A Framework for the Integration of Partial Evaluation and Abstract Interpretation of Logic Programs

Michael Leuschel
Declarative Systems and Software Engineering
Department of Electronics and Computer Science
University of Southampton
Southampton SO17 1BJ, UK
FAX: +44 23 80 59 3045
E-MAIL: mal@ecs.soton.ac.uk
WWW: http://www.ecs.soton.ac.uk/~mal

Recently the relationship between abstract interpretation and program specialization has received a lot of scrutiny, and the need has been identified to extend program specialization techniques so to make use of more refined abstract domains and operators. This paper clarifies this relationship in the context of logic programming, by expressing program specialization in terms of abstract interpretation. Based on this, a novel specialization framework, along with generic correctness results for computed answers and finite failure under SLD-resolution, is developed.

This framework can be used to extend existing logic program specialization methods, such as partial deduction and conjunctive partial deduction, to make use of more refined abstract domains. It is also shown how this opens up the way for new optimizations. Finally, as shown in the paper, the framework also enables one to prove correctness of new or existing specialization techniques in a simpler manner.

The framework has already been applied in the literature to develop and prove correct specialization algorithms using regular types, which in turn have been applied to the verification of infinite state process algebras.

Categories and Subject Descriptors: D.1.2 [**Programming Techniques**]: Automatic Programming; I.2.2 [**Artificial Intelligence**]: Automatic Programming; I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving—*logic programming*; D.1.6 [**Programming Techniques**]: Logic Programming; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*logic programming*

Additional Key Words and Phrases: Partial Deduction, Partial Evaluation, Program Transformation, Abstract Interpretation, Logic Programming, Flow analysis

1. INTRODUCTION

Program specialization aims at improving the overall performance of programs by performing source to source transformations. The central idea is to specialize a given source program for a particular application domain, with the goal of obtaining a less general but more efficient program. This is (mostly) done by a *well-automated* application of parts of the Burstall and Darlington unfold/fold [Burstall and Darlington 1977] transformation framework. Program specialization encom-

Part of the work was done while the author was Post-doctoral Fellow of the Fund for Scientific Research - Flanders Belgium (FWO) at the K.U. Leuven, Belgium as well as visiting DIKU, University of Copenhagen, Denmark.

passes traditional compiler optimization techniques [Muchnick 1997], such as *constant folding* (i.e., the evaluation of expressions whose arguments are constants) and *in-lining* (i.e., the substitution of a procedure call by the procedure's body), but uses more aggressive transformations, yielding both (much) greater speedups and more difficulty in controlling the transformation process. It is thus similar in concept to, but in several ways stronger than highly optimizing compilers. A common approach, known as *partial evaluation* is to guide the transformation by partial knowledge about the input. In the context of pure logic programs, partial evaluation is sometimes referred to as *partial deduction*.

Program analysis is about statically inferring information about dynamic program properties. *Abstract interpretation* [Cousot and Cousot 1977] was developed as a very general, formal framework for specifying and validating program analyses. The main idea of using abstract interpretation for program analysis is to interpret the programs to be analyzed over some *abstract domain*. This is done in such a way as to ensure termination of the abstract interpretation and to ensure that the so derived results are a *safe approximation* of the programs' concrete runtime behavior(s).

Abstract Interpretation vs. Program Specialization. At first sight *abstract interpretation* and *program specialization* might appear to be unrelated techniques: abstract interpretation focusses on *correct and precise* analysis, while the main goal of program specialization is to produce more *efficient specialized code* (for a given task at hand). Nonetheless, it is often felt that there is a close relationship between abstract interpretation and program specialization and, recently, there has been a lot of interest in the integration and interplay of these two techniques (see, e.g., [Consel and Khoo 1993; Puebla and Hermenegildo 1995; Leuschel and De Schreye 1996; Jones 1997; Puebla et al. 1997; Puebla et al. 1999; Puebla and Hermenegildo 1999; Gallagher and Peralta 2001]).

From Partial Deduction to Abstract Partial Deduction. In this paper we would like to make the relationship between partial deduction and abstract interpretation more concrete, and provide a formal framework for integrating these two techniques. This will also pave the way for new, much more powerful specialization (and analysis) techniques, e.g., by using more refined abstract domains. Indeed, "classical" partial deduction turns out to be often too limited (see, e.g., [Gallagher and de Waal 1992; de Waal and Gallagher 1994; Leuschel and De Schreye 1996; Leuschel and Lehmann 2000] to name just a few) and a lot of extensions have been developed to remedy its shortcomings (such as partial deduction with characteristic trees [Gallagher and Bruynooghe 1991; Leuschel et al. 1998], constrained partial deduction [Leuschel and De Schreye 1998], conjunctive partial deduction [Leuschel et al. 1996; Glück et al. 1996; De Schreye et al. 1999]). However, every time such an extension is developed, correctness has to be re-established from scratch: a very tedious and time-consuming process. By providing a very general framework, we want to reduce this work to minimum (at the same time allowing more powerful extensions): when developing a new instance of the framework one just has to prove some basic properties of the underlying operations and one can then re-apply the correctness results presented in this paper with minimal effort. Finally, the framework also allows the tupling [Chin and Khoo 1993] and deforestation [Wadler 1990]

capabilities of conjunctive partial deduction to be added to abstract interpretation.

Overview. After introducing the essence of partial deduction in Section 2, we investigate the relationship between partial deduction and program analysis in Section 3. Then, we define the notion of abstract domains in Section 4, we present in Section 5 the important concepts of abstract unfolding and abstract resolution which will be at the heart of our framework. In Section 6 we then show how these concepts can be used to develop atomic abstract partial deduction. In Section 7 we then show how this can be extended to cover abstract conjunctions. In Section 8 we then formally prove our generic correctness results. In Section 9 we cast some existing techniques into our framework. We show how success information propagation can be added to our framework in Section 10. We conclude with a discussion of related and further work in Sections 11 and 12.
This paper is based on the earlier conference paper [Leuschel 1998b].

2. BASICS OF PARTIAL DEDUCTION

In this section we present the technique of partial deduction, which originates from [Komorowski 1982]. Other introductions to partial deduction can be found in [Komorowski 1992; Gallagher 1993; Leuschel 1999]. Note that, for clarity's sake, we deviate slightly from the original formulation of [Lloyd and Shepherdson 1991] and use the formulation from [Leuschel and Bruynooghe 2002]. We also restrict our attention to definite logic programs and the SLD procedural semantics.

In contrast to ordinary evaluation, partial evaluation is processing a given program P along with only *part* of its input, called the *static input*. The remaining part of the input, called the *dynamic input*, will only be known at some later point in time (which we call *runtime*). Given the static input S, the partial evaluator then produces a *specialized* version P_S of P which, when given the dynamic input D, produces the same output as the original program P. The program P_S is also called the *residual program*.

Partial evaluation [Consel and Danvy 1993; Jones et al. 1993; Jones 1996; Mogensen and Sestoft 1997] has been applied to many programming languages: e.g., functional programming languages, logic programming languages, functional logic programming languages, term rewriting systems, or imperative programming languages. In the context of logic programming [Apt 1990; Lloyd 1987], full input to a program P consists of a goal G and evaluation can be seen as constructing a complete SLD-tree for $P \cup \{G\}$. For partial evaluation, the static input takes the form of a goal G' which is more general (i.e., less instantiated) than a typical goal G at runtime. In contrast to other programming languages, one can still execute P for G' and (try to) construct an SLD-tree for $P \cup \{G'\}$. So, at first sight, it seems that partial evaluation for logic programs is almost trivial and just corresponds to ordinary evaluation. However, since G' is not yet fully instantiated, the SLD-tree for $P \cup \{G'\}$ is usually infinite and ordinary evaluation will not terminate. A technique which solves this problem is known under the name of *partial deduction*. Its general idea is to construct a finite number of finite, but possibly *incomplete* SLD trees and to extract from these trees a new program that allows any instance of the goal G' to be executed.

Before formalizing the notion of partial deduction, we briefly recall some basics of

logic programming [Apt 1990; Lloyd 1987]. Syntactically, programs are built from an alphabet of variables (as usual in logic programming, variable names start with a capital), function symbols (including constants) and predicate symbols. Terms are inductively defined over the variables and the function symbols. Formulas of the form $p(t_1, \ldots, t_n)$ with p/n a predicate symbol of arity $n \geq 0$ and $t_1, \ldots, t_n$ terms are atoms. A *definite clause* is of the form $a \leftarrow B$ where the head a is an atom and the body B is a conjunction of atoms. A formula of the form $\leftarrow B$ with B a conjunction of atoms is a *definite goal*. Definite *programs* are sets composed of definite clauses. In analogy with terminology from other programming languages, an atom in a clause body or in a goal is sometimes referred to as a *call*. As we restrict our attention to definite clauses, programs, and goals we will often drop the "definite" prefix and just refer to clauses, programs, and goals.

As detailed in [Apt 1990; Lloyd 1987] a *derivation step* selects an atom in a definite goal according to some *selection rule*. Using a program clause, it first renames apart the program clause to avoid variable clashes and then computes a most general unifier (*mgu*) between the selected atom and the clause head and, if an *mgu* exists, derives the *resolvent*, a new definite goal. (We also say that the selected atom is *resolved* with the program clause.) Now, we are ready to introduce our notion of SLD-derivation. As common in works on partial deduction, it differs from the standard notion in logic programming theory by allowing a derivation that ends in a nonempty goal where no atom is selected.

Definition 2.1 Let P be a definite program and G a definite goal. An *SLD-derivation* for $P \cup \{G\}$ consists of a possibly infinite sequence $G_0 = G$, G_1, ... of goals, a sequence C_1, C_2, ... of properly renamed clauses of P, a sequence $L_0, L_1 \ldots$ of selected atoms and a sequence θ_1, θ_2, ... of mgus such that each G_{i+1} is derived from G_i and C_{i+1} using selected literal L_i and *mgu* θ_{i+1}.

The initial goal of an SLD-derivation is also called the *query*. An SLD-derivation is a successful derivation or refutation if it ends in the empty goal, a failing derivation if it ends in a goal with a selected atom that does not unify with any properly renamed clause head, an incomplete derivation if it ends in a nonempty goal without selected atom; if none of these, it is an infinite derivation. In examples, to distinguish an incomplete derivation from a failing one, we will extend the sequence of a failing derivation with the atom **fail**. The totality of SLD-derivations form a search space. One way to organize this search space is to structure it in an SLD-tree. The root is the initial goal; the children of a (non-failing) node are the resolvents obtained by selecting an atom and performing all possible derivation steps (a process that we call the *unfolding* of the selected atom). Each branch of the tree represents an SLD-derivation. A *trivial* tree is a tree consisting of a single node —the root— without selected atom.

We now examine how specialized clauses can be extracted from SLD-derivations and trees.

Definition 2.2 Let P be a program, $G = \leftarrow Q$ a goal, D a finite SLD-derivation of $P \cup \{G\}$ ending in $\leftarrow B$, and θ the composition of the *mgu*s in the derivation steps. Then the formula $Q\theta \leftarrow B$ is called the *resultant* of D. Also, θ restricted to the variables of Q is called the *computed answer substitution (c.a.s.)* of D. If D is

a refutation then θ restricted to the variables of Q is also simply called a *computed answer*.

Note that the formula $Q\theta \leftarrow B$ is a clause when Q is a single atom, which will always be the case for classical partial deduction. *Conjunctive partial deduction* (cf. Section 7) also allows Q to be a conjunction of several atoms. The relevant information to be extracted from an SLD-tree is the set of resolvents and the set of atoms occurring in the literals at the non-failing leaves.

Definition 2.3 Let P be a program, G a goal, and τ a finite SLD-tree for $P \cup \{G\}$. Let $D_1, \ldots, D_n$ be the non-failing SLD-derivations associated with the branches of τ. Then the *set of resultants*, *resultants*(τ), is the set whose elements are the resultants of $D_1, \ldots, D_n$ and the *set of leaves*, *leaves*(τ), is the set of atoms occurring in the final goals of $D_1, \ldots, D_n$.

With the initial goal atomic, the extracted resultants are program clauses: the partial deduction of the atom.

Definition 2.4 Let P be a definite program, A an atom, and τ a finite non-trivial SLD-tree for $P \cup \{\leftarrow A\}$. Then the set of clauses *resultants*(τ) is called a *partial deduction of A in P*. If $\mathcal{A}$ is a finite set of atoms, then a *partial deduction of $\mathcal{A}$ in P* is the union of the sets obtained by taking one partial deduction for each atom in $\mathcal{A}$.

In summary, the specialized program is extracted from SLD trees by constructing one specialized clause per non-failing branch. This can yield a more efficient program, as a *single* resolution step with a specialized clause now corresponds to performing *all* the resolutions steps (using original program clauses) on the associated branch. Also, failing branches have been completely removed from the specialized program, which can lead to further efficiency improvements.

Example 2.5 Let P be the following metainterpreter taken from [Leuschel 2002], which counts resolution steps:

$$solve([], Depth, Depth) \leftarrow$$
$$solve([Head|Tail], DSoFar, Res) \leftarrow clause(Head, Bdy),$$
$$\qquad solve(Bdy, s(DSoFar), IntD),\ solve(Tail, IntD, Res)$$
$$clause(mem(X, [X|T]), []) \leftarrow$$
$$clause(mem(X, [Y|T]), [mem(X, T)]) \leftarrow$$
$$clause(app([], L, L), []) \leftarrow$$
$$clause(app([H|X], Y, [H|Z]), [app(X, Y, Z)]) \leftarrow$$

Figure 1 represents an incomplete SLD-tree τ for $P \cup \{\leftarrow solve(mem(X, L), D, R)\}$. This tree has two non-failing branches and *resultants*(τ) thus contains the two clauses:

$$solve(mem(X, [X|L]), D, s(D)) \leftarrow$$
$$solve(mem(X, [Y|L]), D, R) \leftarrow solve(mem(X, L), s(D), R)$$

These two clauses are a partial deduction of $\mathcal{A} = \{solve(mem(X, L), D, R)\}$ in P. Note that the complete SLD-tree for $P \cup \{\leftarrow solve(mem(X, L), D, R)\}$ is infinite.

Observe how one resolution step in the partial deduction corresponds to three to four resolution steps in the original program. This results in the specialized

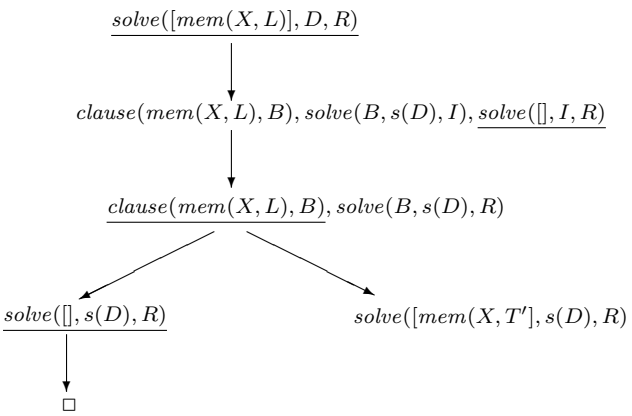

Fig. 1. Incomplete SLD-tree for Example 2.5

program being substantially faster than the original one. E.g., on a typical Prolog system and for typical runtime queries the specialized program is more than three times faster than the original.[1]

In analogy with terminology in partial evaluation, the partial deduction of A in P is also referred to as the *residual clauses* of A and the partial deduction of $\mathcal{A}$ in P as the *residual program*.

The intuition underlying partial deduction is that a program P can be replaced by a partial deduction of $\mathcal{A}$ in P and that both programs are *equivalent* with respect to queries which are constructed from instances of atoms in $\mathcal{A}$. Almost all works on partial deduction aim at preserving the procedural equivalence under SLD (and SLDNF). Before defining the extra conditions required to ensure it, we introduce a few more concepts:

Definition 2.6 Let A_1, A_2, A_3 be three atoms, such that $A_3 = A_1\theta_1$ and $A_3 = A_2\theta_2$ for some substitutions θ_1 and θ_2. Then A_3 is called a *common instance* of A_1 and A_2. Let $\mathcal{A}$ be a finite set of atoms and S a set containing atoms, conjunctions, and clauses. Then S is $\mathcal{A}$*-closed* iff each atom in S is an instance of an atom in $\mathcal{A}$. Furthermore we say that $\mathcal{A}$ is *independent* iff no pair of atoms in $\mathcal{A}$ has a common instance.

The main result of [Lloyd and Shepherdson 1991] about procedural equivalence can be formulated as follows:

Theorem 2.7
Let P be a definite program, $\mathcal{A}$ a finite, independent set of atoms, and P' a partial deduction of $\mathcal{A}$ in P. For every goal G such that $P' \cup \{G\}$ is $\mathcal{A}$-closed the following holds:

(1) $P' \cup \{G\}$ has an SLD-refutation with computed answer θ iff $P \cup \{G\}$ does.
(2) $P' \cup \{G\}$ has a finitely failed SLD-tree iff $P \cup \{G\}$ does.

[1] E.g., 3.4 times faster on Sicstus Prolog 3.8.7 running on a Powerbook G4 667 Mhz with 1 Gb RAM and Mac OS X 10.1.4.

The theorem states that P and P' are procedurally equivalent with respect to the existence of success-nodes and associated answers for $\mathcal{A}$-closed goals. The fact that partial deduction preserves equivalence only for $\mathcal{A}$-closed goals distinguishes it from e.g. unfold/fold program transformations which aim at preserving equivalence for all goals. Note that the theorem does not tell us how to obtain $\mathcal{A}$, an issue which is tackled by the *control* of partial deduction (see, e.g., [Leuschel and Bruynooghe 2002]).

Returning to Example 2.5, we have that the partial deduction of the set $\mathcal{A} = \{solve(mem(X, L), D, R)\}$ in P satisfies the conditions of Theorem 2.7 for the goals $\leftarrow solve(mem(X, [a]), 0, R)$ and $\leftarrow solve(mem(a, [X, Y]), s(0), R)$ but not for the goal $\leftarrow solve(app([], [], L), 0, R)$. Indeed, the latter goal succeeds in the original program but fails in the specialised one. Intuitively, if $P' \cup \{G\}$ is not $\mathcal{A}$-closed, then an SLD-derivation of $P' \cup \{G\}$ may select a literal for which no clauses exist in P' while clauses did exist in P. Hence, a query may fail while it succeeds in the original program.

If $\mathcal{A}$ is not independent then a selected atom may be resolved with clauses originating from the partial deduction of two distinct atoms. This may lead to computed answers that, although correct, are not computed answers of the original program. However, this can be easily remedied by a *renaming* transformation, generating new predicate names for atoms which are not independent [Benkerimi and Hill 1993]. To improve the efficiency of specialised programs, all partial deduction systems we know of, perform renaming together with so-called *filtering* [Gallagher and Bruynooghe 1990; 1991; Leuschel and Sørensen 1996; Proietti and Pettorossi 1993], which filters out constants and function symbols. E.g., for our Example 2.5, a filtered partial deduction of $\mathcal{A}$ in P would be something like the following, which delivers an additional speedup of over 1.5 compared to the partial deduction in Example 2.5:

$$solve\_1(X, [X|L], D, s(D)) \leftarrow$$
$$solve\_1(X, [Y|L], D, R) \leftarrow solve\_1(X, L, s(D), R)$$

In practice it is thus the $\mathcal{A}$-closedness condition which is the most important one. It is also this condition which best illustrates the link between partial deduction and program analysis. Indeed, as we will show in the next section, the $\mathcal{A}$-closedness condition for the residual program P' in Theorem 2.7 ensures that *together* the SLD-trees, from which the clauses in P' are derived, form a *complete description* of all possible calls that can occur for all goals G which are $\mathcal{A}$-closed.

