ASAP

IST-2001-38059

Advanced Analysis and Specialization for Pervasive Systems

Integrated Abstract Interpretation and Online Specialization

Deliverable number:	D4
Workpackage:	Basic Specialization Techniques (WP3)
Preparation date:	1 November 2003
Due date:	1 May 2003
Classification:	Public
Lead participant:	Univ. of Southampton
Partners contributed:	Univ. of Southampton, Tech. Univ. of Madrid (UPM), Roskilde Univ

Project funded by the European Community under the "Information Society Technologies" (IST) Programme (1998–2002).

Short description:

In the first part of the deliverable, we present a theoretical framework that 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 framework opens up the way for new optimizations and enables a simpler correctness-proving of specialization techniques. This part will appear in May 2004 in the ACM Transactions on Programming Languages and Systems.

The second part describes *abstract specialization* which is at the heart of the specialization system in `CiaoPP`. We discuss its potential applications, which include program parallelization, optimization of dynamic scheduling (concurrency), and integration of partial evaluation techniques. This part is based on an invited talk at PEPM'03.

In the third part we present an abstract domain based on regular types for its usage in top-down abstract interpretation and present a new widening which is more precise than those previously proposed while being efficient. This part is based on a paper presented at SAS. Also in this part an abstract specialization algorithm using an abstract domain based on convex hulls is described. The core algorithm is based on the framework described in the first part, propagating both abstract calls and answers. This is based on a paper in LOPSTR'02.

In the fourth part we present some insights into using program specialization (with abstract interpretation) as applied to inductive theorem proving and model checking, which will be of importance when applying our technique to the tasks in workpackage 5 later in the project. This part is based on an invited paper at LOPSTR'03.

The fifth part describes a new class of abstract domains which appears to be promising for both online and offline specialization (and has already been applied in the BTA algorithm, see part V of Db5). These domains are constructed from regular types, by converting the given types into disjoint types. The resulting finite domain is condensing, which suggests more efficient propagation of answers and calls.

Finally, the sixth part describes an algorithm which extends abstract interpretation by including partial deduction capabilities. This allows improving both existing analysis and specialization techniques.

Contents

Part I

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

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 part of the document 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, as well as proving 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.

1 Background

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 [19] transformation framework. Program specialization encompasses traditional compiler optimization techniques [163], 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 ap-

proach, 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* [35] 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., [33, 184, 143, 102, 182, 192, 188, 74]).

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., [66, 44, 143, 135] to name just a few) and a lot of extensions have been developed to remedy its shortcomings (such as partial deduction with characteristic trees [60, 140], constrained partial deduction [128], conjunctive partial deduction [129, 79, 42]). 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 [24] and deforestation [215] 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 [123].

2 Basics of Partial Deduction

In this section we present the technique of partial deduction, which originates from [111]. Other introductions to partial deduction can be found in [112, 57, 125]. Note that, for clarity's sake, we deviate slightly from the original formulation of [152] and use the formulation from [127]. 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 [31, 106, 101, 161] 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 [4, 150], 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 [4, 150]. 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 [4, 150] 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*, $resultants(\tau)$, is the set whose elements are the resultants of $D_1, \ldots, D_n$ and the *set of leaves*, $leaves(\tau)$, 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 $resultants(\tau)$ 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 [119], which counts resolution steps:

$$solve([], Depth, Depth) \leftarrow$$

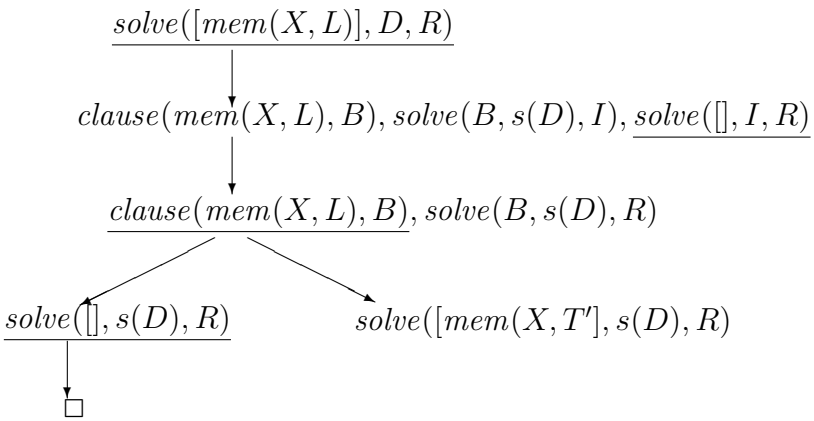


Figure 1: Incomplete SLD-tree for Example 2.5

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

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 $resultants(\tau)$ 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 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*.

