P(x) \leftarrow Q(x,y), Q(y,x)$ from $C = P(x) \leftarrow Q(x,x)$, even though D subsumes C. Thus Theorem 2 of that paper (which states the completeness of its inverse subsumption operator) is not correct.

set (up to alphabetical variants), which can be constructed by an algorithm. Moreover, $F_i \subseteq F_{i+1}$, for each $i \geq 0$. Since T' is bounded by some complexity measure, if we enumerate the finite sets $F_0, F_1, F_2, \ldots$, we are guaranteed to have, after a finite number of steps, a set F_i containing T'. F_i is something like the intermediate program T'' we used in the proof of Theorem 4.

However, since T' is unknown in advance, we do not know in advance for *which* i the set F_i will contain T'. Neither is it immediately obvious which subset of F_i is T'—or, equivalently, which clauses should be deleted from F_i to get T'. For this we have to take the positive and negative examples into account. That is, we should search for a correct UDS-specialization guided by the examples. For instance, all clauses in F_i which are not needed to derive any of the positive examples may be deleted.

Recall that the Introduction of this paper started with an example of how unfolding operates on SLD-trees (figure 1). For our proof of the completeness—the *existence* of a correct specialization—in the previous section, we did not need SLD-trees, but only the subsumption theorem. In fact, exclusive focus on SLD-trees might be misleading. An SLD-tree for $T \cup \{G\}$ contains all possible SLD-refutations of $T \cup \{G\}$ (for some computation rule). But an SLD-refutation of $T \cup \{G\}$ consists solely of resolution steps, and does not involve subsumption. Therefore, concentrating on the SLD-tree (such as in [BI94]) or on SLD-refutations in general (as in [Bos95a]) can lead one to ignore the subsumption step, which is necessary for completeness.

On the other hand, the SLD-tree of $T \cup \{\leftarrow P(x_1, \ldots, x_n)\}$ may be very useful in the *search* for a correct specialization, since it contains all examples of the predicate P that are derivable from the program. Note that every predicate that has to be learned, has its own SLD-tree. That is, if the examples are instances of $P(x)$ and of $Q(x, y)$, then we should consider both the SLD-tree of $T \cup \{\leftarrow P(x)\}$, and the SLD-tree of $T \cup \{\leftarrow Q(x, y)\}$. By examining these SLD-trees, we might obtain information from the examples about which clauses to unfold. For instance, consider the tree on the left of figure 1 in the Introduction. Here we only need to unfold C_1 upon $Q(x, y)$, because this is sufficient to seperate the positive example in the tree from the negative example.

In fact, the SLD-tree also shows us where we should apply a subsumption step. Consider the following specialization problem: $T = \{P(x)\}$, $E^+ = \{P(f(a)), P(f^2(a)), P(f^3(a)), \ldots\}$, and $E^- = \{P(a)\}$. The SLD-tree for $T \cup \{\leftarrow P(x)\}$ is shown in figure 7. The only leaf in this tree corresponds to the computed answer ε (the empty substitution), which shows that $P(x)$ is implied by the program. Thus this leaf corresponds to the negative example $P(a)$, but also to all positive examples. Such leaves, which correspond to positive and negative examples at the same time, are called *ambivalent* leaves.

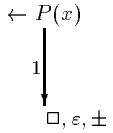


Figure 7: An SLD-tree with an ambivalent leaf

Ambivalent leaves pose a problem for UD_2-specialization, because the only way for UD_2-specialization to get rid of the negative example $P(a)$, is by pruning the whole leaf from the tree, i.e. by deleting $P(x)$ from T. However, since this leaf also corresponds to all atoms in E^+, these positive atoms are irretrievably lost also. This provides another motivation for the introduction of a subsumption step in UDS-specialization: the subsumption step is needed to get rid of ambivalent leaves. For instance, since $P(x)$ subsumes $P(f(x))$, the correct specialization $T' = \{P(f(x))\}$ can be obtained if we allow a subsumption step.

These examples show how an implementation of UDS-specialization can make use of the properties of the SLD-trees for T and the predicates that have to be learned. As we have seen, ambivalent leaves in the tree induce subsumption steps. Similarly, negative leaves indicate which clauses could be deleted (after all, our whole object is to prune negative leaves by clause deletion). The positive leaves should be used for control, they may not be pruned from the tree. Constructing an efficient strategy for finding UDS-specializations, along the lines of the positive and negative examples in the SLD-tree, is an important topic for future research.

Another extension of the research of this paper would be to consider general clauses, rather than only definite clauses as we have done here. In this case, we should allow the use of factors in the unfolding step. Then the completeness of this extended UDS-specialization can be proved using the subsumption theorem for unconstrained resolution, in the same way as we have shown the completeness of UDS-specialization for definite clauses here. Here we should take into account that derivations of unconstrained resolution may be arbitrary binary trees. The dual case of inverse resolution can also be extended to general clauses, by including the inversion of factors and by inverting arbitrary binary resolution-trees.