3. PARTIAL DEDUCTION AND PROGRAM ANALYSIS

Below we denote by 2^S the power-set of some set S, by $Clauses$ the set of all clauses, by $Atoms$ the set of all atoms, and by $\mathcal{Q}$ the set of all conjunctions.

3.1 Partial Deduction as Program Analysis

In the context of a logic program P there are plenty of program properties that are of interest, such as, e.g., the logical consequences of P or the computed answers of P. The following property is a key concept in termination analysis [De Schreye and Decorte 1994] and will be of interest in relating partial deduction and program analysis.

Definition 3.1 For a program P and a conjunction Q the *call set* of $P \cup \{\leftarrow Q\}$, denoted by $calls(P, Q)$, is the set of selected atoms within all possible complete SLD-trees for $P \cup \{\leftarrow Q\}$.

We have seen in the previous section that the $\mathcal{A}$-closedness condition ensures correctness of the specialised program and the condition must thus ensure that all possible calls that can occur when running the specialised program have been taken into account by partial deduction. It is thus to be expected that some relationship between partial deduction and call sets can be established. The following proposition shows that under certain circumstances, the result of a partial deduction can indeed be viewed as a program analysis inferring information about various call sets.

Proposition 3.2 Let P be a definite program and Q a conjunction. Let $\mathcal{A}$ be a finite set of atoms, and P' a partial deduction of $\mathcal{A}$ in P such that $P' \cup \{\leftarrow Q\}$ is $\mathcal{A}$-closed. If the SLD-trees whose resultants make up P' are such that every SLD-tree has a depth of 1, i.e., every tree contains just a single unfolding step, then the following holds: $calls(P, Q) \subseteq \{A\theta \mid A \in \mathcal{A}\}$.

In the above proposition we have restricted ourselves to very simple SLD-trees, containing exactly one unfolding step. In fact, if one allows more than one unfolding step, then the relationship between $\mathcal{A}$ and the call set becomes more complicated, detracting from the point we are trying to make.[2] Below we will describe a procedure which, given P and Q, will construct $\mathcal{A}$ and P' such that $P' \cup \{\leftarrow Q\}$ is $\mathcal{A}$-closed.

Let us first illustrate Proposition 3.2 using an example.

Example 3.3 Let P be the following program:

$mem(X, [X|L]) \leftarrow$

$mem(X, [Y|L]) \leftarrow mem(X, L)$

The partial deduction P' of $\mathcal{A} = \{mem(a, L)\}$, which we obtain by performing just a single unfolding step for $P \cup \{\leftarrow mem(a, L)\}$, is as follows:

$mem(a, [a|L]) \leftarrow$

$mem(a, [Y|L]) \leftarrow mem(a, L)$

Note that $P' \cup \{\leftarrow mem(a, L)\theta\}$ is $\mathcal{A}$-closed for any substitution θ. As stated by Proposition 3.2, for any substitution θ, all elements of $calls(P, mem(a, L)\theta)$ are instances of an element of $\mathcal{A}$. Partial deduction has thus "deduced" structural information about the call set: all calls to *mem* have the constant 'a' in the first argument position.

Having identified one relationship between partial deduction and program analysis, we will now formalize this process more precisely in the abstract interpretation framework. This will clarify their relationship and pave way to an integration of abstract interpretation and partial deduction.

[2] Basically $\mathcal{A}$ then only contains information about calls at certain "program points" and infers information about the calls on successful branches only, rather than about any call.

3.2 Abstract Interpretation

Abstract interpretation [Cousot and Cousot 1977] provides a general formal framework for performing sound program analysis and has been successfully applied to the analysis of logic programs [Cousot and Cousot 1992; Bruynooghe 1991; Hermenegildo et al. 1992]. To make program analysis tractable, abstract interpretation distinguishes between a concrete domain $\mathcal{C}$ of program properties and an *abstract domain* $\mathcal{AD}$ of properties. The latter contains finite, approximate representations of (sets of) concrete properties. The concrete properties are used by a semantic function *sem* which assigns to every program P and a set of calls[3] S its (concrete) semantics $sem(P, S) \in 2^{\mathcal{C}}$. The abstract domain is linked to the concrete domain via a *concretization function* $\gamma : \mathcal{AD} \to 2^{\mathcal{C}}$, which assigns to each abstract property the (possibly infinite) set of concrete properties it represents. Program analysis is then performed by abstractly executing a program P to be analyzed in the abstract domain rather than in the concrete one. For this, abstract counterparts of the concrete operations of P have to be developed. These abstract operations have to be a *safe approximation*, in the sense that for every concrete operation $op : 2^{\mathcal{C}} \to 2^{\mathcal{C}}$, the corresponding abstract operation $op_\alpha : \mathcal{AD} \to \mathcal{AD}$ must satisfy $\gamma(op_\alpha(A)) \supseteq op(\gamma(A))$.

Under certain conditions (see [Cousot and Cousot 1977; 1992]) the overall result $abs\_sem(P, A)$ of the abstract execution of P for some abstract input value A is then also a safe approximation of the concrete properties of the program, in the sense that:

$$\gamma(abs\_sem(P, A)) \supseteq sem(P, \gamma(A))$$

3.3 Partial Deduction as Abstract Interpretation

Proposition 3.2 shows that we can view the set of (concrete) atoms $\mathcal{A}$ of a partial deduction also as an abstract program property, approximating the call set *calls*. If we try to view this in abstract interpretation terms, we would have to choose $\mathcal{C} = \mathcal{Q}$ as concrete domain and $\mathcal{AD} = 2^{\mathcal{Q}}$ as abstract domain. The proposition also suggests a concretization function γ_{inst} defined by

$$\gamma_{inst}(S) = \{A\theta \mid A \in S \wedge \theta \text{ is a substitution}\}$$

Thus $\gamma_{inst}(\{p(X, X)\})$ contains, e.g., $p(a, a)$, $p(b, b)$, $p(X, X)$, but not $p(a, b)$. An atom in the abstract domain thus represents all its instances in the concrete domain (and thus also itself).

Observe that if $P' \cup \{\leftarrow Q\}$ is $\mathcal{A}$-closed then so is $P' \cup \{\leftarrow Q\theta\}$ for any substitution θ. We can thus obtain an instance of our equation $\gamma(abs\_sem(P, A)) \supseteq sem(P, \gamma(A))$ above, by using $A = \{Q\}$, $sem(P, Qs) = \bigcup_{Q' \in Qs} calls(P, Q')$, and by substituting $abs\_sem(P, A) = \mathcal{A}$, yielding the equation:

$$\gamma_{inst}(\mathcal{A}) \supseteq \bigcup_{Q' \in \gamma_{inst}(\{Q\})} calls(P, Q')$$

[3] Programs are usually analyzed for a set of calls rather than for an individual call. Also, sometimes the semantics function is goal-independent and assigns every program P its concrete semantics $sem(P)$.

In other words, the set $\mathcal{A}$ of atoms of a partial deduction is a safe approximation of the call set, provided single unfolding steps are used and $P \bigcup \{\leftarrow Q\}$ is $\mathcal{A}$-closed.

Controlling Partial Deduction. Can we also cast the process of constructing $\mathcal{A}$ in an abstract interpretation manner, i.e., as executing abstract counterparts of concrete operations? To answer this question we first present more details on how partial deduction is actually controlled.

We first need the following definition.

Definition 3.4 An *unfolding rule* is a function which, given a program P and a conjunction Q, returns the resultants $resultants(\tau)$ of a finite, non-trivial SLD-tree τ for $P \cup \{\leftarrow Q\}$.

We also define the operation $split : 2^{\mathcal{Q}} \rightarrow 2^{Atoms}$ by

$$split(S) \;=\; \{A_i \mid A_1 \wedge \ldots \wedge A_i \wedge \ldots \wedge A_n \in S\}$$

Next, the operation $resolve : Clauses \times \mathcal{Q} \rightarrow 2^{\mathcal{Q}}$ resolves a clause with a conjunction and is defined by

$$\begin{aligned} resolve(C, A_1 \wedge \ldots \wedge A_n) = \{&A_1 \wedge \ldots A_{i-1} \wedge B\theta \wedge A_{i+1} \ldots \wedge A_n \mid \\ &\theta = mgu(H, A_i) \text{ and } H \leftarrow B \text{ is a renamed apart version of } C \;\} \end{aligned}$$

The following is a typical way (see, e.g., [Gallagher 1991; 1993; Leuschel and Bruynooghe 2002]) of controlling classical partial deduction [Lloyd and Shepherdson 1991].

Procedure 1. (Classical Partial Deduction)
Input: A program P and a conjunction Q
Output: A specialised program P' and a set of atoms $\mathcal{A}_i$ such that $P' \cup \{\leftarrow Q\}$ is $\mathcal{A}_i$-closed.
Initialize: $i = 0$, $\mathcal{A}_0 = split(Q)$
repeat
  let $\mathcal{R}_i := \{R \mid R \in resolve(C, A) \wedge\ A \in \mathcal{A}_i \wedge C \in unfold(P, A)\}$;
  let $\mathcal{N}_i := \{N \mid N \in split(R_i) \wedge\ N \notin \gamma_{inst}(\mathcal{A}_i)\}$;
  let $\mathcal{A}_{i+1} := generalize(\mathcal{A}_i \cup \mathcal{N}_i)$; **let** $i := i + 1$;
until $\mathcal{A}_{i-1} = \mathcal{A}_i$
Let $P' = \bigcup_{A \in \mathcal{A}_i} unfold(A)$

The procedure is parametrized by two operations: an unfolding rule *unfold* (cf, Definition 3.4) and a generalization operation *generalize*. The former is usually referred to as the local control while the latter embodies the so-called global control and must satisfy $\gamma_{inst}(generalize(S)) \supseteq \gamma_{inst}(S)$. This guarantees that if the procedure terminates, then $P' \cup \{\leftarrow Q\}$ is $\mathcal{A}_i$-closed. *generalize* is usually devised such that Procedure 1 terminates (cf, [Leuschel and Bruynooghe 2002]), and can then be seen as a widening operator in the abstract interpretation sense. More on that below.

The use of the *split* operation embodies the fact that classical partial deduction specializes individual atoms and not conjunctions.

Fixpoints. Before formally defining our concrete semantics, we need the following concepts.

Let T be a mapping $2^D \mapsto 2^D$, for some D. We then define $T \uparrow^0 (S) = S$ and $T \uparrow^{i+1} (S) = T(T \uparrow^i (S))$. We also define $T \uparrow^\omega (S) = \bigcup_{i<\omega} T \uparrow^i (S)$.

By the well known Knaster-Tarski fixpoint theorem we know that if T is monotonic ($I \subseteq J \Rightarrow T(I) \subseteq T(J)$) then T has a least fixpoint. Another well known fact is that if T is continuous (i.e., T is monotonic and for every sequence $I_0 \subseteq I_1 \subseteq \ldots$ we have $T(\bigcup_{n<\omega} I_n) \subseteq \bigcup_{n<\omega} T(I_n)$) then $T \uparrow^\omega (\emptyset)$ is its least fixpoint. Furthermore, it is also easy to see (by applying the above to $T_S(I) = T(I) \cup S$) that $T \uparrow^\omega (S)$ will be the least fixpoint containing S.

Concrete Semantics. We can now formalize our concrete semantics, the call set from Definition 3.1, in terms of a least fixpoint of a concrete operator $R_P : 2^{\mathcal{Q}} \rightarrow 2^{\mathcal{Q}}$ defined by

$$R_P(S) = S \cup \bigcup_{Q \in S \wedge C \in P} \mathit{resolve}(C, Q)$$

R_P is monotonic and continuous and $R_P \uparrow^\omega$ thus computes least fixpoints. The least fixpoint $R_P \uparrow^\omega (Q)$ of this operator does not yet give us the call set $\mathit{calls}(P, Q)$; it computes all possible subgoals for $P \cup \{\leftarrow Q\}$, not the selected atoms within the subgoals. To extract the selected atoms we can use the *split* operation introduced above, and we can express the call set in terms of R_P as follows: $\mathit{calls}(P, Q) = \mathit{split}(R_P \uparrow^\omega (\{Q\}))$.

Abstract semantics. We will now try to reformulate Procedure 1 as computing a fixpoint of an abstract version of R_P. Let us first define the following abstract operator $R_P^\alpha : 2^{Atoms} \rightarrow 2^{Atoms}$ defined by

$$R_P^\alpha(S) = S \cup \bigcup_{A \in S \wedge C \in \mathit{unfold}(P,A)} \mathit{resolve}(C, A)$$

First, we would like to show that R_P^α is a sound approximation of R_P and that a fixpoint of R_P^α safely approximates the least fixpoint of R_P.

First, it is straightforward to show (e.g., using Lemma 4.12 from [Lloyd and Shepherdson 1991]) that in the above definition and for single step unfolding, we can replace the condition $C \in \mathit{unfold}(P, A)$ simply by $C \in P$. Thus R_P^α is actually identical to R_P. However, we have to be careful as R_P^α works on the abstract domain, where every conjunction represents all its instances. Thus, it does not immediately follow that R_P^α is a safe approximation of R_P. To establish this, let us look at a single concrete resolution step performed by $\mathit{resolve}(C, A)$. As usual in abstract interpretation, we lift this concrete operation to sets of atoms: $\mathit{resolve}^*(C, S) = \{\mathit{resolve}(C, A) \mid A \in S\}$. The abstract counterpart in R_P^α is simply $\mathit{resolve}_\alpha(C, A) = \mathit{resolve}(C, A)$, which is a sound approximation of *resolve*, i.e., $\gamma_{inst}(\mathit{resolve}_\alpha(C, A)) \supseteq \mathit{resolve}^*(C, \gamma_{inst}(A))$. This is a corollary of Proposition 5.6 later in the paper. We have thus that

$$R_P(\gamma_{inst}(A)) \subseteq \gamma_{inst}(R_P^\alpha(A))$$

In other words, R_P^α is a safe approximation of R_P.

Observe that, in general, we do not have equality between $\gamma_{inst}(\mathit{resolve}_\alpha(C, A))$

and $resolve^*(C, \gamma_{inst}(A))$. Take, for example, $C = p \leftarrow q(X)$ and $A = p$, and we have $q(a) \in \gamma_{inst}(resolve_\alpha(p \leftarrow q(X), p))$ while $resolve^*(C, \gamma_{inst}(A)) = \{q(X)\}$.

In addition to R_P^α, Procedure 1 also applies the operations *generalize* and *split*. The former has the property $\gamma_{inst}(generalize(S)) \supseteq \gamma_{inst}(S)$ but unfortunately, it is generally not the case that $\gamma_{inst}(split(S)) \supseteq \gamma_{inst}(S)$. E.g., $\gamma_{inst}(\{p(a), q(a)\}) \not\supseteq \gamma_{inst}(\{p(a) \wedge q(a)\})$. In other words, we cannot view *split* as a generalization operator wrt γ_{inst}, and the output $\mathcal{A}_i$ of Procedure 1 is not a safe approximation of the least fixpoint of R_P.

To remedy this problem we have to use a different concretization function $\gamma^\wedge_{inst}$ which acknowledges the fact that conjunctions can be split up and which is defined by

$$\gamma^\wedge_{inst}(S) \;=\; \{Q_1 \wedge \ldots \wedge Q_n \mid Q_i \in \gamma_{inst}(S)\}$$

For $\gamma^\wedge_{inst}$, *split* is a generalization operation: $\gamma^\wedge_{inst}(split(S)) \supseteq \gamma^\wedge_{inst}(S)$, and so is *generalize*: $\gamma^\wedge_{inst}(generalize(S)) \supseteq \gamma^\wedge_{inst}(S)$. Also, the condition $N \notin \gamma_{inst}(\mathcal{A}_i)$ obviously does not affect the concretizations of $\mathcal{A}_i$. This means that termination of Procedure 1 implies that $\mathcal{A}_i$ is a semantic fixpoint wrt $\gamma^\wedge_{inst}$, in the sense that: $\gamma^\wedge_{inst}(\mathcal{A}_i) = \gamma^\wedge_{inst}(R_P^\alpha(\mathcal{A}_i))$. Even when not using Procedure 1, $\mathcal{A}$-closedness of P' in Theorem 2.7 ensures that $\mathcal{A}$ is a semantic fixpoint of R_P^α: $\gamma^\wedge_{inst}(\mathcal{A}) = \gamma^\wedge_{inst}(R_P^\alpha(\mathcal{A}))$.

Also, if an operation is a safe approximation wrt γ_{inst} then it is also a safe approximation wrt $\gamma^\wedge_{inst}$. We have thus that

$$R_P(\gamma^\wedge_{inst}(A)) \subseteq \gamma^\wedge_{inst}(R_P^\alpha(A))$$

In other words, R_P^α is a safe approximation of R_P wrt $\gamma^\wedge_{inst}$, and one can establish using the abstract interpretation framework that a fixpoint of R_P^α safely approximates the least fixpoint of R_P wrt $\gamma^\wedge_{inst}$.

From this we can thus conclude that $\mathcal{A}$-closedeness of $P' \cup \{\leftarrow Q\}$ in Proposition 3.2 ensures that $R_P \uparrow^\omega (\gamma^\wedge_{inst}(\{Q\})) \subseteq \gamma^\wedge_{inst}(\mathcal{A})$. As *split* is monotonic wrt $\gamma^\wedge_{inst}$, we can formally deduce Proposition 3.2 as follows: $calls(P, Q) \subseteq calls(P, \gamma^\wedge_{inst}(\{Q\}))$ $= split(R_P \uparrow^\omega (\gamma^\wedge_{inst}(\{Q\}))) \subseteq split(\gamma^\wedge_{inst}(\mathcal{A})) = \gamma_{inst}(\mathcal{A}) = \{A\theta \mid A \in \mathcal{A}\}$.

In summary, we have re-formulated partial deduction as a particular abstract interpretation, where

- the abstract domain is simply the powerset of the concrete domain,
- the concretisation function simply instantiates variables,
- the concrete semantics is based on SLD resolution,
- and where we have used this to formally prove Proposition 3.2.

Extension to Conjunctive Partial Deduction. Having recast the program analysis aspect of classical partial deduction as a safe abstract interpretation, it is actually not very difficult to extend this result to conjunctive partial deduction: the only[4] modification to Procedure 1 is that instead of using *split* we use a partitioning function (cf., [De Schreye et al. 1999]) *partition* satisfying $\gamma^\wedge_{inst}(partition(S)) \supseteq \gamma^\wedge_{inst}(S)$. Whereas *split* always splits conjunctions into its individual atoms, *partition* does not have to do so. For example, while $split(\{q(X) \wedge p(X) \wedge r(Z)\}) = \{p(X), q(X), r(Z)\}$ we could have $partition(\{q(X) \wedge p(X) \wedge r(Z)\}) = \{p(X) \wedge q(X), r(Z)\}$.