[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 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 [152] 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.

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., [127]).

In 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 [6]. To improve the efficiency of specialised programs, all partial deduction systems we know of, perform renaming together with so-called *filtering* [59, 60, 145, 177], 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 [41] 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.

3.2 Abstract Interpretation

Abstract interpretation [35] provides a general formal framework for performing sound program analysis and has been successfully applied to the analysis of logic programs [36, 14, 91]. To

[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.

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 [35, 36]) 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., [55, 57, 127]) of controlling classical partial deduction [152].

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 $unfold$ (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, [127]), 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

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

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 $calls(P, Q)$; 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: $calls(P, Q) = split(R_P \uparrow^\omega (\{Q\}))$.

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

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

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. It is straightforward to show (e.g., using Lemma 4.12 from [152]) that in the above definition and for single step unfolding, we can replace the condition $C \in unfold(P, A)$ 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 $resolve(C, A)$. As usual in abstract interpretation, we lift this concrete operation to sets of atoms: $resolve^*(C, S) = \{resolve(C, A) \mid A \in S\}$. The abstract counterpart in R_P^α is simply $resolve_\alpha(C, A) = resolve(C, A)$, which is a sound approximation of $resolve$, i.e., $\gamma_{inst}(resolve_\alpha(C, A)) \supseteq resolve^*(C, \gamma_{inst}(A))$. 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 $\gamma_{inst}(resolve_\alpha(C, A))$ 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 $\gamma_{inst}^\wedge$ which acknowledges the fact that conjunctions can be split up and which is defined by

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

For $\gamma_{inst}^\wedge$, $split$ is a generalization operation: $\gamma_{inst}^\wedge(split(S)) \supseteq \gamma_{inst}^\wedge(S)$, and so is $generalize$: $\gamma_{inst}^\wedge(generalize(S)) \supseteq \gamma_{inst}^\wedge(S)$. 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 $\gamma_{inst}^\wedge$, in the sense that: $\gamma_{inst}^\wedge(\mathcal{A}_i) = \gamma_{inst}^\wedge(R_P^\alpha(\mathcal{A}_i))$. 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^{α}_{P}: $\gamma^{\wedge}_{inst}(\mathcal{A}) = \gamma^{\wedge}_{inst}(R^{\alpha}_{P}(\mathcal{A}))$.

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

$$R_P(\gamma^{\wedge}_{inst}(A)) \subseteq \gamma^{\wedge}_{inst}(R^{\alpha}_{P}(A))$$

In other words, R^{α}_{P} 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^{α}_{P} 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., [42]) $partition$ 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 $partition(\{q(X) \wedge p(X) \wedge r(Z)\})$ = $\{p(X) \wedge q(X), r(Z)\}$.

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.

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

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 [182, 192]):

- 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 [42] 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 [10] and to some extent also [155].

- 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 [154, 155] 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., [107] and [29]). A more popular semantics for top-down abstract interpretation is based on And-Or trees [14, 91, 97, 167, 115], where it is easier to capture and propagate success information.

The various limitations of partial deduction have been realized by many researchers (e.g., [43, 66, 44, 184, 117, 138, 143, 182, 192]), and various extensions of partial deduction have been developed over the years (e.g., [60, 128, 143, 140, 74]) 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.

[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.

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:

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., [166] 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 $\perp$ whose concretisation is empty.

One particular abstract domain, which arises in the formalization of (classical) partial deduction [152] 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 $\exists R \in resultants(\tau)$ 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 [109] or [47]), 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:

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

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

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 $aunfold(P, \mathbf{A})$. 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 $aunfold(P, \mathbf{A})$. Some of these points are illustrated in Figure 2 below (where $s_1 = \langle L, \theta\rangle$ and $s_2 = \langle L', \theta'\rangle$).