11 Conclusion

In this paper, we have discussed the problem of specializing a definite program with respect to sets of positive and negative examples, following [BI94]. We have shown that there exist sets of examples that have no correct program. Hence it only makes sense to talk about specialization problems for which a solution exists. This led to a reformulation of the specialization problem. To solve this problem, we first introduced UD_1-specialization, based upon the transformation rule *unfolding*. Since UD_1-specialization is incomplete, we generalized it to the stronger UD_2-specialization, which also turned out to be incomplete. We related unfolding to the subsumption theorem for SLD-resolution. We then defined UDS-specialization (a generalization of UD_2-specialization), and showed it to be complete. We also described the relation between program specialization by unfolding and generalization by inverse resolution. Efficient methods for the implementation of this specialization-technique will be a topic for future research.

References

[AGB95] Alexin, Z., Gyimóthy T., and Boström, H., 'Integrating Algorithmic Debugging and Unfolding Transformation in an Interactive Learner', in: *Proc. of the Fifth Int. Workshop on Inductive Logic Programming (ILP-95)*, Leuven, September 1995, pp. 437–453.

[BI94] Boström, H., and Idestam-Almquist, P., 'Specialization of Logic Programs by Pruning SLD-Trees', in: *Proc. of the Fourth Int. Workshop on Inductive Logic Programming (ILP-94)*, 1994, pp. 31–48.

[Bos95a] Boström, H., 'Specialization of Recursive Predicates', in: *Proc. of the European Conference on Machine Learning (ECML-95)*, Springer-Verlag, Berlin, 1995, pp. 92–106.

[Bos95b] Boström, H., 'Covering vs. Divide-and-Conquer for Top-Down Induction of Logic Programs', in: *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI-95)*, Morgan Kaufmann, 1995, pp. 1194–1200.

[LN93] van der Laag, P., and Nienhuys-Cheng, S.-H., 'Subsumption and Refinement in Model Inference', in: *Proc. of the European Conference on Machine Learning (ECML-93)*, Springer-Verlag, Berlin, 1993, pp. 95–114.

[Lee67] Lee, R. C. T., *A Completeness Theorem and a Computer Program for Finding Theorems Derivable from Given Axioms*, PhD Thesis, University of California, Berkeley, 1967.

[Llo87] Lloyd, J., *Foundations of Logic Programming*, 2nd edition, Springer-Verlag, Berlin, 1987.

[MB92] Muggleton, S., and Buntine, W., 'Machine Invention of First-Order Predicates', in: Muggleton, S. (ed.), *Inductive Logic Programming*, Academic Press, 1992, pp. 261–278.

[Mug92] Muggleton, S., 'Inductive Logic Programming', in: Muggleton, S. (ed.), *Inductive Logic Programming*, Academic Press, 1992, pp. 3–27.

[MP94] Muggleton, S., and Page, C. D., 'Self-Saturation of Definite Clauses', in: *Proc. of the Fourth Int. Workshop on Inductive Logic Programming (ILP-94)*, 1994, pp. 161–174.

[NW95a] Nienhuys-Cheng, S.-H., and de Wolf, R., 'The Subsumption Theorem in Inductive Logic Programming: Facts and Fallacies', in: *Proc. of the Fifth Int. Workshop on Inductive Logic Programming (ILP-95)*, Leuven, September 1995, pp. 147–160.

[NW95b] Nienhuys-Cheng, S.-H., and de Wolf, R., 'Specializing Definite Programs by Unfolding', in: *Proc. of Benelearn'95*, Workreport of the Université Libre de Bruxelles, September 1995.

[NW95c] Nienhuys-Cheng, S.-H., and de Wolf, R., 'The Subsumption Theorem Revisited: Restricted to SLD-resolution', to appear in: *Proc. of Computing Science in the Netherlands (CSN-95)*, Utrecht, November 1995.

[Rou92] Rouveirol, C., *Extensions of Inversion of Resolution Applied to Theory Completion*, in: Muggleton, S. (ed.), *Inductive Logic Programming*, Academic Press, 1992, pp. 63–92.

[SA93] Sato, T., and Akiba, S., 'Inductive Resolution', in: Lecture Notes in Artificial Intelligence, no. 744, Springer Verlag, 1993, pp. 101–110.

[Sha81] Shapiro, E., *Inductive Inference of Theories from Facts*, Research report 192, Yale University, 1981.

[TS84] Tamaki, H., and Sato, T., 'Unfold/Fold Transformation of Logic Programs', in: *Proc. Second Int. Logic Programming Conference*, Uppsala, 1984, pp. 127–138.