[4] One actually also has to extend Definition 2.4 to perform a renaming from conjunctions in heads of resultants to atoms.

The result $\mathcal{A}_i$ of the thus adapted conjunctive partial deduction Procedure 1 still safely approximates the least fixpoint of R_P wrt $\gamma^{\wedge}_{inst}$, but we no longer have $split(\gamma^{\wedge}_{inst}(\mathcal{A}_i)) = \gamma_{inst}(\mathcal{A}_i)$ as $\mathcal{A}_i$ now may contain conjunctions.

3.4 Discussion

Having established a strong relationship between partial deduction and abstract interpretation, what sets partial deduction apart from abstract interpretation in general? The major difference is linked to the use of the unfolding rule *unfold* within R^{α}_{P} (see also [Puebla et al. 1997; Puebla et al. 1999]):

—First, unless we use a simple one-step unfolding rule, this hides certain program points from the analysis. These program points are not relevant from the point of view of partial deduction, as they disappear within the residual program.

—Second, via *unfold* partial deduction constructs residual code. While the analysis component of partial deduction is a safe approximation of the call set, the requirements for the residual code are stronger: it must be *totally correct*. As we have seen in Theorem 2.7 the residual code preserves *exactly* the computed answers (no over-approximation) and the finite failures. This is something that the abstract interpretation framework does not provide.

Thus, not all of partial deduction can be cast in an abstract interpretation framework. Apart from those fundamental differences, there are further aspects that distinguish partial deduction from techniques commonly used to perform abstract interpretation of logic programs.

—Partial deduction can make use of conjunctions [De Schreye et al. 1999] with relatively little effort. This can be used to achieve optimizations such as tupling and deforestation, and can increase precision by analyzing calls together, rather than in isolation. Logic program analysis techniques typically do not analyze conjunctions, but analyze atoms in isolation (but have mechanisms of propagating some information from one call to another). However, there are exceptions such as [Boulanger and Bruynooghe 1993] and to some extent also [Marriott et al. 1990].

—The abstract domain of partial deduction is fixed and does not allow for very precise generalisation; e.g., the most specific generalisation possible of $p(a)$ and $p(b)$ is $p(X)$. To our knowledge, only one other abstract interpretation technique [Marriott et al. 1988; 1990] uses the same abstract domain. The abstract domain has the advantage of being close to the concrete domain, and we can obtain very precise results as long as we do not need generalisation (in the absence of existential variables abstract execution will be identical to concrete execution).

—In abstract interpretation of logic programs one distinguishes between bottom-up methods, based on approximating goal-independent, declarative semantics (usually T_P or model based) and top-down methods based on abstracting a goal dependent, top-down semantics (operational semantics or denotational).
Partial deduction uses the SLD procedural semantics as its basis (embodied within R_P) and is thus top-down. However, the use of the SLD procedural semantics is rather atypical. This makes it easier to generate residual code, but makes it difficult or impossible to analyse certain other properties. Notably, no

real information about the answers is derived (just about the call set). Very few abstract interpretation techniques use the SLD procedural semantics as its basis (exceptions are, e.g., [Jones and Søndergaard 1987] and [Comini and Meo 1999]). A more popular semantics for top-down abstract interpretation is based on And-Or trees [Bruynooghe 1991; Hermenegildo et al. 1992; Janssens and Bruynooghe 1992; Muthukumar and Hermenegildo 1992; Le Charlier and Van Hentenryck 1994], where it is easier to capture and propagate success information.

The various limitations of partial deduction have been realized by many researchers (e.g., [de Waal and Gallagher 1991; Gallagher and de Waal 1992; de Waal and Gallagher 1994; Puebla and Hermenegildo 1995; Leuschel 1995; Leuschel and Martens 1995; Leuschel and De Schreye 1996; Puebla et al. 1997; Puebla et al. 1999]), and various extensions of partial deduction have been developed over the years (e.g., [Gallagher and Bruynooghe 1991; Leuschel and De Schreye 1998; 1996; Leuschel et al. 1998; Gallagher and Peralta 2001]) which overcome this particular limitation.

We have made the link of existing partial deduction techniques to abstract interpretation clearer, and will use this as the basis of extending partial deduction and conjunctive partial deduction to new abstract domains. We will then provide generic correctness results for this new setting of abstract partial deduction, and also illustrate the power of this new approach on practical examples.

4. ABSTRACT DOMAINS FOR SPECIALIZATION

In this short section we introduce the concept of abstract domains as required for our framework. First, we need the following definitions. An *expression* is either a term, an atom or a conjunction of atoms. We use $E_1 \preceq E_2$ to denote that the expression E_1 is an instance of the expression E_2. By $vars(E)$ we denote the set of variables appearing in an expression E. By *mgu* we denote a (deterministic) function which computes an idempotent and relevant[5] most general unifier θ of two expressions E_1 and E_2 (and returns *fail* if no such unifier exists).

As above, we denote by $\mathcal{Q}$ the set of all conjunctions. As we have seen, even when performing classical partial deductions on atoms only, conjunctions will still appear, e.g., in the leaves of the SLD-trees produced by the unfolding rules. This justifies why our concrete domain for abstract partial deduction talks about conjunctions rather than atoms.

For $\mathcal{Q}$ we assume that the connective $\wedge$ is associative but not commutative nor idempotent. In other words, for us a conjunction can also be viewed as a list of atoms, but not as a set or multi-set of atoms. This assumption is of relevance mainly for Section 7, where we deal with code generation for conjunctive (abstract) partial deduction.

Definition 4.1 An *abstract domain* $(\mathcal{AQ}, \gamma)$ is a pair consisting of a set $\mathcal{AQ}$ of so-called *abstract conjunctions* and a total *concretization function* $\gamma : \mathcal{AQ} \rightarrow 2^{\mathcal{Q}}$, providing the link between the abstract and the concrete domain, such that $\forall \mathbf{A} \in \mathcal{AQ}$ the following hold:

[5] I.e., $\theta\theta = \theta$ and $vars(\theta) \subseteq vars(E_1) \cup vars(E_2)$. There can be several most general unifiers which satisfy that criterion; the particular choice is, however, not important.

(1) $\forall Q \in \gamma(\mathbf{A})$ we have $\{Q\theta \mid \theta \text{ is a substitution } \} \subseteq \gamma(\mathbf{A})$,
(2) $\exists Q \in \mathcal{Q}$ such that $\gamma(\mathbf{A}) \subseteq \{Q\theta \mid \theta \text{ is a substitution } \}$.

Property 1 expresses the requirement that the image of $\gamma(.)$ is *downwards closed*. This means that certain properties, such as) freeness (e.g., [Muthukumar and Hermenegildo 1991] cannot be captured, but downwards closedeness is required for our correctness proofs.

Property 2 expresses the fact that all conjunctions in $\gamma(\mathbf{A})$ have the same number of conjuncts and with the same predicates at the same position. This property is crucial to enable the construction of (correct) residual code. A conjunction Q satisfying property 2 is called a *concrete dominator* of $\mathbf{A}$. An abstract conjunction such that its concrete dominators are all atoms is called an *abstract atom*.

Observe that property 2 still admits the possibility of a bottom element $\bot$ whose concretisation is empty.

One particular abstract domain, which arises in the formalization of (classical) partial deduction [Lloyd and Shepherdson 1991] and which we have encountered in Section 3.3, is the $\mathcal{PD}$-domain defined as follows.

Definition 4.2 The $\mathcal{PD}$-domain is the abstract domain $(\mathcal{Q}, \gamma_{inst})$ where γ_{inst} is defined by $\gamma_{inst}(Q) = \{Q' \mid Q' \preceq Q\}$.

In other words, we have $\mathcal{AQ} = \mathcal{Q}$ (i.e. the abstract conjunctions are the concrete ones) and an abstract conjunction denotes the set of all its instances. For example, we can use the (concrete) conjunction $p(X) \wedge q(X)$ as an abstract conjunction in the $\mathcal{PD}$-domain with $p(a) \wedge q(a) \in \gamma_{inst}(p(X) \wedge q(X))$ as well as $p(X) \wedge q(X) \in \gamma_{inst}(p(X) \wedge q(X))$, but $p(a) \wedge q(b) \notin \gamma_{inst}(p(X) \wedge q(X))$.

Using the concrete conjunctions as abstract conjunctions is potentially confusing, which has probably obfuscated the relationship between partial deduction and abstract interpretation in the past.

5. ABSTRACT UNFOLDING AND RESOLUTION

Let us now try to remove one limitation of classical partial deduction in general and Procedure 1 in particular: its limitation to the $\mathcal{PD}$-domain. We will tackle the extension to conjunctive partial deduction later in Section 7, although in the exposition below we will (whenever there is no harm to clarity) keep the definitions as general as possible so as to simplify the move to conjunctive partial deduction.

The result of *resolve*(C, A) in Procedure 1 is actually the body of the resultant C generated by *unfold* for $P \cup \{\leftarrow A\}$. Now, a subtle, but important point is that the body of a resultant is thus used in two different ways: First, it is obviously part of the residual code. Second, it is used as an abstract conjunctions in the $\mathcal{PD}$-domain, representing all possible resolvents. In summary, the body of a resultant is not only used as a *concrete conjunction* within the residual code, it is also used as an *abstract* conjunction for a program analysis of the call set (to ensure that all possible calls are covered by the residual code).

In the more general setting we endeavor to develop, these two roles of the bodies of resultants have to be separated out (the residual program still has to be expressed in the concrete domain but we want to be able to use abstract domains different from the $\mathcal{PD}$-domain). This has already been prepared within Procedure 1 by using

the two functions *unfold* and *resolve*. All we have to do now, is to generalize these two functions. In other words, if we want to specialize an abstract atom $\mathbf{A}$ within a program P:

(1) we have to compute a set of resultants, to be denoted by $aunfold(P, \mathbf{A})$ which have to be "totally correct" for all possible calls in $\gamma(\mathbf{A})$, ensuring that no computed answers will be lost or added within the specialised program (we will make this more precise below).

(2) we have to compute, for each resultant C_i in $aunfold(P, \mathbf{A})$ an *abstract* conjunction $\mathbf{A_i}$, to be denoted by $aresolve(C_i, \mathbf{A})$, safely approximating all the possible resolvent goals which can occur after resolving an element of $\gamma(\mathbf{A})$ with C.

We will call step 1. *abstract unfolding* and step 2. *abstract resolution*, and will formally define these concepts in Definitions 5.3 and 5.4 below. For this we need a few auxiliary concepts.

First, we want to be able to formally define when the resultants produced by $aunfold(P, \mathbf{A})$ for a particular abstract conjunction $\mathbf{A}$ are correct, independently of how the rest of the specialised program looks like. In other words, we want a local correctness criterion, just considering the resultants generated for $\mathbf{A}$. The problem is that these resultants are incomplete; they will typically refer to other predicates defined somewhere else in the final specialised program P' and we cannot execute the resultants $aunfold(P, \mathbf{A})$ in isolation. We can, however, perform single resolution steps on these resultants. Suppose, e.g., that $\leftarrow p(X)$ resolves with a resultant $p(Z) \leftarrow q(Z) \in aunfold(P, \mathbf{A})$ giving us the resolvent $\leftarrow q(Z)$ and the *mgu* $\theta = \{X/Z\}$. We cannot view θ as a computed answer substitution for $P' \cup \{\leftarrow p(X)\}$, but we can view the pair $\langle q(Z), \theta \rangle$ as a *conditional answer* for $P' \cup \{\leftarrow p(X)\}$: if we manage to find a computed answer substitution σ for $P' \cup \{\leftarrow q(Z)\}$ then $\theta\sigma$ restricted to the variable X will be a computed answer substitution for $P' \cup \{\leftarrow p(X)\}$.

So, in order to reason about correctness of resultants individually, we need to show that the conditional answers obtained using $aunfold(P, \mathbf{A})$ can be put into a one-to-one correspondence with conditional answers of the original program. To be able to express this formally, we now define the concept of *conditional answers* as obtained from possibly incomplete SLD-trees in the original program and from resultants.

Definition 5.1 ($\rightsquigarrow_\tau$, $\rightsquigarrow_R$) Let P be a program and Q a conjunction. Given an SLD-tree τ for $P \cup \{\leftarrow Q\}$ we denote by $Q \rightsquigarrow_\tau \langle L, \theta \rangle$ the fact that a leaf goal $\leftarrow L$ of τ can be reached from Q via c.a.s. θ. $\langle L, \theta \rangle$ is also called a *conditional computed answer* for Q in P.

Given a resultant R and a conjunction Q we denote by $Q \rightsquigarrow_R \langle L, \theta \rangle$ the fact that $\theta = \theta' \downarrow_{vars(Q)}$, $L = B\theta'$ where $\theta' = mgu(Q, H)$, $H \leftarrow B$ is some variant of R which has no variables in common with Q, and $\theta' \downarrow_{vars(Q)}$ denotes the restriction of θ' to the variables in Q.

If Q and the head of R are atoms $Q \rightsquigarrow_R \langle L, \theta \rangle$ is equivalent to saying that $\leftarrow Q$ resolves with the clause R via c.a.s. θ yielding $\leftarrow L$ as resolvent. For example, $p(X, b) \rightsquigarrow_{p(a,Z) \leftarrow q(Z)} \langle q(b), \{X/a\} \rangle$. The above definition can also be applied if Q is a conjunction and R is a resultant which is not a clause. Take for example,

$R = p_1(a) \wedge p_2(Z) \leftarrow q(Z)$ and $Q = p_1(X) \wedge p_2(b)$. We then obtain $Q \rightsquigarrow_R \langle q(b), \{X/a\}\rangle$. This will be of relevance mainly when we consider conjunctive partial deduction later on. Intuitively this treatment does not introduce a new computation paradigm; it just corresponds to renaming conjunctions into atoms and general resultants into Horn clauses and then applying ordinary resolution. In the above example, if we rename Q into $Q' = p'(X, b)$ and R into $R' = p'(a, Z) \leftarrow q(Z)$ we obtain the same partial computed answer $Q' \rightsquigarrow_{R'} \langle q(b), \{X/a\}\rangle$.

Observe that $Q \rightsquigarrow_\tau \langle L, \theta\rangle$ implies that $\exists R \in \mathit{resultants}(\tau)$ such that $Q \rightsquigarrow_R \langle L, \theta\rangle$.

In order to define correctness criteria, we have to reason about equivalence of conditional computed answers and computed answer substitutions in the original program and in the residual program. However, substitutions (and renaming substitutions) within SLD-trees are notoriously difficult to handle (see [Ko and Nadel 1991] or [Doets 1993]), and proving identity of computed answer substitutions is often very tricky or impossible to achieve. To avoid these technical problems we introduce the following notion, characterizing when two conditional computed answers are equivalent (in the context of a particular goal Q).

Definition 5.2 ($\approx_Q$) Given three conjunctions Q, L, L' and two substitutions θ, θ' we say that $\langle L, \theta\rangle \approx_Q \langle L', \theta'\rangle$ iff $Q\theta \leftarrow L$ is a variant of $Q\theta' \leftarrow L'$.

For example, we have $\langle q(Z), \{X/Z\}\rangle \approx_{p(X)} \langle q(V), \{X/V, Z/V\}\rangle$ as $p(Z) \leftarrow q(Z)$ is a variant of $p(V) \leftarrow q(V)$.

We can now formalize the notion of abstract unfolding and resolution.

Definition 5.3 Let $(\mathcal{Q}, \gamma)$ be an abstract domain. An *abstract unfolding* operation *aunfold* for a program P and $(\mathcal{Q}, \gamma)$ maps abstract conjunctions to finite sets of resultants and has the property that for all $\mathbf{A} \in \mathcal{AQ}$ and $Q \in \gamma(\mathbf{A})$ there exists a non-trivial SLD-tree τ for $P \cup \{\leftarrow Q\}$ such that:

$$Q \rightsquigarrow_\tau s_1 \quad \Rightarrow \quad \exists C_i \in \mathit{aunfold}(P, \mathbf{A}) \mid Q \rightsquigarrow_{C_i} s_2 \ \wedge \ s_1 \approx_Q s_2 \tag{1}$$

$$Q \rightsquigarrow_{C_i} s_2 \ \wedge \ C_i \in \mathit{aunfold}(P, \mathbf{A}) \quad \Rightarrow \quad \exists s_1 \mid Q \rightsquigarrow_\tau s_1 \ \wedge \ s_1 \approx_Q s_2 \tag{2}$$

Point 1 requests that the code generated by *aunfold* is *complete* in the sense that every conditional computed answer s_1 can be reproduced by at least one of the resultants in $\mathit{aunfold}(P, \mathbf{A})$. Point 2 additionally requests *soundness* (as we want to have residual code which is *totally correct* and not just a safe approximation), in the sense that every conditional computed answer s_2 can be achieved within the original program as well. Together, Points 1 and 2, thus express that there must be a *one-to-one correspondence* between conditional computed answers in the original program and the resultants $\mathit{aunfold}(P, \mathbf{A})$. Some of these points are illustrated in Figure 2 below (where $s_1 = \langle L, \theta\rangle$ and $s_2 = \langle L', \theta'\rangle$).

Definition 5.4 Let $(\mathcal{Q}, \gamma)$ be an abstract domain. An *abstract resolution* operation *aresolve* for $(\mathcal{Q}, \gamma)$ maps abstract conjunctions and concrete resultants to abstract conjunctions such that for all $\mathbf{A} \in \mathcal{AQ}$, $C_i \in \mathit{aunfold}(P, \mathbf{A})$, and $Q \in \gamma(\mathbf{A})$:

$$Q \rightsquigarrow_{C_i} \langle L', \theta'\rangle \quad \Rightarrow \quad L' \in \gamma(\mathit{aresolve}(\mathbf{A}, C_i)) \tag{3}$$

Point 3 requires that $\mathbf{A_i} = \mathit{aresolve}(\mathbf{A}, C_i)$ is a safe approximation of the possible resolvents of C_i, in the sense that every possible resolvent of $Q \in \gamma(\mathbf{A})$ with C_i is a concretisation of $\mathbf{A_i}$ (but not necessarily vice-versa).

Unless explicitly stating otherwise, we suppose that the abstract unfolding *aunfold* and abstract resolution operators *aresolve*, along with the abstract domain $(\mathcal{Q}, \gamma)$, are fixed.

How to construct abstract unfoldings. aresolve is thus basically a safe approximation of a resolution step, and we can thus develop *aresolve* by reusing abstract interpretation techniques. We will thus not discuss this issue in much detail here, but refer the reader to the abstract interpretation literature.

The development of a correct abstract unfolding operation is another issue, and is not something that can be found within the abstract interpretation literature.

Note that the definition of *aunfold* does not stipulate how the resultants are to be obtained; it just describes how a "correct" set of resultants should look like. In particular, in contrast to classical partial deduction, the resultants do *not* necessarily have to be extracted from SLD-trees. In classical partial deduction, we have $\mathit{aunfold}(P, A) = \mathit{resultants}(\tau')$ where τ' is an SLD-tree for $P \cup \{\leftarrow A\}$, and the conditions of Definition 5.3 are thus trivially met (we have to choose as τ for $P \cup \{\leftarrow Q\}$ and "adapted" version of τ' where some branches may be removed as Q is an instance of A).