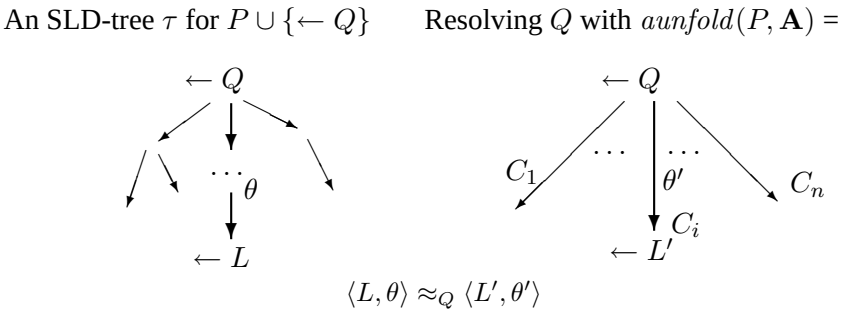


Figure 2: One-to-one correspondence of conditional computed answers for abstract unfolding

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 $\mathbf{A} \in \mathcal{AQ}$, $C_i \in aunfold(P, \mathbf{A})$, and $Q \in \gamma(\mathbf{A})$:

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

Point 3 requires that $\mathbf{A_i} = aresolve(\mathbf{A}, C_i)$ 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 $aunfold(P, A) = resultants(\tau')$ 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 [127]: termination (i.e., building finite SLD-trees), achieving good specialization and avoiding slowdowns. To ensure termination, well-founded measures [16, 156] and well-quasi-orders can be used [198, 8]. The well-quasi orders based on the homeomorphic embedding relation [203, 122] have recently been very popular. To avoid slowdowns, determinacy [60, 55], 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 [127] 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 $aunfold(P, \mathbf{A}) = resultants(\tau)$. This always satisfies 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.

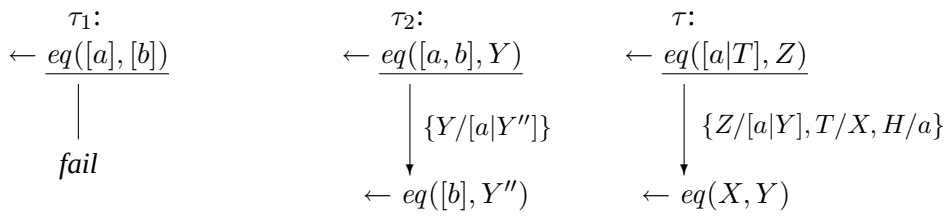


Figure 3: SLD-trees for Example 5.5

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 [55] or the most specific versions of [154, 155]. 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 dominator for $\mathbf{Q}$. Let τ be a SLD-tree for $P \cup \{\leftarrow Q\}$ and let $R \subseteq resultants(\tau)$ be a set of resultants such that for all resultants $Q\theta \leftarrow B \in (resultants(\tau) \setminus R)$ we have that no instance of $Q\theta$ is in $\gamma(\mathbf{Q})$. Then $aunfold(P, \mathbf{Q}) = R$ satisfies Definition 5.3.

Proof (Sketch) Let us first assume that $R = resultants(\tau)$, i.e., $aunfold(P,\mathbf{Q})$ = $\{Q\theta_1 \leftarrow B_1, \ldots, Q\theta_k \leftarrow B_k\}$ 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 [152].

– Point 2: This is a direct corollary of Lemma 4.9 in [152] (cf., proof of Lemma 8.3 for more details).

– Point 4: Take $Q' = Q$. This will unify with all $Q\theta_i$ via $mgu\ \sigma$ 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 $R \subset resultants(\tau)$ we only have to re-check Point 1. We can deduce that the head H of every resultant $C \in (resultants(\tau) \setminus R)$ 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 [152] 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 $aunfold(P, \mathbf{A})$ = $\{C_1, C_3\}$, $aresolve(\mathbf{A}, C_1) = \Box$ and $aresolve(\mathbf{A}, C_3) = \mathbf{A}$ 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 [43, 66, 44] (which detects and removes redundant clauses as a post-processing).