A A proof of Proposition 1

In this appendix, we give the proof of Proposition 1. This proof uses several results from [Llo87].

Proposition 1 *Let T be a definite program, G a definite goal, and $T_{u1,C,i}$ the type 1 program resulting from unfolding C upon B_i in T. Then $T \cup \{G\} \vdash_{rs} \square$ iff $T_{u1,C,i} \cup \{G\} \vdash_{rs} \square$.*

Proof

$\Rightarrow$: Suppose $T \cup \{G\} \vdash_{rs} \square$, and suppose C (the unfolded clause), is $A \leftarrow B_1, \ldots, B_i, \ldots, B_n$, which we abbreviate to $A \leftarrow \overline{B_1}, B_i, \overline{B_2}$ (where $\overline{B_1} = B_1, \ldots, B_{i-1}$ and $\overline{B_2} = B_{i+1}, \ldots, B_n$). B_i is the atom unfolded upon. If there is an SLD-refutation of $T \cup \{G\}$ in which C isn't used as an input clause, then this is also an SLD-refutation of $T_{u1,C,i} \cup \{G\}$. But suppose C is used as input clause in all SLD-refutations of $T \cup \{G\}$. We will prove that from such a refutation, a refutation of $T_{u1,C,i} \cup \{G\}$ can be constructed.

Suppose we have a refutation of $T \cup \{G\}$ of length n, with goals $G_0, \ldots, G_n$ and input clauses $C_1, \ldots, C_n$, which uses C at least once as input clause. By the independence of the computation rule (Theorem 9.2 of [Llo87]), we can assume that for any k, if C is the input clause in the step leading from G_{k-1} to G_k, then the instance of B_i that is inserted in G_k by C, is the selected atom in G_k.

Suppose the j-th input clause is C. We picture this part of the refutation on the left of figure 8. For this picture, we make the following notational conventions:

- G_{j-1}, the $j \Leftrightarrow 1$-th goal, is the goal $\leftarrow A_1, \ldots, A_k, \ldots, A_m$, which we abbreviate to $\leftarrow \overline{A_1}, A_k, \overline{A_2}$.
- The input clause used in the $(j+1)$-th step is $C_{j+1} = A' \leftarrow \overline{B'}$, where $\overline{B'}$ is an abbreviation of $B'_1, \ldots, B'_r$.

- θ_j is an mgu of A_k and A (used in the j-th resolution step).
- θ_{j+1} is an mgu of $B_i\theta_j$ and A' (used in the $(j+1)$-th resolution step).

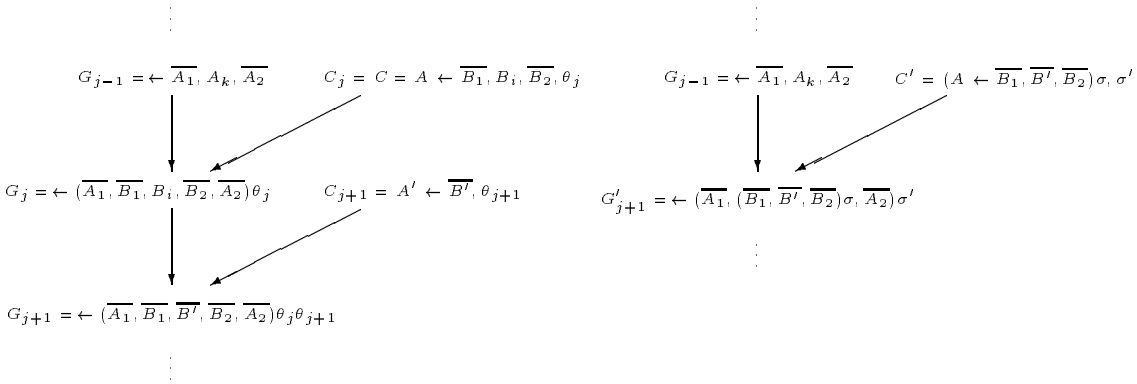


Figure 8: From the tree on the left, we can construct the tree on the right, using C' instead of C.

Since the $(j+1)$-th step of the tree on the left of figure 8 shows that B_i and A' can be unified (say, with mgu σ), the clause $C' = (A \leftarrow \overline{B_1}, \overline{B'}, \overline{B_2})\sigma$ (the result of resolving C with $C_{j+1} = A' \leftarrow \overline{B'}$) must be in $U_{C,i}$. We assume without loss of generality that G_{j-1}, $C_j = C$, C_{j+1}, and C' are standardized apart (i.e., they have no variables in common).