Many unfolding techniques have been developed in the context of classical partial deduction. Issues for concern are [Leuschel and Bruynooghe 2002]: termination (i.e., building finite SLD-trees), achieving good specialization and avoiding slowdowns. To ensure termination, well-founded measures [Bruynooghe et al. 1992; Martens and De Schreye 1996] and well-quasi-orders can be used [Sahlin 1993; Bol 1993]. The well-quasi orders based on the homeomorphic embedding relation [Sørensen and Glück 1995; Leuschel 1998a] have recently been very popular. To avoid slowdowns, determinacy [Gallagher and Bruynooghe 1991; Gallagher 1991], only selecting atoms that unify with a single clause head, has been successful. The strategy can be refined with a so-called "look-ahead" to detect failure at a deeper level. We refer the interested reader to [Leuschel and Bruynooghe 2002] for a recent survey of these techniques.

For abstract partial deduction, we can always do a similar thing: given $\mathbf{A}$ chose a concrete dominator A of $\mathbf{A}$ (cf., Point 2 of Definition 4.1), construct an SLD-tree τ for $P \cup \{\leftarrow A\}$ and simply set $\mathit{aunfold}(P, \mathbf{A}) = \mathit{resultants}(\tau)$. This always satisfies

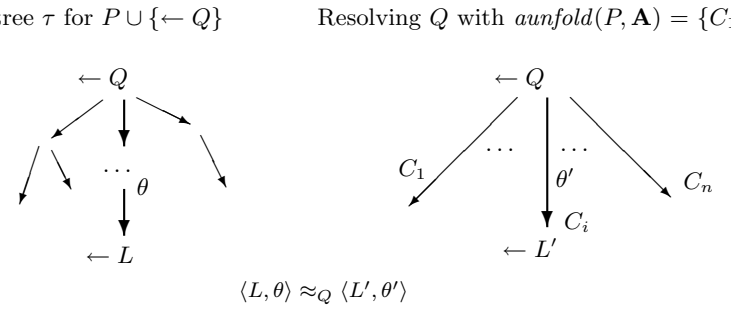

Fig. 2. One-to-one correspondence of conditional computed answers for abstract unfolding

Definition 5.3. The following example illustrates this on the $\mathcal{PD}$-domain.

Example 5.5 Let P be the following program checking equality of lists:

$eq([],[]) \leftarrow$

$eq([H|X],[H|Y]) \leftarrow eq(X,Y)$

Let $\mathbf{A} = eq([a|T],Z)$ in the $\mathcal{PD}$-domain and let τ be the SLD-tree depicted in Figure 3 for $P \cup \{\leftarrow eq([a|T],Z)\}$ (i.e., we use $\mathbf{A}$ as a concrete dominator of itself). Let us perform abstract unfolding in a classical manner, by taking the resultants of τ:

- $aunfold(P,\mathbf{A}) = resultants(\tau) = \{C_1\}$, where $C_1 = eq([a|X],[a|Y]) \leftarrow eq(X,Y)$,
- $aresolve(\mathbf{A},C_1) = eq(X,Y)$

These two definitions satisfy all points of Definitions 5.3 and 5.4 for $\mathbf{A}$. For example, let us examine the 2 concretisations $A_1 = eq([a],[b]) \in \gamma_{inst}(\mathbf{A})$and $A_2 = eq([a,b],Y) \in \gamma_{inst}(\mathbf{A})$ of $\mathbf{A}$. Figure 3 shows that for each of those we can construct SLD-trees which satisfy Definition 5.3. For example, A_1 has a failed SLD-tree and A_1 does not unify with the head $eq([a|X],[a|Y])$ of C_1 either. We thus trivially have the required one-to-one correspondence of conditional answers (and satisfy Definition 5.4 as well). For A_3 we have $A_3 \rightsquigarrow_{C_1} \langle eq([b],Y'),\{Y/[a|Y']\}\rangle$ and $A_3 \rightsquigarrow_{\tau_3} \langle eq([b],Y''),\{Y/[a|Y'']\}\rangle$

We have $\langle eq([b],Y'),\{Y/[a|Y']\}\rangle \approx_{A_3} \langle eq([b],Y''),\{Y/[a|Y'']\}\rangle$ and thus again the required one-to-one correspondence.

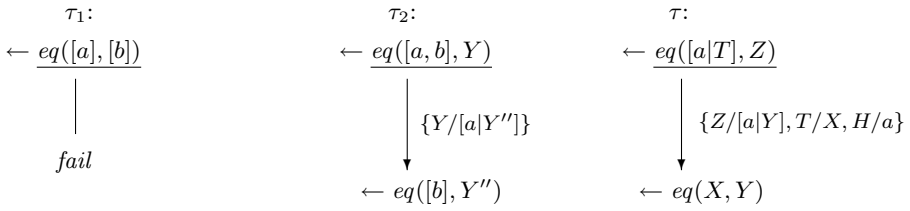

Fig. 3. SLD-trees for Example 5.5

While computing *aunfold* by taking the resultants from SLD-trees of concrete dominators is correct, it does not yet make much use of the information within $\mathbf{A}$. One can use the information within $\mathbf{A}$ to further instantiate those resultants; inspired by the more specific resolution steps [Gallagher 1991] or the most specific versions of [Marriott et al. 1988; 1990]. For example, replacing C_1 in Example 5.5 by $eq([H|X],[H|Y]) \leftarrow eq(X,Y)$ is also correct. Also, even replacing C_1 by $eq([Z|X],[a|Y]) \leftarrow eq(X,Y)$ is still correct. But note that this resultant is no longer sound for calls which are not concretisations of $\mathbf{A}$ (e.g., the call $\leftarrow eq([b],[a])$ yields a conditional computed answer $\langle eq([],[]),\{\}\rangle$ which cannot be matched by the original program). We will return to this issue in Section 10.

One further possible improvement, is to remove from $resultants(\tau)$ all those resultants $A\theta \leftarrow B$ which, although they resolve with A, cannot resolve with any concretisation of $\mathbf{A}$. This again, always satisfies Definition 5.3, as the following proposition shows.

Proposition 5.6 Let $\mathbf{Q}$ be an abstract conjunction and let Q be a concrete dom-

inator for $\mathbf{Q}$. Let τ be a SLD-tree for $P \cup \{\leftarrow Q\}$ and let $R \subseteq \mathit{resultants}(\tau)$ be a set of resultants such that for all resultants $Q\theta \leftarrow B \in (\mathit{resultants}(\tau) \setminus R)$ we have that no instance of $Q\theta$ is in $\gamma(\mathbf{Q})$. Then $\mathit{aunfold}(P, \mathbf{Q}) = R$ satisfies Definition 5.3.

PROOF. (Sketch) Let us first assume that $R = \mathit{resultants}(\tau)$, i.e., $\mathit{aunfold}(P,\mathbf{Q}) = \{Q\theta_1 \leftarrow B_1, \ldots, Q\theta_k \leftarrow B_k\}$ are the resultants of a finite SLD-tree τ_Q for $P \cup \{\leftarrow Q\}$. Now take $Q\sigma \in \gamma(Q)$ and build the SLD-tree τ for $P \cup \{\leftarrow Q\sigma\}$ according to τ_Q (i.e., selecting the same literals, to the same depth; some branches might be missing in τ because of failed unifications). All the requirements of Definitions 5.3 and 7.1 are met:

- Point 1: This is a direct corollary of Lemma 4.12 in [Lloyd and Shepherdson 1991].
- Point 2: This is a direct corollary of Lemma 4.9 in [Lloyd and Shepherdson 1991] (cf., proof of Lemma 8.3 for more details).
- Point 4: Take $Q' = Q$. This will unify with all $Q\theta_i$ via *mgu* σ and we thus have $Q \rightsquigarrow_{C_i} \langle B_i\sigma, \sigma\rangle$.

body trivially satisfies Definition 5.4: if some $Q\gamma$ resolves with H via mgu θ we get the resolvent $B\theta$ which is a concretisation of B.

Now, if $R \subset \mathit{resultants}(\tau)$ we only have to re-check Point 1. We can deduce that the head H of every resultant $C \in (\mathit{resultants}(\tau) \setminus R)$ does not unify with $Q\sigma$, because any instance of H is not in $\gamma(\mathbf{Q})$ while any instance of $Q\sigma$ is. Hence, again by Lemma 4.12 in [Lloyd and Shepherdson 1991] we can deduce that the branch corresponding to C in τ is finitely failed. □

The following simple example illustrates this possibility. (Note that we denote by □ the empty goal as well as the empty conjunction.)

Example 5.7 Let P be the following program:

(C_1) $p(a) \leftarrow$
(C_2) $p(f(X)) \leftarrow p(X)$
(C_3) $p(g(X)) \leftarrow p(X)$

Let $\mathbf{A}$ be an abstract atom within some abstract domain $(\mathcal{Q}, \gamma)$ such that $\gamma(\mathbf{A}) = \{p(a), p(g(a)), p(g(g(a))), \ldots\}$. Then $\mathit{aunfold}(P, \mathbf{A}) = \{C_1, C_3\}$, $\mathit{aresolve}(\mathbf{A}, C_1) =$ □ and $\mathit{aresolve}(\mathbf{A}, C_3) = \mathbf{A}$ is correct wrt Definitions 5.3 and 5.4. We were thus able to safely remove the redundant clause C_2, in they style of [de Waal and Gallagher 1991; Gallagher and de Waal 1992; de Waal and Gallagher 1994] (which detects and removes redundant clauses as a post-processing).

[Gallagher and Peralta 2000; 2001] and [Leuschel and Gruner 2001] show how such abstract unfoldings can be developed for a particular abstract domain based upon regular types. [Leuschel and Gruner 2001] also shows how resultants can be instantiated using the regular type information.

But even more exotic abstract unfoldings are possible. Suppose for example that the computed instances of some concrete dominator A of $\mathbf{A}$ are a superset of $\gamma(\mathbf{A})$. One can then just create a single fact for $\mathit{aunfold}(P, \mathbf{A})$; e.g., if $A = p(f(X), Z)$ simply produce $\mathit{aunfold}(P, \mathbf{A}) = \{p(X, Y) \leftarrow\}$.

Observe, that in Definition 5.3 above, nothing forces one to use the *same* structure (i.e. same selected literal positions, same clauses) for *all* the concretisations of $\mathbf{A}$. Indeed, this enables some very powerful optimizations not achievable within existing "classical" specialization frameworks. For instance, in the example below we are able to completely eliminate a type-like check from the residual program.

Example 5.8 Let P be the program from Example 5.5 and $\mathbf{A}$ be the set of all calls $eq(t,t)$ where t is a bounded list, i.e, a list whose skeleton is fixed but whose individual elements can be variables or contain variables. For example, $eq([],[])$ and $eq([X],[X])$ are in $\gamma(\mathbf{A})$ but not $eq([],[a])$ nor $eq([X|T],[X|T])$. This can obviously not be represented in the $\mathcal{PD}$-domain.
Then $aunfold(P,\mathbf{A}) = C_1 = \{eq(X,Y) \leftarrow\}$ and $aresolve(\mathbf{A},C_1) = \Box$ are correct according to the above definition! Take the concretisations $A_1 = eq([],[])$ and $A_2 = eq([a],[a])$. We have $A_1 \leadsto_{C_1} \langle\Box,\{\}\rangle$ and $A_2 \leadsto_{C_1} \langle\Box,\{\}\rangle$ As can be seen in Figure 4 we can produce for each of them an SLD-tree (with a different structure) which satisfies Definitions 5.3 and 5.4.
One can thus generate the residual program:

$eq(X,Y) \leftarrow$

Observe that this residual code is only sound for concretisations of $\mathbf{A}$ but not, e.g., for the call $eq(a,[])$.

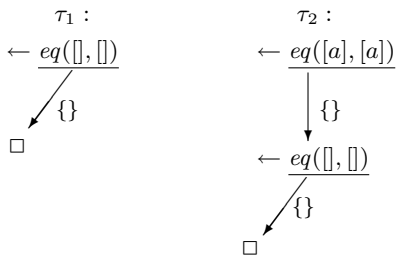

Fig. 4. SLD-trees for Example 5.8

To our knowledge, these powerful optimizations are not possible within existing partial deduction or partial evaluation techniques. It is related to the notion of abstract executability used in [Puebla and Hermenegildo 1995; 1996; 1999]. In practice, such optimizations can be very useful and have already been implemented, e.g., in the static assertion checker of the Ciao Prolog preprocessor [Puebla et al. 2000b; 2000a].

One can extend this approach to cover built-ins as well. E.g., if we know that a given variable X represents an integer we can, e.g., specialize both $atomic(X)$ or $number(X)$ into *true*. One can imagine various other optimizations not possible in conventional techniques based upon the $\mathcal{PD}$-domain, like specializing *arg* or *functor* calls based upon type information of the arguments. A similar idea has been used in [Puebla and Hermenegildo 1996; 1999] to remove redundant tests and calls to builtins from the residual program which analysis information allows abstractly executing to true, false, or error. This technique has been applied to optimizing automatically parallelized programs.

In summary, we believe that our framework is very general, and has the potential to cover many new, specialization techniques. While it is still far from trivial to develop those, proving the correctness of such new specialization methods should now be much easier.

6. ATOMIC ABSTRACT PARTIAL DEDUCTION

The definition of an abstract partial deduction is now very straightforward:

Definition 6.1 (abstract atomic partial deduction) Let P be a program, $\mathcal{A}$ a set of abstract atoms and *aunfold* is an abstract unfolding rule. We then define *the abstract atomic partial deduction of P wrt $\mathcal{A}$ and aunfold* to be the program P' = $\{C \mid C \in \mathit{aunfold}(P, \mathbf{A}) \wedge\ \mathbf{A} \in \mathcal{A}\}$. We also call P' *an abstract atomic partial deduction of P wrt $\mathcal{A}$.*

6.1 Correctness of Atomic Abstract Partial Deduction

If we have an abstract unfolding *aunfold* at our disposal, all we have to figure out is which set $\mathcal{A}$ of abstract atoms should we use in the above definition, so as to obtain a correct partial deduction. What we need is the abstract counterpart of the $\mathcal{A}$-closedness condition in Theorem 2.7. In other words, we have to find a condition which ensures that every possible call R that can occur when running the residual program is covered by an appropriate abstract atom $\mathbf{A} \in \mathcal{A}$ such that $R \in \gamma(\mathbf{A})$. In Section 3.3 we have seen that the $\mathcal{A}$-closedness of classical partial deduction could be reformulated as $\mathcal{A}$ being a fixpoint of the operator R_P^α, which is a safe approximation of the concrete operator R_P computing subgoals and calls. We will use that approach here.

We build upon *aunfold* and *aresolve* to extend the R_P^α operator from Section 3.3 into an operator R_P^A mapping sets of abstract conjunctions to sets of abstract conjunctions in the following way:

$$R_P^A(S) = S \cup \{\mathit{aresolve}(\mathbf{A}, C) \mid \mathbf{A} \in S \wedge C \in \mathit{aunfold}(P, \mathbf{A})\}$$

Intuitively, $R_P^A(\mathcal{A})$ is a safe approximation of all resolvents that can arise after a single resolution step of a concretisation of $\mathcal{A}$ with a clause in the atomic partial deduction of P wrt $\mathcal{A}$ using *aunfold*.

We could now say that we have $\mathcal{A}$-closedness for abstract partial deductions iff $\gamma(R_P^A(\mathcal{A})) \subseteq \gamma^\wedge(\mathcal{A})$, where, as in Section 3.3 we extend the concretisation function γ into $\gamma^\wedge(S) = \{Q_1 \wedge \ldots \wedge Q_n \mid Q_i \in \gamma(S)\}$ so as to take into account that conjunctions can be split up by partial deduction.

From an abstract interpretation perspective this is sufficient, as it would ensure that $\mathcal{A}$ covers all possible subgoals that can occur when executing any concretisation of $\mathcal{A}$ using the partial deduction of P wrt $\mathcal{A}$ and *aunfold*. However, it is a bit too liberal in a partial deduction setting as it would allow the concretisations of a single abstract atom or conjunction within $R_P^A(\mathcal{A})$ to be covered by several abstract atoms within $\mathcal{A}$. This would cause problems when applying a renaming transformation which, as we have seen at the end of Section 2, helps overcome the "independence" condition, improves performance, and is unavoidable for conjunctive partial deduction. Suppose, for example, that $\mathcal{A} = \{\mathbf{A_1}, \mathbf{A_2}\}$, $R_P^A(\mathcal{A}) = \{\mathbf{A_1}\}$, with $\mathit{aunfold}(P, \mathbf{A_1}) = \{p(f(X)) \leftarrow p(X)\}$ and $\mathit{aunfold}(P, \mathbf{A_2}) = \{p(g(X)) \leftarrow\}$ and that $\gamma(\mathbf{A_1}) \subseteq \gamma(\mathbf{A_2}) \cup \gamma(\mathbf{A_3})$ while $\gamma(\mathbf{A_1}) \not\subseteq \gamma(\mathbf{A_2})$ and $\gamma(\mathbf{A_1}) \not\subseteq \gamma(\mathbf{A_3})$. We do have $\gamma(\mathcal{A}) = \gamma(R_P^A(\mathcal{A}))$ but it would be impossible to perform a renaming transformation in the classical sense, as we cannot decide whether the call $p(X)$ within $\mathit{aunfold}(P, \mathbf{A_1})$ should be mapped to the renamed version of $\mathbf{A}_1$ or $\mathbf{A}_2$.

In order to circumvent these problems, we introduce the following concepts.

Definition 6.2 Let $(\mathcal{AQ}, \gamma)$ be an abstract domain. First, we extend γ to sequences of abstract conjunctions by defining

$$\gamma(\langle \mathbf{Q_1}, \ldots, \mathbf{Q_n} \rangle) = \{Q_1 \wedge \ldots \wedge Q_n \mid 1 \leq i \leq n \Rightarrow Q_i \in \gamma(\mathbf{Q}_i)\}$$

Let $\mathcal{A}$ be a set of abstract conjunctions. We say that an abstract conjunction $\mathbf{Q}$ is *covered by* $\mathcal{A}$ iff there exists a sequence $\langle \mathbf{Q_1}, \ldots, \mathbf{Q_n} \rangle$ of abstract conjunctions such that $\forall 1 \leq i \leq n$ we have $\mathbf{Q_i} \in \mathcal{A}$ and $\gamma(\mathbf{Q}) \subseteq \gamma(\langle \mathbf{Q_1}, \ldots, \mathbf{Q_n} \rangle)$. A set $\mathcal{A}'$ of abstract conjunctions is *covered by* $\mathcal{A}$ iff every element of $\mathcal{A}'$ is covered by $\mathcal{A}$.

For example, in the $\mathcal{PD}$-domain, both $p(a) \wedge q(a) \wedge p(b)$ and $p(b) \wedge p(a) \wedge q(a) \wedge p(c) \wedge q(c)$ are covered by $\{p(X) \wedge q(X), p(b)\}$ but not $p(a)$ nor $p(a) \wedge p(b) \wedge q(a)$. Here it is of relevance that we treat $\wedge$ as associative, but not as commutative nor idempotent.

We can now define the abstract version of the $\mathcal{A}$-closedness condition, which ensures that renaming can always be performed. We also define the abstract version of the independence condition from Definition 2.6 and Theorem 2.7.