[73, 74] and [131] show how such abstract unfoldings can be developed for a particular abstract domain based upon regular types. [131] 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 $aunfold(P, \mathbf{A})$; e.g., if $A = p(f(X), Z)$ simply produce $aunfold(P, \mathbf{A}) = \{p(X, Y) \leftarrow\}$.

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 $A_1 \rightsquigarrow_{C_1} \langle \Box, \{\} \rangle$ and $A_2 \rightsquigarrow_{C_1} \langle \Box, \{\} \rangle$ 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 $eq(a, [])$.

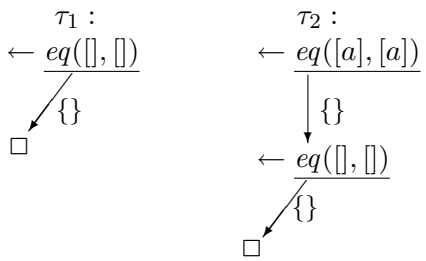


Figure 4: SLD-trees for Example 5.8

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 [184, 186, 188]. 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 [180, 178].

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 [186, 188] 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 $P' = \{C \mid C \in aunfold(P, \mathbf{A}) \wedge \mathbf{A} \in \mathcal{A}\}$. 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^{α}_{P}, 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^{α}_{P} operator from Section 3.3 into an operator R^{A}_{P} mapping sets of abstract conjunctions to sets of abstract conjunctions in the following way:

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

Intuitively, $R^{A}_{P}(\mathcal{A})$ 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 $\gamma(R^A_P(\mathcal{A})) \subseteq \gamma^\wedge(\mathcal{A})$, 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 $R^A_P(\mathcal{A})$ 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 $\mathcal{A} = \{\mathbf{A_1}, \mathbf{A_2}\}$, $R^A_P(\mathcal{A}) = \{\mathbf{A_1}\}$, with $aunfold(P, \mathbf{A_1}) = \{p(f(X)) \leftarrow p(X)\}$ and $aunfold(P, \mathbf{A_2}) = \{p(g(X)) \leftarrow\}$ 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 $\gamma(\mathcal{A}) = \gamma(R^A_P(\mathcal{A}))$ but it would be impossible to perform a renaming transformation in the classical sense, as we cannot decide whether the call $p(X)$ within $aunfold(P, \mathbf{A_1})$ 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.

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

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

let $\mathcal{A}_{i+1} := ageneralize(R^A_P(\mathcal{A}_i))$; **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 [157] 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 $msg(E)$, is the most specific expression M such that all expressions in E are instances of M. The *msg* can be effectively computed [114] and given an expression A, there are no infinite chains of strictly more general expressions [93]. More refined widenings, are based upon well-founded orders, well-quasi orders and characteristic trees (see, e.g, [60, 140, 122], see also [127]).

[73, 74] and [131] 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 [176] (also called deforestation and tupling).

To overcome this limitation in the setting of classical partial deduction, [42] 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., [127]).

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 [79, 42]. For abstract conjunctive partial deduction, this has to be combined with widenings from the abstract interpretation literature. [131] 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 [42] 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:

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

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).

[7] There actually also exists a most specific concrete dominator (by existence of a most specific generalisation msg of two terms [114] and the fact that the strictly more general relation is a well-founded order [93], 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).

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 $H_i \leftarrow B_i \in \mathit{aunfold}(P, \mathbf{A})$ 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 $\mathit{aunfold}$). If our flow analysis also contains $\mathbf{A_i}$ = $\mathit{aresolve}(\mathbf{A}, H_i \leftarrow B_i)$ (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 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.

[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.

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 $msg(\gamma(\mathbf{A}))$, 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 [59] (as well as the one from [145]) 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).

8.1 Correctness for Computed Answers

For technical reasons we have to introduce the concept of admissible renamings (as in [128]).

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 $P \cup \{\leftarrow Q\}$.