What we want is to construct a tree which, instead of using C in the j-th step, uses C'. For this, we will show that G_{j+1} is a variant of the goal G'_{j+1}, which can be derived from G_{j-1} and C'. Then we can replace the j-th step (which uses C) and the $(j+1)$-th step by one single step which doesn't need C anymore, but instead uses C'.

Now θ_{j+1} is an mgu of A' and $B_i\theta_j$ and $A'\theta_j = A'$ (because of the standardizing apart), so $\theta_j\theta_{j+1}$ is a unifier of A' and B_i. σ is an mgu of A' and B_i, so there exists a substitution γ such that $\sigma\gamma = \theta_j\theta_{j+1}$. $A\sigma\gamma = A\theta_j\theta_{j+1} = A_k\theta_j\theta_{j+1} = A_k\sigma\gamma = A_k\gamma$, so γ is a unifier of $A\sigma$ and A_k. This shows that $A\sigma$ and A_k can be unified. Let σ' be an mgu of $A\sigma$ and A_k. Let $G'_{j+1} = \leftarrow (\overline{A_1}, (\overline{B_1}, \overline{B'}, \overline{B_2})\sigma, \overline{A_2})\sigma'$ be the goal derived from G_{j-1} and C'. We will show that G_{j+1} and G'_{j+1} are variants.

1. We have already shown that γ is a unifier of $A\sigma$ and A_k. Furthermore, σ' is an mgu of $A\sigma$ and A_k, so there exists a substitution δ such that $\sigma'\delta = \gamma$.

   Now $G_{j+1} = \leftarrow (\overline{A_1}, \overline{B_1}, \overline{B'}, \overline{B_2}, \overline{A_2})\theta_j\theta_{j+1} = \leftarrow (\overline{A_1}, \overline{B_1}, \overline{B'}, \overline{B_2}, \overline{A_2})\sigma\gamma = \leftarrow (\overline{A_1}, \overline{B_1}, \overline{B'}, \overline{B_2}, \overline{A_2})\sigma\sigma'\delta = \leftarrow (\overline{A_1}, (\overline{B_1}, \overline{B'}, \overline{B_2})\sigma, \overline{A_2})\sigma'\delta = G'_{j+1}\delta$

2. σ' is an mgu of A_k and $A\sigma$, and $A_k\sigma = A_k$ (because of the standardizing apart), so $\sigma\sigma'$ is a unifier of A_k and A. Furthermore, θ_j is an mgu of A_k and A, so there exists a substitution γ' such that $\theta_j\gamma' = \sigma\sigma'$.

   $A'\gamma' = A'\theta_j\gamma' = A'\sigma\sigma' = B_i\sigma\sigma' = B_i\theta_j\gamma'$, so γ' is a unifier of A' and $B_i\theta_j$. θ_{j+1} is an mgu of A' and $B_i\theta_j$, so there exists a substitution δ' such that $\theta_{j+1}\delta' = \gamma'$.

Now $G'_{j+1} = \leftarrow (\overline{A_1}, (\overline{B_1}, \overline{B'}, \overline{B_2})\sigma, \overline{A_2})\sigma' = \leftarrow (\overline{A_1}, \overline{B_1}, \overline{B'}, \overline{B_2}, \overline{A_2})\sigma\sigma' = \leftarrow (\overline{A_1}, \overline{B_1}, \overline{B'}, \overline{B_2}, \overline{A_2})\theta_j\gamma' = \leftarrow (\overline{A_1}, (\overline{B_1}, \overline{B'}, \overline{B_2})\sigma, \overline{A_2})\theta_j\theta_{j+1}\delta' = G_{j+1}\delta'$

We have shown that $G_{j+1} = G'_{j+1}\delta$ and $G'_{j+1} = G_{j+1}\delta'$, so G_{j+1} and G'_{j+1} are variants.

Since G_{j+1} and G'_{j+1} are variants, we have shown that the two resolution steps leading from G_{j-1} to G_{j+1} can be replaced by a single resolution step, which uses C' as input clause. In the same way, we can eliminate all other uses of C as input clause in the rest of the tree, by constructing a refutation which uses some clause in $U_{C,i}$ to replace a usage of C, each time replacing two resolution steps by one single resolution step. Finally we get an SLD-refutation of $T \cup U_{C,i} \cup \{G\}$ which doesn't use C at all. This means that we have in fact found an SLD-refutation of $T_{u1,C,i} \cup \{G\}$.

$\Leftarrow$: Suppose $T_{u1,C,i} \cup \{G\} \vdash_{rs} \square$. Then by the soundness of resolution, $T_{u1,C,i} \cup \{G\}$ is unsatisfiable. It is easy to see that $T \models T_{u1,C,i}$. Hence $T \cup \{G\}$ is unsatisfiable, and by Theorem 8.4 of [Llo87], we have $T \cup \{G\} \vdash_{rs} \square$.

$\square$