Definition 6.3 We say that a set $\mathcal{A}$ of abstract conjunctions is *covered* wrt P and *aunfold* iff $R^A_P(\mathcal{A})$ is covered by $\mathcal{A}$.

We say that $\mathcal{A}$ is *independent* iff $\forall \mathbf{A}_1, \mathbf{A}_2 \in \mathcal{A}$ with $\mathbf{A}_1 \neq \mathbf{A}_2$ we have $\gamma(\mathbf{A}_1) \cap \gamma(\mathbf{A}_2) = \emptyset$.

We need one more definition before formulating our first correctness theorem.

Definition 6.4 Given two expressions L and L', we write $L \approx L'$ to denote that L is a variant of L.

Theorem 6.5 Let P' be an abstract atomic partial deduction of P wrt an independent set of abstract atoms $\mathcal{A}$. Let $\mathcal{A}$ be covered wrt P and *aunfold* and let $Q \in \gamma(\mathcal{A})$. Then

(1) If $P \cup \{\leftarrow Q\}$ has an SLD-refutation with computed answer θ then $P' \cup \{\leftarrow Q\}$ has an SLD-refutation with computed answer θ' such that $Q\theta \approx Q\theta'$.
(2) If $P' \cup \{\leftarrow Q\}$ has an SLD-refutation with computed answer θ' then $P \cup \{\leftarrow Q\}$ has an SLD-refutation with computed answer θ such that $Q\theta \approx Q\theta'$.
(3) If $P' \cup \{\leftarrow Q\}$ has a finitely-failed SLD-tree then so does $P \cup \{\leftarrow Q\}$.
(4) If $P \cup \{\leftarrow Q\}$ has a finitely-failed SLD-tree then so does $P' \cup \{\leftarrow Q\}$.

This theorem is a special case of the Theorems 8.2 and 8.7 which we present and prove later.

6.2 A Generic Procedure for Abstract Partial Deduction

We now define a generalisation operator for abstract conjunctions, suitable for our framework:

Definition 6.6 A *generalisation operator* is a function[6] $ageneralize : 2^{\mathcal{AQ}} \mapsto 2^{\mathcal{AQ}}$ such that $\mathcal{A}$ is covered by $ageneralize(\mathcal{A}))$ for all $\mathcal{A} \in 2^{\mathcal{AQ}}$.

A generalisation operator is called *atomic* if for every $S \in 2^{\mathcal{AQ}}$, $ageneralize(S)$ is a set of abstract atoms.

[6]It is of course possible to give extra parameters to *ageneralize*, e.g., so that it can take the specialization history into account.

An atomic generalisation operator thus embodies the functions of both *split* and *generalize* from Section 3.3. If $\mathcal{A}$ is a fixpoint of $U(S) = \mathit{ageneralize}(R^A_P(S))$ then this ensures that $\mathcal{A}$ is covered.

Based upon the notions introduced above, we can now present a generic procedure for top-down program specialization, which tries to find such fixpoints, in a very concise manner:

PROCEDURE 2. **(Abstract Partial Deduction)**
Input: A program P and an abstract conjunction $\mathbf{A}$
Output: A specialised program P'
Initialize: $i = 0$, $\mathcal{A}_0 = \{\mathbf{A}\}$
repeat
let $\mathcal{A}_{i+1} := \mathit{ageneralize}(R^A_P(\mathcal{A}_i))$; **let** $i := i + 1$;
until $\mathcal{A}_{i-1} = \mathcal{A}_i$
Let P' be an abstract partial deduction wrt $\mathcal{A}_i$

It is obvious that if the above algorithm terminates, $\mathcal{A}_i$ is covered and hence, e.g., Theorem 8.2 can be applied. By combining widening operators from the abstract interpretation literature with generalisation operators from the partial deduction literature, it is now possible to ensure termination of this procedure.

One of the earliest [Martens et al. 1994] widenings for partial deduction for the $\mathcal{PD}$-domain was based on the *most specific generalisation* or *least general generalisation* of a finite set of expressions E, denoted by $\mathit{msg}(E)$, is the most specific expression M such that all expressions in E are instances of M. The *msg* can be effectively computed [Lassez et al. 1988] and given an expression A, there are no infinite chains of strictly more general expressions [Huet 1980]. More refined widenings, are based upon well-founded orders, well-quasi orders and characteristic trees (see, e.g, [Gallagher and Bruynooghe 1991; Leuschel et al. 1998; Leuschel 1998a], see also [Leuschel and Bruynooghe 2002]).

[Gallagher and Peralta 2000; 2001] and [Leuschel and Gruner 2001] present non-trivial generalisation operators for abstract domains based upon regular types.

7. CONJUNCTIVE ABSTRACT PARTIAL DEDUCTION

Classical partial deduction, as defined in Definition 2.4 specializes a *set of atoms* $\mathcal{A}$. Even though conjunctions of atoms may appear within the SLD-trees constructed for these atoms, only atoms are allowed to appear within $\mathcal{A}$. A similar picture holds for atomic abstract partial deduction, introduced in the previous Section 6, where only abstract atoms are allowed to appear within $\mathcal{A}$ of Definition 6.1. In other words, when we stop unfolding, every conjunction at the leaf is automatically split into its atomic constituents which are then specialised (and possibly further abstracted) separately. This restriction often considerably restricts the potential power of partial deduction, e.g., preventing the elimination of unnecessary variables [Proietti and Pettorossi 1991] (also called deforestation and tupling).

To overcome this limitation in the setting of classical partial deduction, [De Schreye et al. 1999] presents a relatively small extension of partial deduction, called *conjunctive partial deduction*. This technique extends the standard partial deduction approach by considering sets $S = \{C_1, \ldots, C_n\}$ where the elements C_i are now *conjunctions* of atoms instead of just single atoms. Conjunctive partial deduction

also solves a dilemma of classical partial deduction related to efficiency and precision and makes the local control much easier (see, e.g., [Leuschel and Bruynooghe 2002]).

All the definitions related to the abstract unfolding and abstract resolution operations (5.1, 5.2, 5.3, 5.4) already cater for abstract conjunctions. Definitions 6.6 and 6.3 also already cater for sets of abstract conjunctions. Thus, to perform conjunctive partial deduction using Procedure 2 we just have to remove the restriction that *ageneralize* is atomic. Of course, this raises a new termination problem: in addition to having to worry about infinitely many atomic atoms *ageneralize* now also has to worry about an infinite number of growing abstract conjunctions. In other words, the generalisation operation *ageneralize* has to be more refined. It has been well studied how to devise such generalisation operators for the $\mathcal{PD}$-domain [Glück et al. 1996; De Schreye et al. 1999]. For abstract conjunctive partial deduction, this has to be combined with widenings from the abstract interpretation literature. [Leuschel and Gruner 2001] shows how to do this for an abstract domain based upon regular types.

There is also the issue of code generation which becomes more involved. Indeed, the resultants $C = H_i \leftarrow B_i$ in Definition 6.1 are not necessarily Horn clauses (because H_i can be a conjunction). To transform such resultants back into standard clauses, conjunctive partial deduction [De Schreye et al. 1999] employs a *renaming* transformation, from conjunctions to atoms, which practical partial deduction systems already perform anyway. We will do the same here, and present the full details in Section 7.1.

7.1 Generating Residual Code for Conjunctive Partial Deduction

All that is missing to present a generic abstract specialization algorithm is a way of generating executable residual code from the resultants $H_i \leftarrow B_i$ produced by the abstract unfolding. For this we have to transform the resultants into Horn clauses. This can be achieved by mapping the abstract conjunctions produced by the flow analysis to concrete atoms and then appropriately renaming the heads H_i and the bodies B_i.

Definition 7.1 An *abstract unfolding* operation *aunfold* is said to have the *no-garbage property* iff the following equation holds:

$$\forall \mathbf{A} \in \mathcal{AQ} \;\; \forall R \in \mathit{aunfold}(P, \mathbf{A}) : \quad \exists s \exists Q' \in \gamma(\mathbf{A}) \mid Q' \rightsquigarrow_R s \tag{4}$$

This property prevents *aunfold* from producing garbage resultants which unify with no concretisation. From now on we suppose that all abstract unfolding operations satisfy this property. This obvious requirement will simplify the code generation but it is not strictly necessary.

Before formalizing the whole renaming process, let us first examine on a simple example how it can be achieved.

Example 7.2 Suppose we have the set $\mathcal{A} = \{A_1, A_2\}$ of abstract conjunctions in the $\mathcal{PD}$-domain with $A_1 = p(a, X)$ and $A_2 = p(b, Z) \wedge p(Z, d)$. Suppose that a resultant for A_2 is

$$p(b, c) \wedge p(c, d) \leftarrow p(a, b) \wedge p(b, e) \wedge p(e, d)$$

In order to translate this resultant into a Horn clause we have to rename all concretisations of A_2 to atoms. For this we can chose an atom, say $pp(Z)$, which contains all the variables in A_2 (viewed as a concrete conjunction). Now we can rename the head of the resultant into $pp(c)$ by instantiating Z to the proper value. We now have a Horn clause, but we still have to rename the body so that its conjunctions are renamed to call the proper residual predicates. For this we split up the body into subconjunctions $p(a,b)$, $p(b,e) \wedge p(e,d)$ so that each subconjunction is a concretisation of an element in $\mathcal{A}$. We can now rename each subconjunction to obtain:

$$pp(c) \leftarrow p(a,b) \wedge pp(e)$$

In the above example we had to chose an atom ($pp(Z)$) with the same variables as the abstract conjunction A_2 viewed as a concrete conjunction. Now, in general, an abstract conjunction cannot be viewed as a concrete conjunction. Hence we introduce the following concept which allows us to derive for every abstract conjunction a concrete one which covers all its concretisations.

Definition 7.3 Recall that a *concrete dominator* of an abstract conjunction $\mathbf{A}$ is a concrete conjunction Q such that all $Q' \in \gamma(\mathbf{A})$ are instances of Q. A *skeleton* for an abstract conjunction $\mathbf{A}$ is a maximally general concrete dominator of $\mathbf{A}$.

A skeleton for A_2 in Example 7.2 is $p(X_1,X_2) \wedge p(X_3,X_4)$. By Definition 4.1 of abstract domains we know that a concrete dominator (and thus skeleton) exists for all abstract conjunctions.[7] By $\lceil \mathbf{A} \rceil$ we denote some skeleton for $\mathbf{A}$.

Definition 7.4 An *atomic renaming* ρ for a set of abstract conjunctions $\mathcal{A}$ returns for every $\mathbf{A} \in \mathcal{A}$ an atom A, denoted by $\rho_{\mathbf{A}}$, such that $vars(\lceil \mathbf{A} \rceil) = vars(A)$. Also, for any $Q \preceq \lceil \mathbf{A} \rceil$ we define $\rho_{\mathbf{A}}(Q) = A\theta$ where θ is such that $Q = \lceil \mathbf{A} \rceil \theta$.

For $A_2 = p(b,Z) \wedge p(Z,d)$, of Example 7.2 we might have $\lceil A_2 \rceil = p(X_1,X_2) \wedge p(X_3,X_4)$, $\rho_{\mathbf{A_2}} = pp(X_1,X_2,X_3,X_4)$. For $Q = p(b,c) \wedge p(c,d)$ we then have $\rho_{\mathbf{A_2}}(Q) = pp(b,c,c,d)$.

Observe that for all $Q \preceq \lceil \mathbf{A} \rceil$ we have $\rho_{\mathbf{A}}(Q\theta) = \rho_{\mathbf{A}}(Q)\theta$, $vars(Q) = vars(\rho_{\mathbf{A}}(Q))$, and for all $Q' \preceq \lceil \mathbf{A} \rceil$ we can also assume that $mgu(Q,Q') = mgu(\rho_{\mathbf{A}}(Q), \rho_{\mathbf{A}}(Q'))$ (see Lemma 8.5). Also, to avoid name clashes, we will always suppose that for any $\mathbf{A} \neq \mathbf{A}'$ the predicate symbols used by $\rho_{\mathbf{A}}$ and $\rho_{\mathbf{A}'}$ are different.

Given a resultant $H_i \leftarrow B_i \in aunfold(P,\mathbf{A})$ we can now produce an actual Horn clause by renaming H_i and B_i. Renaming H_i is easy: we just calculate $\rho_{\mathbf{A}}(H_i)$ (which is always defined as $H_i \preceq \lceil \mathbf{A} \rceil$ by the Point 4 of Definition 7.1 of *aunfold*). If our flow analysis also contains $\mathbf{A_i} = aresolve(\mathbf{A}, H_i \leftarrow B_i)$ (and thus code for $\mathbf{A_i}$ will be generated) then renaming B_i is just as easy: we just calculate $\rho_{\mathbf{A_i}}(B_i)$. However, suppose that we have used generalisation and that we actually did not specialise $\mathbf{A_i}$ itself but rather the abstract conjunctions $\mathbf{G_1}, \ldots, \mathbf{G_n}$ such that $\mathbf{A_i}$ is covered by $\langle \mathbf{G_1}, \ldots, \mathbf{G_n} \rangle$ (just like in Example 7.2). In that case B_i has to be

[7] There actually also exists a most specific concrete dominator (by existence of a most specific generalisation *msg* of two terms [Lassez et al. 1988] and the fact that the strictly more general relation is a well-founded order [Huet 1980], i.e., the *msg* of all elements in $\gamma(\mathbf{A})$ exists). In the $\mathcal{PD}$-domain this is the conjunction itself (viewed as a concrete conjunction).

split up and then renamed using the renaming functions of the abstraction. We thus extend our atomic renaming function so that it accomplishes this:

Definition 7.5 Given a concrete conjunction B, an abstract conjunction $\mathbf{A}$, and a set $\mathcal{A}$ of abstract conjunctions we define:

$$\rho_{\mathcal{A},\mathbf{A}}(B) = \rho_{\mathbf{G_1}}(B_1) \wedge \ldots \wedge \rho_{\mathbf{G_n}}(B_n)$$

where $\mathbf{A}$ is covered by $\langle \mathbf{G_1}, \ldots, \mathbf{G_n} \rangle$ and $B = B_1 \wedge \ldots \wedge B_n$ is one possible way to split up B such that $\mathbf{G_i} \in \mathcal{A}$ and $B_i \preceq \lceil \mathbf{G_i} \rceil$ If no such partitioning exists then we leave $\rho_{\mathcal{A},\mathbf{A}}(B)$ undefined.

Note, by Point 4 of Definition 7.1, we know that if we can find a sequence $\langle \mathbf{G_1}, \ldots, \mathbf{G_n} \rangle$ which covers $\mathbf{A}$, then we can also find a partitioning of B such that $B_i \preceq \lceil \mathbf{G_i} \rceil$. Also observe that Definition 6.2 of the "covers concept" and the fact that we do not consider $\wedge$ commutative, imply that we not allow re-ordering of conjunctions within B.[8] It would be, however, relatively straightforward to do so. One just has to be careful to use the *same* reordering for *all* concretisations of $\mathbf{A}$ (otherwise it will be impossible to synchronize the code generation with the abstract resolution).

We can now define how to map resultants to Horn clauses so as to construct abstract partial deductions:

Definition 7.6 (abstract partial deduction) Let $\mathcal{A}$ be a covered set of abstract conjunctions. We then define an *abstract partial deduction of P wrt $\mathcal{A}$* to be the set of clauses:
$\{\rho_{\mathbf{A}}(H) \leftarrow \rho_{\mathcal{A},\mathbf{A}'}(B) \mid H \leftarrow B \in \mathit{aunfold}(P, \mathbf{A}) \wedge \mathbf{A}' = \mathit{aresolve}(\mathbf{A}, H \leftarrow B) \wedge \mathbf{A} \in \mathcal{A}\}$.

It is easy to see that, because $\mathcal{A}$ is covered, the renamings of the bodies B will always be defined.

Observe that, a skeleton always has distinct variables as its only terms. In other words, contrary to Example 7.2, we perform no filtering (i.e. $p(f(a))$ might get renamed into $p'(f(a))$ but never into $p'(a)$ or p'; cf., Section 2). Filtering could be achieved by using a concrete dominator, ideally $\mathit{msg}(\gamma(\mathbf{A}))$, instead of the skeleton $\lceil \mathbf{A} \rceil$ for the definition of $\rho_{\mathbf{A}}$. This, however, makes the exposition more tricky[9] and would detract from the main points of the paper. Anyway, one can always apply the technique of [Gallagher and Bruynooghe 1990] (as well as the one from [Leuschel and Sørensen 1996]) as a post-processing.

8. GENERIC CORRECTNESS RESULTS

In this section we will present and prove two general correctness results (Theorems 8.2 and 8.7).

[8] Nor removal of duplicate calls. In general this does not preserve computed answers (but will produce more general answers) but is, e.g., required for tupling the Fibonacci function. It is quite straightforward to add this possibility to the framework.

[9] Indeed, although all concretisations of $\mathbf{A}$ will be an instance of $\mathit{msg}(\gamma(\mathbf{A}))$, this does not necessarily hold for the heads H and bodies B generated by the abstract unfolding.

8.1 Correctness for Computed Answers

For technical reasons we have to introduce the concept of admissible renamings (as in [Leuschel and De Schreye 1998]).

Definition 8.1 Let Q, Q' be two conjunctions, $\mathcal{A}$ a set of abstract conjunctions, and ρ an atomic renaming for $\mathcal{A}$. Then Q' is called an *admissible renaming of* Q *wrt* $\mathcal{A}$ iff there exist conjunctions $Q_1, \ldots, Q_n$ and abstract conjunctions $\mathbf{A}_1, \ldots, \mathbf{A}_n$ such that:

1. $Q = \leftarrow Q_1, \ldots, Q_n$
2. $\mathbf{A}_i \in \mathcal{A}$
3. $Q_i \in \gamma(\mathbf{A}_i)$
4. $Q' = \leftarrow \rho_{\mathbf{A}_1}(Q_1), \ldots, \rho_{\mathbf{A}_n}(Q_n)$

Any variant of Q' is called an *admissible renamed variant of* Q *wrt* $\mathcal{A}$. A conjunction Q for which an admissible renaming exists is said to be *covered by* $\mathcal{A}$.

Theorem 8.2 Let P' be an abstract partial deduction of P wrt a covered set of abstract conjunctions $\mathcal{A}$ and let Q' be an admissible renamed variant of Q wrt $\mathcal{A}$. Then

(1) If $P \cup \{\leftarrow Q\}$ has an SLD-refutation with computed answer θ then $P' \cup \{\leftarrow Q'\}$ has an SLD-refutation with computed answer θ' such that $Q\theta \approx Q'\theta'$.
(2) If $P' \cup \{\leftarrow Q'\}$ has an SLD-refutation with computed answer θ' then $P \cup \{\leftarrow Q\}$ has an SLD-refutation with computed answer θ such that $Q\theta \approx Q'\theta'$.
(3) If $P' \cup \{\leftarrow Q'\}$ has a finitely-failed SLD-tree then so does $P \cup \{\leftarrow Q\}$.

To prove the theorem, we first have to establish a series of lemmas and some useful notations.

We define, for a substitution $\theta = \{X_1/t_1, \ldots, X_n/t_n\}$, the domain $dom(\theta) = \{X_1, \ldots, X_n\}$ and the range $ran(\theta) = vars(t_1) \cup \ldots vars(t_n)$. We also define $vars(\theta) = ran(\theta) \cup dom(\theta)$.