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

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

- $Q \rightsquigarrow_\tau \langle L, \theta\rangle \Rightarrow Q' \rightsquigarrow_{\tau'} \langle L', \theta'\rangle$ with $Q\theta \leftarrow L \approx Q'\theta' \leftarrow L'$
- $Q' \rightsquigarrow_{\tau'} \langle L', \theta'\rangle \Rightarrow Q \rightsquigarrow_\tau \langle L, \theta\rangle$ 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 [152] 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 [152]). □

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):

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

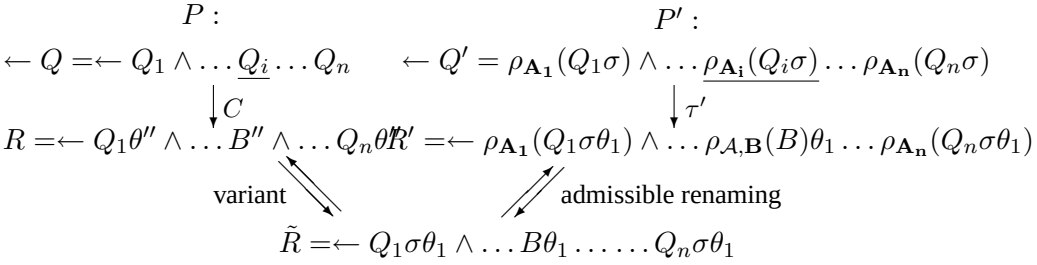


Figure 5: Illustrating the proof of Theorem 8.2

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

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

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 $\mathbf{B} = aresolve(\mathbf{A_i}, C)$ 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 [152]):

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

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

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

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 ([4, 150]) 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 $mgu\ \theta_1$ 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.

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 [152], all failed SLD-trees for the resolvents into one big finitely failed SLD-tree for $P \cup \{\leftarrow Q\}$. □

[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.

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 [199] 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* [129, 120]. 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 [150][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 [152], 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 [152, 57] 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 [129, 79, 120] 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

[43, 66, 44] 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 [117, 140, 120] (and its ancestor [60]) 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 [60, 55].
- $\gamma((A, \tau))$ = $\{A'' \mid A'' \preceq A' \preceq A \wedge A'$ 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 [117, 140, 120]).
- $aresolve((A, \tau), A\theta \leftarrow B)$ = (B, τ') 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 [128] 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 [128] 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 [217, 67] 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 [182, 192].

Instances of our abstract partial deduction framework using regular types have recently been developed. First, [73, 74] 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, [131] presents an extension of [73, 74] 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 [119] has been developed and applied to several examples; one of which we elaborate below. One possible application of the method is the model checking [26] 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 $success_R(t)$ = $\{s \mid$ 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 [74].

Full details on how to implement abstract unfolding, abstract resolution and concrete abstract partial deduction procedures can be found in [73, 74] and [131].

The following example, which was worked out using the implementation presented in [131], 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 [155]) 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 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 [195], 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 [14],

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 [143].

[11] The non-deterministic regular type analysis of [76] actually is precise enough to capture these traces. However, we believe that there will be more complicated system traces which it cannot precisely describe.

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 [14] 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.

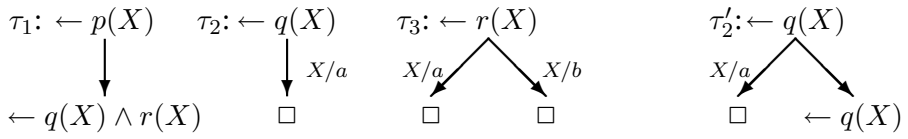


Figure 6: SLD-trees for Example 10.1

The same picture holds even if we add the clause

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

to the above program, thus obtaining the tree τ_2' in Figure 6 instead of τ_2. Indeed, an abstract interpretation [14] 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:

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

[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.

Details of this approach are sketched in [123]. A variation of this approach has been used in [131] 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 1 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 [152]
- CPD: denotes conjunctive partial deduction [42]
- MSV: this is the most specific version abstract interpretation of [154, 155]
- TD-AI: is the top-down abstract interpretation framework of [14]
- Plai: is the already mentioned technique of [182, 192] 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 [184, 188]) 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 1 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")

[13] Only in the journal version [155].

	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 1: A comparison of program specialization and abstract interpretation techniques

with classical partial deduction [152, 57], conjunctive partial deduction [129, 79, 120], ecological partial deduction [117, 140, 120] (and its ancestor [60]), constrained partial deduction [128], and partial deduction with removal of useless clauses [43, 66, 44].