We start out with a useful lemma from

Lemma 8.3 Let $Q \approx Q'$ and let τ be an SLD-tree for $P \cup \{\leftarrow Q\}$. Also, let $\mathcal{X}$ be an arbitrary finite set of variables. Then there exists an SLD-tree τ' for $P \cup \{\leftarrow Q'\}$ such that

- $Q \leadsto_\tau \langle L, \theta \rangle \Rightarrow Q' \leadsto_{\tau'} \langle L', \theta' \rangle$ with $Q\theta \leftarrow L \approx Q'\theta' \leftarrow L'$
- $Q' \leadsto_{\tau'} \langle L', \theta' \rangle \Rightarrow Q \leadsto_\tau \langle L, \theta \rangle$ with $Q\theta \leftarrow L \approx Q'\theta' \leftarrow L'$
- and all the variants of clauses of P used in τ' have no variables in common with $\mathcal{X}$.

Proof. This is an obvious consequence from Lemma 4.9 in [Lloyd and Shepherdson 1991] which states that

> Let R be the resultant of an SLDNF-derivation D from a normal goal $\leftarrow Q$, and α a substitution. If there is a corresponding derivation D' from $\leftarrow Q\alpha$ then its resultant R' is an instance of R.

We apply this Lemma 4.9 twice, once for Q and $Q\alpha = Q'$ and then for Q' and $Q\alpha' = Q$. We know that a "corresponding derivation" exists by (correct versions of) the lifting lemma (e.g., Lemma 4.1 in [Lloyd and Shepherdson 1991]). □

Corollary 8.4 Let $Q \rightsquigarrow_\tau \langle L, \theta\rangle$. Also, let $\mathcal{X}$ be an arbitrary finite set of variables. Then there exists a τ' such that $Q \rightsquigarrow_{\tau'} \langle L', \theta'\rangle$ with $\langle L, \theta\rangle \approx_Q \langle L', \theta'\rangle$ and all the variants of clauses of P used in τ' have no variables in common with $\mathcal{X}$. This also implies $vars(\theta') \cap \mathcal{X} \subseteq vars(Q)$.

Lemma 8.5 Let ρ be an atomic renaming for $\mathcal{A}$ and let $\mathbf{A} \in \mathcal{A}$, $H \preceq \lceil\mathbf{A}\rceil$, $Q \preceq \lceil\mathbf{A}\rceil$. Then $mgu(H, Q) \approx_{H\wedge Q} mgu(\rho_\mathbf{A}(H), \rho_\mathbf{A}(Q))$. We also have that $vars(H) = vars(\rho_\mathbf{A}(H))$ and $\rho_\mathbf{A}(H)\sigma = \rho_\mathbf{A}(H\sigma)$ for any substitution σ.

Proof. $vars(H) = vars(\rho_\mathbf{A}(H))$ is obvious from Definition 7.4, as $\rho_\mathbf{A}(H) = A\theta$, $H = \lceil\mathbf{A}\rceil\theta$, and $vars(A) = vars(\lceil\mathbf{A}\rceil)$.
$\rho_\mathbf{A}(H)\sigma = \rho_\mathbf{A}(H\sigma)$ is again obvious from Definition 7.4. Indeed, we have $\rho_\mathbf{A}(H\sigma) = A\theta'$, with $H\sigma = \lceil\mathbf{A}\rceil\theta'$. From this follows $\lceil\mathbf{A}\rceil\theta' = (\lceil\mathbf{A}\rceil\theta)\sigma$, and thus, as $vars(A) = vars(\lceil\mathbf{A}\rceil)$, we have that $A\theta' = A\theta\sigma$, i.e., $\rho_\mathbf{A}(H)\sigma = \rho_\mathbf{A}(H\sigma)$.
By the point above we have that every unifier σ of H and Q must also be a unifier of $\rho_\mathbf{A}(H)$ and $\rho_\mathbf{A}(Q)$ (indeed, $\rho_\mathbf{A}(H)\sigma = \rho_\mathbf{A}(H\sigma) = \rho_\mathbf{A}(Q\sigma) = \rho_\mathbf{A}(Q)\sigma$) and vice versa. By uniqueness of the *mgu* up to variable renaming we must thus have $mgu(H, Q) \approx_{H\wedge Q} mgu(\rho_\mathbf{A}(H), \rho_\mathbf{A}(Q))$. □

To simplify the presentation of the proofs below, we will from now on assume that the *mgu* is devised so that (this can always be achieved):

$$mgu(H, Q) = mgu(\rho_\mathbf{A}(H), \rho_\mathbf{A}(Q)) \tag{5}$$

We are now in a position to prove our theorem.

Proof. **of Theorem 8.2** Both the proof of soundness and completeness are by induction on the length of the refutations.

First let $Q_1, \ldots, Q_n$ and $\mathbf{A}_1, \ldots, \mathbf{A}_n$ be the concrete and abstract conjunctions which satisfy Definition 8.1 for Q and a variant Q'' of Q'. In particular we have $Q = Q_1 \wedge \ldots Q_n$ with $Q_i \in \gamma(\mathbf{A_i})$. We know that for some renaming substitution σ we have: $Q' = Q''\sigma = \rho_{\mathbf{A_1}}(Q_1)\sigma \wedge \ldots \rho_{\mathbf{A_n}}(Q_n)\sigma = \rho_{\mathbf{A_1}}(Q_1\sigma) \wedge \ldots \rho_{\mathbf{A_n}}(Q_n\sigma)$ (by Lemma 8.5).
Point 2. (soundness of P'):
We proceed by induction on the length of the refutation δ for $P' \cup \{\leftarrow \rho_{\mathbf{A_1}}(Q_1) \wedge \ldots \rho_{\mathbf{A_n}}(Q_n)\}$.
Base Case:
The base case ($len = 0$ and thus $n = 0$ and $\leftarrow Q = \leftarrow Q' = \Box$) is trivial.
Induction Step:
For the induction step let us examine the first resolution step of δ resolving a selected atom $\rho_{\mathbf{A_i}}(Q_i\sigma)$ in Q' with a clause $\rho_{\mathbf{A_i}}(H) \leftarrow \rho_{\mathcal{A},\mathbf{B}}(B)$ via *mgu* θ_1 and where $C \in aunfold(P, \mathbf{A_i})$ with $C \approx H \leftarrow B$ and $\mathbf{B} = aresolve(\mathbf{A_i}, C)$ (and where $H \leftarrow B$ is renamed apart wrt Q'). The resolvent R' of Q' in P' is thus (c.f., Figure 5):

$$\begin{aligned} R' &= \leftarrow \rho_{\mathbf{A_1}}(Q_1\sigma)\theta_1 \wedge \ldots \rho_{\mathcal{A},\mathbf{B}}(B)\theta_1 \wedge \ldots \rho_{\mathbf{A_n}}(Q_n\sigma)\theta_1 \\ &= \leftarrow \rho_{\mathbf{A_1}}(Q_1\sigma\theta_1) \wedge \ldots \rho_{\mathcal{A},\mathbf{B}}(B)\theta_1 \wedge \ldots \rho_{\mathbf{A_n}}(Q_n\sigma\theta_1) \end{aligned}$$

Step 1. We will now show that R' is an admissible renaming of

$$\tilde{R} = \leftarrow Q_1\sigma\theta_1 \wedge \ldots B\theta_1 \wedge \ldots Q_n\sigma\theta_1$$

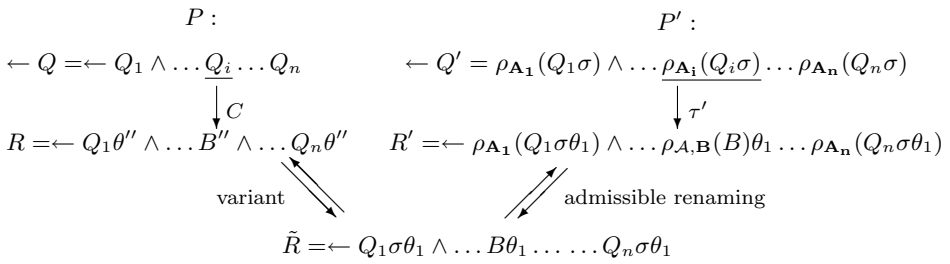

Fig. 5. Illustrating the proof of Theorem 8.2

Below, in step 2., we will show that we can produce a resolvent R in P which is a variant of $\tilde{R}$. Together, this will allow us to apply the induction hypothesis.

Let us first examine the structure of $\rho_{\mathcal{A},\mathbf{B}}(B)\theta_1$. We have by Definition 7.5:

$$\rho_{\mathcal{A},\mathbf{B}}(B)\theta_1 = (\rho_{\mathbf{G_1}}(B_1) \wedge \ldots \rho_{\mathbf{G_k}}(B_k))\theta_1 = \rho_{\mathbf{G_1}}(B_1\theta_1) \wedge \ldots \rho_{\mathbf{G_k}}(B_k\theta_1))$$

where $B = B_1 \wedge \ldots B_k$, $\mathbf{B}$ is covered by $\langle \mathbf{G_1}, \ldots, \mathbf{G_k} \rangle$ with $\mathbf{G_i} \in \mathcal{A}$ and $B_i \preceq \lceil \mathbf{G_i} \rceil$. Let us now verify that the 4 points of Definition 8.1 are satisfied for R and $\tilde{R}$:

1. $\tilde{R} = \leftarrow Q_1\sigma\theta_1 \wedge \ldots B_1\theta_1 \wedge \ldots B_k\theta_1 \wedge \ldots Q_n\sigma\theta_1$ is a valid partitioning of $\tilde{R}$ into subconjunctions
2. We have $\mathbf{A_i} \in \mathcal{A}$ from the fact that Q'' is an admissible renaming of Q.
   We have $\mathbf{G_i} \in \mathcal{A}$ from Definition 7.5.
3. We have $Q_i\sigma\theta_1 \in \gamma(\mathbf{A_i})$ by downwards-closure of $\gamma(.)$ and as $Q_i\sigma \in \gamma(\mathbf{A_i})$ from the fact that Q'' is an admissible renaming of Q.
   We have $B_i\theta_1 \in \gamma(\mathbf{G_i})$ by downwards-closure of $\gamma(.)$ and as $B_i \in \gamma(\mathbf{G_i})$ because by correctness of *aresolve* we have $B \in \gamma(\mathbf{B})$ (by Definition 7.6 we have $\mathbf{B} = \mathit{aresolve}(\mathbf{A_i}, C)$ and we know $Q_i \in \gamma(\mathbf{A_i})$ from the fact that Q'' is an admissible renaming of Q).
4. $R' = \leftarrow \rho_{\mathbf{A_1}}(Q_1\sigma\theta_1) \wedge \ldots \rho_{\mathbf{G_1}}(B_1\theta_1) \wedge \ldots \rho_{\mathbf{G_k}}(B_k\theta_1)) \ldots \rho_{\mathbf{A_n}}(Q_n\sigma\theta_1)$

Step 2. We will now show that a variant R of $\tilde{R}$ is a resolvent of Q in P.

We know, by Lemma 8.5, that θ_1 is also an *mgu* of $Q_i\sigma$ and H. Hence, by our assumption (5) we know that $Q_i\sigma \rightsquigarrow_C \langle B\theta_1, \bar{\theta_1} \rangle$, where $\bar{\theta_1} = \theta_1\|_{vars(Q_i\sigma}$.

As we have $Q_i\sigma \rightsquigarrow_C \langle B\theta_1, \bar{\theta_1} \rangle$, Point 2 of Definition 5.3 (defining *aunfold*) therefore ensures that we can find an SLD-tree τ for $P \cup \{\leftarrow Q_i\sigma\}$ such that

$$Q_i\sigma \rightsquigarrow_\tau \langle B', \theta' \rangle \text{ with } Q_i\sigma\theta_1 \leftarrow B\theta_1 \approx Q_i\sigma\theta' \leftarrow B' \tag{6}$$

Now, as $Q_i \approx Q_i\sigma$, by Lemma 8.3, we can deduce that we can find another SLD-tree τ' for $P \cup \{\leftarrow Q_i\}$ such that

$$Q_i \rightsquigarrow_{\tau'} \langle B'', \theta'' \rangle \text{ with } Q_i\theta'' \leftarrow B'' \approx Q_i\sigma\theta' \leftarrow B' \tag{7}$$

By, Lemma 8.3, we can also always construct τ' such that the clauses of P have not only been renamed apart wrt Q_i but wrt the entire Q. Hence, we can generate a resolvent R in P which has the following form (because we renamed apart wrt the entire Q and by the subderivation lemma [Lloyd and Shepherdson 1991]):

$$R = \leftarrow Q_1\theta'' \wedge \ldots B'' \wedge \ldots Q_n\theta''$$

Let us now prove that R is a variant of $\tilde{R}$:

– By transitivity of $\approx$ we know that $Q_i\theta'' \leftarrow B'' \approx Q_i\sigma\theta_1 \leftarrow B\theta_1$. Hence, we can find substitutions γ and γ^{-1} such that $(Q_i\theta'')\gamma = Q_i\sigma\theta_1$, $(B'')\gamma = B\theta_1$, $Q_i\theta'' = (Q_i\sigma\theta_1)\gamma^{-1}$ and $B'' = (B\theta_1)\gamma^{-1}$. We can also choose γ, γ^{-1} so that there are no superfluous bindings, i.e., $dom(\gamma) \subseteq vars(Q_i\theta'' \leftarrow B'')$, $ran(\gamma) \subseteq vars(Q_i\sigma\theta_1 \leftarrow B\theta_1)$, $dom(\gamma^{-1}) \subseteq vars(Q_i\sigma\theta_1 \leftarrow B\theta_1)$, $ran(\gamma^{-1}) \subseteq vars(Q_i\theta'' \leftarrow B'')$.
– We know that $Q = Q_1 \wedge \ldots Q_n$ is a variant of $Q\sigma = Q_1\sigma \wedge \ldots Q_n\sigma$. Hence we can find a substitution σ^{-1} such that $(Q\sigma)\sigma^{-1} = Q$.
– We will now define two substitutions $\gamma' \supseteq \gamma$ and $\gamma'^{-1} \supseteq \gamma^{-1}$ such that $R\gamma' = \tilde{R}$ and $\tilde{R}\gamma'^{-1} = R$.
  i. By construction of γ and γ^{-1} we already have $(B'')\gamma = B\theta_1$ $B'' = (B\theta_1)\gamma^{-1}$
  ii. We now have to examine the conjunctions $Q_j\sigma$ and Q_j, for $j \neq i \wedge 1 \geq j \geq n$, in $\tilde{R}$ and R respectively. As $Q_j\sigma$ and Q_j are variants we only have to examine the variable positions in $Q_j\sigma$ and Q_j. Let X be a variable at some position in $Q_j\sigma$ and Y the corresponding variable at the same position in Q_j. We have to show that we can map $X\theta_1$ to $Y\theta''$ and vice-versa. There are two possibilities:
  a) $X \in vars(Q_i\sigma)$ As we know that $(Q_i\theta'')\gamma = Q_i\sigma\theta_1$ $Q_i\theta'' = (Q_i\sigma\theta_1)\gamma^{-1}$ we can deduce that $(Y\theta'')\gamma = X\theta_1$ $Y\theta'' = (X\theta_1)\gamma^{-1}$.
  b) $X \notin vars(Q_i\sigma)$ In that case we know that $Y \notin vars(Q_i)$ (otherwise Q is not a variant of $Q\sigma$). Hence we can set $\gamma' = \gamma \cup \{Y/X\}$ and $\gamma'^{-1} = \gamma^{-1} \cup \{X/Y\}$. γ' is a properly defined substitution as X cannot appear in $B\theta_1$ and thus $ran(\gamma)$ because
    – $H \leftarrow B$ is renamed apart wrt $vars(Q') = vars(Q\sigma)$ and
    – θ_1 is a *relevant mgu* of $Q_i\sigma$ and H.

  γ'^{-1} in turn is also a properly defined substitution as Y cannot appear in B'' by a similar reasoning on the *mgu* and renaming apart in τ' (by our earlier assumption on τ', stating that the clauses of P have not only been renamed apart wrt Q_i but wrt the entire Q). We thus trivially have $(Y\theta'')\gamma = X\theta_1$ $Y\theta'' = (X\theta_1)\gamma^{-1}$. Also, note that γ', γ'^{-1} will still satisfy the requirements of case **a)** above.

  We now simply define the final γ' and γ'^{-1} to be the union of all the γ', γ'^{-1} defined for the cases **b)** above. This is a properly defined substitution (as $X\sigma = Y$ and $X\sigma = Z$ implies $Y = Z$, i.e., there can be no conflicts between the bindings) and we thus have found substitutions such that $R\gamma' = \tilde{R}$ and $\tilde{R}\gamma'^{-1} = R$.

Step 3. We can now apply the induction hypothesis, as we have proven that the resolvent R' in P' is an admissible renamed variant of the corresponding resolvent R in P. Notably, we know that for any computed answer θ_2 of R' there exists a computed answer θ of R such that $R\theta \approx R'\theta_2$. In summary, we have Q leads to R via θ'', R has a c.a.s. θ, Q' leads to R' via θ_1, R' has a c.a.s. θ_2. So, we just have to prove that $Q\theta''\theta \approx Q'\theta_1\theta_2$ to complete the soundness proof. We can use Corollary 8.4 to ensure both

$$vars(\theta_2) \cap vars(Q') \subseteq vars(R') \quad \text{and} \quad vars(\theta) \cap vars(Q) \subseteq vars(R) \tag{8}$$

In fact, we can easily establish that $Q\theta'' \approx Q'\theta_1$ because
- indeed the reasoning in point **ii.** above is also valid for $i = j$ [but only subcase **a)** will apply] and
- we can thus use the same substitutions γ', γ'^{-1} to show $Q\theta''\gamma' = Q'\theta_1$ and $Q\theta'' = Q'\theta_1\gamma'^{-1}$.

We thus simply examine every variable position in $Q\theta''$ and the corresponding variable position in $Q'\theta_1$. Let X be a variable at some position in $Q'\theta_1$ and Y the corresponding variable at the same position in $Q\theta''$. We have to show that we can map $X\theta_2$ to $Y\theta$ and vice-versa. There are again two cases:
- If $X \notin vars(R')$ then $X\theta' = X$ (as θ is a c.a.s. for R', i.e., $ran(\theta) \subseteq vars(R')$) and we must also have $X \in Q_i\sigma\theta_1$ and $X \notin Q_j\sigma\theta_1$ for $j \neq i$ ($\rho_{\mathbf{A_j}}(Q_j\sigma\theta_1)$ for $j \neq i$ all appear in R') and hence $Y \in Q_j\theta''$ and $Y \notin Q_j\theta''$ for $j \neq i$ as well (as $X\gamma^{-1} = Y$). This implies $Y \notin vars(R)$ (because $Q_i\theta'' \leftarrow B'' \approx Q_i\sigma\theta_1 \leftarrow B\theta_1$, i.e., Y cannot appear in B'') and we thus have $(Y\theta)\gamma = X\theta_2$ $Y\theta = (X\theta_2)\gamma^{-1}$.
- On the other hand, if $X \in vars(R')$ then $Y \in vars(R)$ (if $X \in Q_i\sigma\theta_1$ then this follows from $Q_i\theta'' \leftarrow B'' \approx Q_i\sigma\theta_1 \leftarrow B\theta_1$; otherwise if $X \in Q_j\sigma\theta_1$ with $j \neq i$ then this follows from $X\gamma^{-1} = Y$ and the fact $Q_j\sigma\theta_1$ features in R') and we know we can map $X\theta_2$ to $Y\theta$ and back using the simplest substitutions γ'', γ''^{-1} which map back and forth between R and R' (i.e., $R\theta\gamma'' = R'\theta_2$, $R'\theta_2\gamma''^{-1} = R\theta$, where also $dom(\gamma'') \subseteq vars(R\theta)$, and $dom(\gamma''^{-1}) \subseteq vars(R'\theta_2)$).

Now, $\gamma' \cup \gamma''$ is a well defined substitution because, by our assumption (8) above on renaming apart of clauses, the variables in the terms $X\theta_2$ cannot be variables that appear in $Q_i\sigma\theta_1$ but not in $vars(R)$, i.e., there is no clash between the bindings in γ' and γ''. By a similar reasoning, $\gamma'^{-1} \cup \gamma''^{-1}$ is a well defined substitution. We have thus established the induction hypothesis for Q and Q' and thus completed the soundness proof.

Point 1. (completeness of P'):

We now proceed by induction on the length of the refutation δ for $P \cup \{\leftarrow Q_1 \wedge \ldots Q_n\}$ which yields the computed answer θ. The base case ($len = 0$ and thus $n = 0$) is again trivial. For the induction step, let Q_i be the selected literal. As $Q_i \in \gamma(\mathbf{A_i})$ we can apply Definition 5.3 of *aunfold* to deduce that there is an SLD-tree τ for $P \cup \{\leftarrow Q_i\}$ such that point 1 of Definition 5.3 holds. By independence of the selection rule ([Apt 1990; Lloyd 1987]) we know that we do not lose any computed answers by enforcing a particular selection rule. Without loss of generality, we can thus assume that a prefix of δ is a branch in τ', i.e., δ unfolds $\leftarrow Q_i$ in the manner prescribed by τ' of the soundness part of the proof.[10]

We can now use point 1 of Definition 5.3 defining *aunfold* to show that when selecting the atom $\rho_{\mathbf{A_i}}(Q_i\sigma)$ in Q' and resolving it with the clause $\rho_{\mathbf{A_i}}(H) \leftarrow \rho_{\mathcal{A},\mathbf{B}}(B)$ via *mgu* θ_1 we get a resolvent R' which has exactly the same structure as in the soundness part of the proof (c.f., Figure 5). The proof that R' is an admissible renamed variant of R is then exactly as in the soundness part (Steps 1 and 2). The same holds for applying the induction hypothesis to prove $Q\theta''\theta \approx Q'\theta_1\theta_2$ (Step 3). The completeness proof is thus complete.

[10] If we want to establish the preservation of finite failure it is vital that the unfoldings performed by τ are fair. For computed answers, however, this does not matter.

Point 3. (soundness for finite failure):
We again do a proof by induction, but this time on the depth of the failed SLD-tree for $P' \cup \{\leftarrow Q'\}$.
Base Case:
The SLD-tree has just a single node in which a literal has been selected which fails immediately, i.e., does not unify with any clause in P'. This implies that the goal Q finitely fails in P, because by point 1 of Definition 5.3 we know we can find an SLD-tree τ for which no s_1 satisfies $Q \rightsquigarrow_\tau s_1$, i.e., a finitely failed SLD-tree for $P \cup \{\leftarrow Q\}$.
Induction Step:
We will do the exact same resolution step as in the proof for the soundness part: we suppose that we select an atom $\rho_{\mathbf{A_i}}(Q_i\sigma)$ in Q'. We we now resolve the selected atom with a clause $\rho_{\mathbf{A_i}}(H) \leftarrow \rho_{\mathcal{A},\mathbf{B}}(B)$ of P' we get exactly the same picture as in the soundness part (the proof in the soundness part works for any resolvent!). So, we can re-use Steps 1 and 2 of the proof of the soundness part for every resultant R' to establish that R' it is an admissible renamed variant of the corresponding resolvent R in P. We can thus apply the induction hypothesis to conclude that for each resolvent R we can construct a finitely failed SLD-tree.

The only thing we have to establish, to be able to combine all the results into a big finitely failed tree for Q, is that the initial SLD-tree τ' used in the soundness proof can be made to be the *same for all* resolvents R'. This can be easily ensured using Lemma 8.3 and because Definition 5.3 provides us with a single SLD-tree τ valid for all resolvents!
We can thus combine, using the subderivation lemma [Lloyd and Shepherdson 1991], all failed SLD-trees for the resolvents into one big finitely failed SLD-tree for $P \cup \{\leftarrow Q\}$. □

8.2 Preservation of Finite Failure

In order to derive results about the preservation of finite failure in P' we have to impose that the unfolding operation *aunfold* is in some sense *fair*, i.e. when computing $aunfold(P, \mathbf{A})$ it eventually selects every conjunct of $Q \in \gamma(\mathbf{A})$ in every non-failing branch. Otherwise, the unfolding *aunfold* might impose an unfair selection rule onto the specialised program, and finite failure might no longer be preserved. For example, one should not be able to transform the program $P = \{t \leftarrow p \wedge fail,\ p \leftarrow p\}$ into $P' = \{t \leftarrow pf,\ pf \leftarrow pf\}$, where, e.g., $\mathbf{A} = p \wedge fail$ in the $\mathcal{PD}$-domain and $\rho_{\mathbf{A}} = pf$. (This condition is quite similar to the local improvement condition in [Sands 1996] for functional programs.)

Definition 8.6 Let the goal $G' = \leftarrow (A_1 \wedge \ldots A_{i-1} \wedge B_1 \wedge \ldots B_k \wedge A_{i+1} \wedge \ldots A_n)\theta$ be derived via an SLD-resolution step from the goal $G = \leftarrow A_1 \wedge \ldots A_i \wedge \ldots A_n$, and the clause $H \leftarrow B_1 \wedge \ldots B_k$, with selected atom A_i. We say that the atoms $A_1\theta, \ldots, A_{i-1}\theta, A_{i+1}\theta, \ldots, A_n\theta$ are *inherited from G in G'*. We extend this notion to derivations by taking the transitive and reflexive closure.

An complete SLD-tree τ for $P \cup \{G\}$ is said to be *fair* iff every branch is either finitely failed, or for every goal G_i in a non-failing branch there exists a descendant G_j such that no atoms are inherited from G_i in G_j. A finite, incomplete SLD-tree τ for $P \cup \{G\}$ is said to be *fair* iff no atom in a leaf goal L of a non-failing branch

of τ is inherited from G in L.

We call an abstract unfolding rule *fair* if we can always find a finite, fair SLD-tree τ which satisfies the points 1, 2 of Definition 5.3.

Note that a finite, complete SLD-tree is always fair. We can now present the following theorem about the preservation of finite failure.

Theorem 8.7 Let P' be an abstract partial deduction of P wrt a covered set of abstract conjunctions $\mathcal{A}$ using a fair abstract unfolding *aunfold*, and let Q' be an admissible renamed variant of Q wrt $\mathcal{A}$.

– If $P \cup \{\leftarrow Q\}$ has a finitely-failed SLD-tree then so does $P' \cup \{\leftarrow Q'\}$.

Note that for atomic abstract conjunctions, every finite, non-trivial SLD-tree is fair. So, if we just have atomic abstract conjunctions, finite failure will always be preserved (non-trivial trees are disallowed in Definition 5.3). Hence Theorem 6.5 is a direct consequence of Theorems 8.2 and 8.7.

One can actually extend the result to allow *aunfold* to be just *weakly fair* [Leuschel et al. 1996; Leuschel 1997]. Intuitively, this means that $aunfold(P, \mathbf{Q})$ can be unfair for a certain number of atoms, as long as we can be sure that these atoms will eventually be selected (for non-failing derivations) within other abstract conjunctions.

The proof of the theorem is as follows:

Proof. **of Theorem 8.7** We use the same assumptions about the structure of Q and Q' as at the beginning of the proof for Theorem 8.2. Notably, again, let $Q_1, \ldots, Q_n$ and $\mathbf{A}_1, \ldots, \mathbf{A}_n$ be the concrete and abstract conjunctions which satisfy Definition 8.1 for Q and a variant Q'' of Q'. Again, we have $Q = Q_1 \wedge \ldots Q_n$ with $Q_i \in \gamma(\mathbf{A_i})$ and we chose the same renaming substitution σ such that: $Q' = Q''\sigma = \rho_{\mathbf{A_1}}(Q_1)\sigma \wedge \ldots \rho_{\mathbf{A_n}}(Q_n)\sigma = \rho_{\mathbf{A_1}}(Q_1\sigma) \wedge \ldots \rho_{\mathbf{A_n}}(Q_n\sigma)$ (by Lemma 8.5).

We know by Theorem 13.6 in [Lloyd 1987][page 77] that if there exists a finitely failed SLD-tree for $P \cup \{\leftarrow Q\}$ then *every* fair SLD-tree for $P \cup \{\leftarrow Q\}$ is finitely failed.

We proceed by induction on the depth of the finitely failed SLD-tree for $P \cup \{\leftarrow Q_1 \wedge \ldots Q_n\}$.

Let Q_i be the selected literal at the root. As $Q_i \in \gamma(\mathbf{A_i})$ we can apply Definition 5.3 of *aunfold* to deduce that there is a *fair* SLD-tree τ' for $P \cup \{\leftarrow Q_i\}$ such that point 1 of Definition 5.3 holds.

Base Case: If this SLD-tree τ' is finitely failed we are in the base case of our induction, and we know by that $P \cup \{\leftarrow Q'\}$ fails immediately when selecting $\rho_{\mathbf{A_i}}(Q_i\sigma)$.

Induction Step: As τ' is fair, we know that, without loss of generality, we can assume that τ' is the initial subtree of a *finitely* failed SLD-tree for $P \cup \{\leftarrow Q\}$ (and always choosing such τ''s will lead to a finitely failed SLD-tree).

We now do the exact same resolution step for $P' \cup \{\leftarrow Q'\}$ as in the proof for the soundness proof of Theorem 8.2: i.e., we select the atom $\rho_{\mathbf{A_i}}(Q_i\sigma)$ in Q'. We we now resolve the selected atom with all matching clauses $\rho_{\mathbf{A_i}}(H) \leftarrow \rho_{\mathcal{A},\mathbf{B}}(B)$ of P' and for every resolvent we get exactly the same picture as in the soundness proof of Theorem 8.2 for some leaf goal R in τ' (the proof in the soundness part works for any resolvent!). So, we can re-use Steps 1 and 2 of the proof of the soundness part for every resultant R' to establish that R' it is an admissible renamed variant

of the corresponding resolvent R in P. We can thus apply the induction hypothesis to conclude that for each resolvent R' we can construct a finitely failed SLD-tree for $P' \cup \{\leftarrow R'\}$.

The only thing we have to establish, to be able to combine all the results into a big finitely failed tree for Q', is that the initial SLD-tree τ used in the soundness proof for $Q\sigma$ can be made to be the *same for all* resolvents R and R'. This can be easily ensured using Lemma 8.3 and because Definition 5.3 provides us with a single SLD-tree τ' valid for all resolvents!

We can thus again combine, using the subderivation lemma from [Lloyd and Shepherdson 1991], all the failed SLD-trees for the resolvents into one big finitely failed SLD-tree for $P' \cup \{\leftarrow Q'\}$. □

9. SOME INSTANCES OF ABSTRACT PARTIAL DEDUCTION

In this section we show how some of the existing logic program specialization techniques can be cast into our framework, and how easily the correctness results can be re-used. In fact, to re-use our correctness results one has to prove that the particular *aunfold* under consideration satisfies Definition 5.3, that *aresolve* satisfies Definition 5.4 and finally that the widening *ageneralize* satisfies $ageneralize(\mathcal{A}) \sqsupseteq_{split} \mathcal{A}$.

9.1 Classical and Conjunctive Partial Deduction

Classical partial deduction [Lloyd and Shepherdson 1991; Gallagher 1993] can be seen as an instance of our framework simply by taking

- the $\mathcal{PD}$-domain (i.e. the concrete domain is the abstract domain and an abstract element represents all its instances) as our abstract domain,
- abstract unfolding is done by an unfolding rule as defined in Definition 3.4. I.e., *aunfold* builds an SLD-tree and returns the resultants of the tree.
- abstract resolution simply returns the bodies of the above resultants: $aresolve(\mathbf{A}, H \leftarrow B) = B$.
- *ageneralize* is such that it only produce sets of atoms and the initial abstract conjunction $\mathbf{A}$ is an atom.

To represent *conjunctive* partial deduction [Leuschel et al. 1996; Glück et al. 1996; Leuschel 1997] we just have to drop the last requirement.

As a corollary of Proposition 5.6, we know that we satisfy Definition 5.3 of an abstract unfolding. The fact that abstract resolution $aresolve(\mathbf{A}, H \leftarrow B) = B$ satisfies Definition 5.4 follows from our discussions in Section 3.3. We can thus apply Theorem 8.2. For classical partial deduction of atoms, fairness of *aunfold* trivially follows from the fact that τ is non-trivial. We can hence also apply Theorem 8.7.

It can also be easily verified that the generalization operations used in existing classical or conjunctive partial deduction techniques satisfy our requirements in Definition 6.6.

Removal of Redundant Clauses. [de Waal and Gallagher 1991; Gallagher and de Waal 1992; de Waal and Gallagher 1994] present a classical partial deduction approach, but where a resultant $Q\theta_k \leftarrow B_k$ is removed from $aunfold(P, Q)$ if it can be proven by a bottom-up abstract interpretation that B_k fails. Such a resultant is called *redundant*. In case B_k fails finitely, it is very easy to prove that this extension of partial deduction satisfies Definitions 5.3 and 5.4 (simply use, in the proof of

Proposition 5.6, a tree τ' instead of τ where all branches ending in a redundant B_j are fully expanded until failure). In case B_k fails infinitely, the situation is more complicated, and we cannot directly use our top-down framework. We will return to the issue of combining bottom-up and top-down approaches in Section 10.

9.2 Ecological and Constrained Partial Deduction

Ecological partial deduction [Leuschel 1995; Leuschel et al. 1998; Leuschel 1997] (and its ancestor [Gallagher and Bruynooghe 1991]) specializes sets of characteristic atoms of the form (A, τ), where A is an ordinary atom and τ a characteristic tree (basically a representation of the shape of an SLD-tree). Intuitively (A, τ) represents all instances of A which have τ as a characteristic tree. Ecological partial deduction can be seen as an instance of the above generic framework by using an abstract domain $(\mathcal{AQ}, \gamma)$ with

- $\mathcal{AQ} = (\mathcal{A}, T)$, where $\mathcal{A}$ is the set of atoms and T is the set of characteristic trees [Gallagher and Bruynooghe 1991; Gallagher 1991].
- $\gamma((A, \tau)) = \{A'' \mid A'' \preceq A' \preceq A \wedge A' \text{ has characteristic tree } \tau \}$,

and where abstract unfolding and resolution are defined by

- $aunfold(P, (A, \tau))$ is based on using the SLD-tree for $P \cup \{\leftarrow A\}$ according to the shape indicated by τ (and removing the resultants which are not present in τ; see [Leuschel 1995; Leuschel et al. 1998; Leuschel 1997]).
- $aresolve((A, \tau), A\theta \leftarrow B) = (B, \tau')$ where τ' is the characteristic tree for an SLD-tree for $P \cup \{\leftarrow B\}$.

It is again very easy to prove that the above operations satisfy our requirements in Definitions 5.3 and 5.4, thus making our correctness results immediately applicable.

Constrained partial deduction [Leuschel and De Schreye 1998] specialises sets of constrained atoms of the form $c\Box A$ where A is an ordinary atom an c a constraint on the variables in A. For e.g., the concretisation function we have $\gamma(c\Box A) = \{A\theta \mid \mathcal{D} \models \forall(c\theta)\}$, where $\mathcal{D}$ is the underlying constraint structure and we can cast constrained partial deduction into the our framework and the correctness results from [Leuschel and De Schreye 1998] are again a special case of our generic results.

The present framework can now be used to easily extend both methods to handle conjunctions or even to integrate all of these methods into one powerful top-down specialization method.

9.3 Partial Deduction using Regular Types

Regular types encoded as regular unary logic programs [Yardeni and Shapiro 1990; Gallagher and de Waal 1994] have proven to be successful both for program analysis and specialization. Indeed, using regular types as an abstract domain for specialization was already proposed in [Puebla et al. 1997; Puebla et al. 1999].

Instances of our abstract partial deduction framework using regular types have recently been developed. First, [Gallagher and Peralta 2000; 2001] presents several atomic abstract partial deduction methods, one of which is formally cast into our framework. An implementation has been produced, which has been validated on practical examples.

Second, [Leuschel and Gruner 2001] presents an extension of [Gallagher and Peralta 2000; 2001] which can specialize abstract conjunctions. It is formally shown

how to perform abstract unfolding and resolution in such a setting, and the practical usefulness of combining regular types with conjunctions has been demonstrated on several examples. An implementation, using the ECCE system [Leuschel 2002] has been developed and applied to several examples; one of which we elaborate below. One possible application of the method is the model checking [Clarke et al. 1999] of process algebras.

We present some aspects of these instances of our framework below.

Definition 9.1 A *canonical regular unary clause* is a clause of the form:

$$t_0(f(X_1,\ldots,X_n)) \leftarrow t_1(X_1) \wedge \ldots \wedge t_n(X_n)$$

where $n \geq 0$ and $X_1,\ldots,X_n$ are distinct variables. A *regular unary logic (RUL) program* is a finite set of regular unary clauses, in which no two different clause heads have a common instance, together with the single fact $any(X) \leftarrow$. Given a (possibly non-ground) conjunction T and a RUL program R, we write $R \models \forall(T)$ iff $R \cup \{\leftarrow T\}$ has an SLD-refutation with the empty computed answer. Finally, the success set of a predicate t in a RUL program R is defined by $success_R(t) = \{s \mid \text{s is ground} \wedge R \models \forall(t(s))\}$.

Example 9.2 For example, given the following RUL-program R, we have $R \models \forall(t1([a]))$ and $R \models \forall(t1([X,Y]))$.

```
t1([]).                          any(X).
t1([H|T]) :- any(H),t1(T).
```

Definition 9.3 We define the *RUL-domain* $(\mathcal{AQ},\gamma)$ to consist of abstract conjunctions of the form $\langle Q,T,R\rangle \in \mathcal{AQ}$ where Q,T are concrete conjunctions and R is a RUL program such that: $T = t_1(X_1)\wedge\ldots\wedge t_n(X_n)$, where $vars(Q) = \{X_1,\ldots,X_n\}$ and t_i are predicates defined in R. The *concretisation function* γ is defined as follows: $\gamma(\langle Q,T,R\rangle) = \{Q\theta \mid R \models \forall(T\theta)\}$. T is called a *type conjunction*.

Using R from Ex. 9.2 we have that $\gamma(\langle p(X),t1(X),R\rangle) = \{p([]),p([X]),p([a]),\ldots,p([X,Y]),p([X,X]),p([a,X]),\ldots\}$. Note that abstract conjunctions from our RUL-domain are called R-conjunctions in [Gallagher and Peralta 2001].

Full details on how to implement abstract unfolding, abstract resolution and concrete abstract partial deduction procedures can be found in [Gallagher and Peralta 2000; 2001] and [Leuschel and Gruner 2001].

The following example, which was worked out using the implementation presented in [Leuschel and Gruner 2001], shows a particular verification example where conjunctions and regular types both play an important role.

Example 9.4 Take the following simple program, which simulates several problems that can happen during model checking of infinite state process algebras. Here, the predicate `trace/2` describes the possible traces of a particular (infinite state) system. In `sync_trace/2` we describe the possible traces of two synchronized copies of this system, with different start states.

```
trace(s(X),[dec|T]) :- trace(X,T).
trace(0,[stop]).
trace(s(X),[inc|T]) :- trace(s(s(X)),T).
trace(f(X),[dec|T]) :- trace(X,T).
trace(f(X),[inc|T]) :- trace(f(f(X)),T).
```

```
trace(a,[inc,stop]).
sync_trace(T) :- trace(s(0),T), trace(f(a),T).
```

As one can see, the synchronization of `s(0)` with `f(a)` will never produce a complete trace, and hence `sync_trace` will always fail. Classical partial deduction is unable to infer failure of `sync_trace`, even when using conjunctions, due to the inherent limitation of the $\mathcal{PD}$-domain to capture the possible states of our system, i.e., the possible first arguments to `trace/2`. In the RUL domain we can retain much more precise information about the calls to `trace/2`. E.g., our implementation was able to infer that the first argument to calls to `trace/2` descending from `trace(f(a),T)` will always have the type `t940` defined by:

```
t940(a):-true.
t940(f(_460)) :- t940(_460).
```

This is the residual program generated by ECCE.

```
sync_trace([inc,A|B]) :-  p_conj__2(0,A,B,a).
sync_trace__1([inc,A|B]) :-  p_conj__2(0,A,B,a).
p_conj__2(A,dec,[B|C],D) :-  p_conj__3(A,B,C,D).
p_conj__2(A,inc,[B|C],D) :-  p_conj__2(s(A),B,C,f(D)).
p_conj__3(A,dec,[B|C],D) :-   p_conj__4(A,B,C,D).
p_conj__3(A,inc,[B|C],D) :-  p_conj__2(A,B,C,D).
p_conj__4(s(A),dec,[B|C],f(D)) :-  p_conj__4(A,B,C,D).
p_conj__4(s(A),inc,[B|C],f(D)) :-  p_conj__2(A,B,C,D).
```

This program contains no facts and a simple bottom-up post-processing (e.g., the one implemented in ECCE based upon [Marriott et al. 1990]) can infer that `sync_trace` fails.

Observe that a deterministic regular type analysis on its own (i.e., without conjunctions) cannot infer failure of `sync_trace`. The reason is that, while the regular types are precise enough to characterize the possible states of our infinite state system, they are not precise enough to characterize the possible traces of the system! For example, the top-down regular type analysis of the SP system produces the following result for the possible answers of `sync_trace`:

```
sync_trace__ans(X1) :- t230(X1).
t230([X1|X2]) :- t231(X1),t232(X2).
t231(inc) :- true.                      t233(inc) :- true.
t231(dec) :- true.                      t233(dec) :- true.
t231(stop) :- true.                     t233(stop) :- true.
t232([X1|X2]) :- t233(X1),t232(X2).
t232([]) :- true.
```

In other words, the regular type analysis on its own was incapable of detecting the failure. Using our approach, the conjunctive partial deduction component achieves "perfect" precision (by keeping the variable link between the two copies of our system), and it is hence not a problem that the traces cannot be accurately described by regular types.[11] This underlines our hope that adding conjunctions to regular

[11] The non-deterministic regular type analysis of [Gallagher and Puebla 2002] actually is precise

types will be useful for a more precise treatment of synchronization in infinite state systems. We also believe that it will be particularly useful for refinement checking [Roscoe 1999], where a model checker tries to find a trace T that can be performed by one system but not by the other. Such refinement checking can be encoded by the following clause:

```
not_refinement_of(S1,S2,T) :- trace(S1,T), \+(trace(S2,T)).
```

This clause is similar to the clause defining `sync_trace` and a non-conjunctive regular type analysis will face the same problems as above.

10. PROPAGATING SUCCESS INFORMATION

In this section we address one remaining limitation of our framework compared to existing top-down abstract interpretation approaches. Indeed, compared to the top-down abstract interpretation framework of [Bruynooghe 1991],

(1) our framework can use abstract *conjunctions* instead of abstract atoms, and can make use of sophisticated *abstract unfoldings* rather than just a single abstract resolution steps. Apart from producing more efficient specialised programs, these features sometimes allow for a more precise analysis [Leuschel and De Schreye 1996].

(2) on the other hand, there is no propagation or inference of *success* information in our framework. The following examples explains and illustrates this limitation.

Example 10.1 Consider the following tiny program:

$p(X) \leftarrow q(X) \wedge r(X)$
$q(a) \leftarrow$
$r(a) \leftarrow$
$r(b) \leftarrow$

Let us suppose we apply the instance of Algorithm 2 described in Section 9.1, i.e., classical partial deduction within the $\mathcal{PD}$-domain. For a given query $\leftarrow p(X)$, one possible (although very suboptimal) outcome of the algorithm is the final set $\mathcal{A}_i = \{p(X), q(X), r(X)\}$ of abstract conjunctions and the SLD-trees τ_1, τ_2 and τ_3 presented in Figure 6 (generated by *aunfold*).

With this result of the analysis, the transformed program is identical to the original one. Note that in τ_2 we have derived that the only answer for $\leftarrow q(X)$ is X/a. An abstract interpretation algorithm such as the one in [Bruynooghe 1991] would propagate this success-information to the leaf of τ_1, yielding that (under the left-to-right selection rule) the call $\leftarrow r(X)$ becomes more specific, namely $\leftarrow r(a)$. This information would then be used in the analysis of the $r/1$ predicate, allowing to remove the right branch of τ_3 and thus the clause generated from it. This clause is redundant, because for no concretisation of $\leftarrow p(X)$ will this clause appear in a successful refutation.
The same picture holds even if we add the clause

$q(X) \leftarrow q(X)$

---

enough to capture these traces. However, we believe that there will be more complicated system traces which it cannot precisely describe.

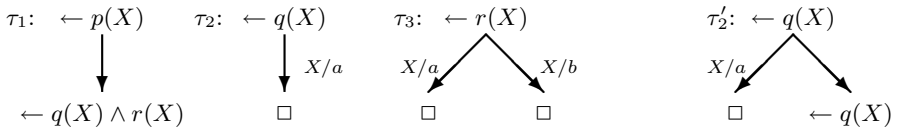

Fig. 6. SLD-trees for Example 10.1

to the above program, thus obtaining the tree τ_2' in Figure 6 instead of τ_2. Indeed, an abstract interpretation [Bruynooghe 1991] of $q(X)$ will return that the only possible computed answer substitution for $q(X)$ is $\{X/a\}$. Hence, assuming a left-to-right selection rule, the predicate $r/1$ will again only ever be called with its argument instantiated to a.

The possibility to do such sideways and bottom-up information passing can actually be relatively easily added to our framework.[12] In fact, all we have to do is replace Definition 6.2, defining the concretisation function γ for sequences of abstract conjunction, by the following definition:

Definition 10.2 Let $\langle \mathcal{AQ}, \gamma \rangle$ be an abstract domain. We define γ for sequences of abstract conjunctions in the context of a program P inductively as follows:

- $\gamma(\langle \mathbf{A_1} \rangle) = \gamma(\mathbf{A_1})$
- $\gamma(\langle \mathbf{A_1}, \ldots, \mathbf{A_n} \rangle) = \{ Q_s \wedge Q_n \mid Q_s \in \gamma(\langle \mathbf{A_1}, \ldots, \mathbf{A_{n-1}} \rangle),\ Q_n \preceq \lceil \mathbf{A_n} \rceil$ and $(P \models \forall(Q_s)) \Rightarrow Q_n \in \gamma(\mathbf{A_n}) \ \}$

Intuitively, for a conjunction $q(t) \wedge r(t)$ to be a concretisation of a sequence $\langle \mathbf{A_1}, \mathbf{A_2} \rangle$ of abstract conjunctions, the atom $r(t)$ must only be a concretisation of A_2 in case $P \models \forall(q(t))$, i.e., if $q(t)$ is a computed instance.

For example, in the $\mathcal{PD}$-domain and in the context of Example 10.1 we have $q(a) \wedge r(a) \in \gamma(\langle q(X), r(a) \rangle)$ but also $q(b) \wedge r(b) \in \gamma(\langle q(X), r(a) \rangle)$, as $P \not\models q(b)$. Similarly, we have $q(X) \wedge r(X) \in \gamma(\langle q(X), r(a) \rangle)$, as $P \not\models \forall X.q(X)$. Observe that neither $q(b) \wedge r(b)$ nor $q(X) \wedge q(X)$ are an element of $\gamma(q(X) \wedge r(a))$.

Using the revised Definition 10.2 we have that $\langle q(X), r(a) \rangle$ is an abstraction of $q(X) \wedge r(X)$ and Algorithm 2 can thus produce the outcome $\mathcal{A}_i = \{p(X), q(X), r(a)\}$ and sideways and bottom-up information passing has been achieved.

The change made in Definition 10.2 means that Theorems 8.2 and 8.7 will no longer hold for any SLD-refutation and finitely failed SLD-tree, but only for LD-refutations and finitely failed LD-trees (SLD-derivations and SLD-trees which follow a left-to-right selection rule are called LD-derivations and LD-trees respectively). Furthermore, the abstract unfolding operation will now have to satisfy the requirements of Definition 5.3 not for some SLD-tree τ but for some LD-tree τ.

Finally, it is possible to go even further and implement a stronger, selection rule independent, bottom-up success propagation, that would not only instantiate $r(X)$ to $r(a)$ in Example 10.1 but also instantiate $p(X)$ to $p(a)$. Abstract partial deduction could then produce the outcome $\mathcal{A}_i = \{p(a), q(a), r(a)\}$ and the specialised program:

[12] Another possible solution is to analyse the calls $q(X)$ and $r(X)$ in conjunction, thus achieving "perfect" success information passing. However, due to termination considerations this is not always possible or desirable.

$p(a) \leftarrow$
$q(a) \leftarrow$
$r(a) \leftarrow$

Details of this approach are sketched in [Leuschel 1998b]. A variation of this approach has been used in [Leuschel and Gruner 2001] to obtain a concrete specialization procedure and a practical implementation. We basically can instantiate the resultants using bottom-up success information. However, this specialization approach can change the termination characteristic of the program and no longer preserves the finite failure semantics, because infinite failure can be replaced by finite one.

11. MORE RELATED WORK

Abstract Interpretation of Logic Programs. Table I presents a brief comparison of how the specialization and abstract interpretation techniques discussed in the paper relate to each other. The abbreviations in the table for the column headings are as follows:

- PD: stands for classical partial deduction [Lloyd and Shepherdson 1991]
- CPD: denotes conjunctive partial deduction [De Schreye et al. 1999]
- MSV: this is the most specific version abstract interpretation of [Marriott et al. 1988; 1990]
- TD-AI: is the top-down abstract interpretation framework of [Bruynooghe 1991]
- Plai: is the already mentioned technique of [Puebla et al. 1997; Puebla et al. 1999] which extends an existing abstract interpreter for Prolog so that it produces specialised code. This can be seen as abstract partial deduction on atoms, using arbitrary abstract domains (provided by the abstract interpreter), and (contrary to [Puebla and Hermenegildo 1995; 1999]) it can use an abstract unfolding which performs more than one unfolding step.
- BU-AI: this classical bottom-up abstract interpretation based on approximating T_P and computing a fixpoint of this abstraction.
- APD: this is abstract partial deduction as developed in this paper up until Section 8.
- APD$^+$: this is the abstract partial deduction with success information propagation, as extended in Section 10.

The first row of Table I indicates which abstract domain can be used by the respective methods. The second row indicates whether the method can analyse conjunctions of atoms, while the third row indicates whether the method can make use of an unfolding rule. The fourth row indicates whether success information can be inferred and propagated, while the last row indicates the semantics on which the abstractions are based.

Specialization and Transformation of Logic Programs. We have already discussed in Section 9 the relationship of our abstract partial deduction framework (namely "more general than") with classical partial deduction [Lloyd and Shepherdson 1991;

[13] Only in the journal version [Marriott et al. 1990].

	PD	CPD	MSV	TD-AI	Plai	BU-AI	APD	APD$^+$
Abs. Domain	PD	PD	PD	any	any	any	any	any
Conjunctions	no	yes	yes[13]	no	no	no	yes	yes
Unfolding	yes	yes	no	no	yes	no	yes	yes
Success Info	no	no	yes	yes	yes	yes	no	yes
Semantics	SLD	SLD	T_P	And-Or	And-Or	T_P	SLD	SLD+T_P

Table I. A comparison of program specialization and abstract interpretation techniques

Gallagher 1993], conjunctive partial deduction [Leuschel et al. 1996; Glück et al. 1996; Leuschel 1997], ecological partial deduction [Leuschel 1995; Leuschel et al. 1998; Leuschel 1997] (and its ancestor [Gallagher and Bruynooghe 1991]), constrained partial deduction [Leuschel and De Schreye 1998], and partial deduction with removal of useless clauses [de Waal and Gallagher 1991; Gallagher and de Waal 1992; de Waal and Gallagher 1994].

The following techniques in the functional/logic setting, are also closely related. [Gallagher and Lafave 1996] presents a variation of ecological partial deduction for functional and logic languages, using trace terms instead of characteristic trees. [Lafave and Gallagher 1997] is a technique in the style of constrained partial deduction for functional-logic programs. [Alpuente et al. 1998; Albert et al. 1998] can be viewed as a conjunctive partial deduction technique (i.e., abstract partial deduction in the classical $\mathcal{PD}$-domain) for functional-logic languages.

Another, strongly related work is [Pettorossi and Proietti 1996b], which uses an unfold/fold program transformation approach to specialise logic programs in a given *context*. This context is another predicate of the logic program under consideration. In contrast to our general technique, [Pettorossi and Proietti 1996b] performs syntactic transformations only, and has a more limited abstract unfolding possibility. Also, the side-condition has to be expressed as a logic program predicate, i.e., it may not be obvious how to easily handle characteristic trees from ecological partial deduction or more general constraints. Finally, the results of [Pettorossi and Proietti 1996b] are for the least Herbrand model semantics and not (yet) for computed answer or finite failure semantics. Nonetheless, it should be possible to cast [Pettorossi and Proietti 1996b] (or a suitably adapted version thereof) in our framework and thus gain correctness results for computed answers and finite failure.

Functional Programming. Supercompilation [Sørensen et al. 1996; Glück and Sørensen 1994], is very related to conjunctive partial deduction (in fact, conjunctive partial deduction was in part inspired by supercompilation). Indeed, the abstract domain for supercompilation can be seen as the concrete domain of functional programming expressions augmented with variables (which already exist in the concrete domain of logic programming). Tupling [Chin and Khoo 1993], deforestation [Wadler 1990], and generalized partial computation [Futamura et al. 1991] are also closely related to conjunctive partial deduction (see [De Schreye et al. 1999; Leuschel 1999], [Sørensen et al. 1994]) and thus abstract partial deduction in the "classical" $\mathcal{PD}$-domain. We believe that it is possible to adapt the present paper to a functional programming setting, thus making it possible to extend the above techniques to use richer, more expressive abstract domains.

One of the earliest combinations of abstract interpretation and partial evalua-

tion has been developed by Consel and Khoo [Consel and Khoo 1993]. They give a framework for a first-order functional language parametrised on algebras. Another related functional programming technique is type specialization [Hughes 1999]. It already uses a domain based upon types, richer than the $\mathcal{PD}$-domain. It is still unclear whether a logic programming version of type specialization can be developed, and whether it can then be cast into our framework.

Imperative Programming. [Jones 1999] presents a very generic framework which can model various (non-conjunctive) partial evaluation and driving techniques in the context of imperative programs. It has a concept of abstract stores, which represent sets of possible concrete stores of the imperative program. The paper also contains soundness and completeness criteria, and clarifies the relationship between partial evaluation and driving (i.e., supercompilation). However, in contrast to our paper, it has more limited abstract unfolding: in essence every abstract unfolding step must correspond to exactly one concrete step (there is, however, a post-processing compression phase of transient transitions).

12. FUTURE WORK AND CONCLUSION

Future Work. A lot of avenues can be pinpointed for further work. First, on the practical side, one should of course implement further, useful instances of the generic algorithms presented in this paper. [Gallagher and Peralta 2000; 2001] and [Leuschel and Gruner 2001] have already developed instances of our framework based upon regular types, and some promising applications for infinite state model checking of process algebras have been hinted at. These techniques can probably be further improved, by using the possibilities opened up by our very general definition of abstract unfolding (cf., Section 5). It should also be possible to move to more precise abstract domains, such as non-deterministic regular types [Gallagher and Puebla 2002] without too much difficulty.

New abstract partial deduction techniques, based upon other abstract domains from the abstract interpretation literature also look very promising for specialization purposes.

On the theoretical side, one could try to extend the language treated by our framework. We can already handle definite logic programs with declarative builtins such as *is*, *call*, *functor*, *arg*, $\backslash ==$. This allows to express a large number of interesting, practical programs; one can even implement and use certain higher-order features such as *map*/3. However, we cannot yet handle normal logic programs with negation or constraint logic programs, and one should strive to extend our framework to handle such programs. Ideally, one should aim at making our framework programming language independent and thus not only covering normal and constraint logic programs, but functional and imperative programs as well. This would provide a unified correctness framework in which most specialization techniques could be cast.

One can also endeavor to cover ever more powerful, but ever more difficult to automate, specialization methods such as goal replacement, specialization of disjunctions of conjunctions [Pettorossi et al. 1997] or specialization of conjunctions of unlimited length [Pettorossi and Proietti 1996a].

Conclusion. In this paper we have presented a very generic framework for top-down logic program specialization. We have established several *generic correctness results* and have cast several existing techniques in our framework, thereby re-using the correctness results in a simple manner. We have also shown how the additional generality of our framework can be exploited in practice, for improved generalisation, unfolding and code-generation. Instances of our framework, based upon regular types, have already been developed in the literature and their usefulness has been demonstrated. In the course of this paper, we have also clarified the relationship of top-down partial deduction with abstract interpretation, establishing a *common basis* and terminology. We believe we have made an important step towards a full reconciliation of abstract interpretation and program specialization. In summary, the new framework with its generic algorithm and correctness results, provides the foundation for new, powerful specialization techniques.

ACKNOWLEDGEMENTS

Maurice Bruynooghe provided very valuable feedback this paper. The author also greatly benefited from discussions with Danny De Schreye, John Gallagher, Stefan Gruner, Neil Jones, Jesper Jørgensen, Helko Lehmann, Bern Martens, Torben Mogensen, Germán Puebla, Jens-Peter Secher, Morten Heine Sørensen, and comments of anonymous referees of JICSLP'98. Finally, I am very greatful to the anonymous referees of ACM Toplas; their detailed comments and constructive criticisms have substantially helped me to improve the article.