The following techniques in the functional/logic setting, are also closely related. [69] presents a variation of ecological partial deduction for functional and logic languages, using trace terms instead of characteristic trees. [113] is a technique in the style of constrained partial deduction for functional-logic programs. [3, 2] 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 [172], 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, [172] 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 [172] are for the least Herbrand model semantics and not (yet) for computed answer or finite failure semantics. Nonetheless, it should be possible to cast [172] (or a suitably adapted version thereof) in our framework and thus gain correctness results for computed answers and finite failure.

Functional Programming Supercompilation [205, 81], 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 [24], deforestation [215], and generalized partial computation [54] are also closely related to conjunctive partial deduction (see [42, 125], [204]) 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 evaluation has been developed by Consel and Khoo [33]. They give a framework for a first-order functional language parametrised on algebras. Another related functional programming technique is type specialization [94]. 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 [105] 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. [73, 74] and [131] 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 [76] 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 built-ins 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 [175] or specialization of conjunctions of unlimited length [171].

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.

Part II

Abstract Specialization and its Applications

The aim of program specialization is to optimize programs by exploiting certain knowledge about the context in which the program will execute. There exist many program manipulation techniques which allow specializing the program in different ways. Among them, one of the best known techniques is *partial evaluation*, often referred to simply as program specialization, which optimizes programs by specializing them for (partially) known input data. In this work we describe *abstract specialization*, a technique whose main features are: (1) specialization is performed with respect to "abstract" values rather than "concrete" ones, and (2) *abstract interpretation* rather than standard interpretation of the program is used in order to propagate information about execution states. The concept of abstract specialization is at the heart of the specialization system in `CiaoPP`, the `Ciao` system preprocessor. In this work we present a unifying view of the different specialization techniques used in `CiaoPP` and discuss their potential applications by means of examples. The applications discussed include program parallelization, optimization of dynamic scheduling (concurrency), and integration of partial evaluation techniques.

13 Background

The aim of program optimisation is, given a program P to obtain another program P' which is semantically equivalent to P but behaves better for some criteria of interest. One typical way of optimizing programs is by *specializing* them for some particular context. This allows automatically overcoming losses in performance which are due to general purpose algorithms. This situation is becoming more and more frequent due to the use of techniques such as reuse of general-purpose programs and libraries, and software components, which facilitate development but can result in large programs and even waste of computing resources. More precisely, the aim of program specialization is, given a program P and certain knowledge ϕ about the context in which P will be executed, to obtain a program P_ϕ which is equivalent to P for all contexts which satisfy ϕ and which behaves better from some given point of view.

In the case of *partial evaluation* [32, 99], the knowledge ϕ which is exploited is the so-called *static* data, which corresponds to (partial) knowledge at specialization (compile) time about the input data. Data which is not known at specialization time is called *dynamic*. The program is

optimised by performing at specialization time those parts of the program execution which only depend on static data.

Another very general setting for specialization specially relevant in the context of logic programs, which has been proposed in [173], is to define the knowledge about the context as a so-called *static property* $\phi(X_1, \ldots, X_n)$, where $X_1, \ldots, X_n$ are the formal arguments of the top-level procedure P and ϕ is defined as a logic program. However, this approach suffers from an important difficulty in using the context information in an automated and effective way.

The approach we follow in *abstract specialization* is that the information ϕ available on the context is captured by an *abstract substitution*. One advantage of this approach is that there are well known techniques which allow handling information represented as abstract substitutions by using *abstract interpretation* techniques [35].

13.1 An Overview of Specialization Techniques

For the purpose of comparing different existing techniques, let us classify the existing specialization techniques according to how the final, optimised, program is obtained. Of course, this classification is rather crude and many of the existing techniques can be seen as a combination of the three approaches which we will discuss. The first approach which we describe, and which we will call *program with annotations*, consists of two phases. During the first phase, some *static program analysis* technique is used in order to annotate the program with analysis information. In the second phase, the program is optimised using the information obtained. This approach is conceptually simple, though either or both of the phases mentioned can indeed be rather complex. A well known example of this kind of techniques is the "off-line" approach to partial evaluation, in which a *binding-time analysis* phase is followed by another one in which the residual program is generated.

The second class of techniques we consider, which we will call the *transformational* approach, is based on program transformation techniques, such as fold/unfold transformations (such as the ones developed in [20, 206]). In this scheme, a series of n semantic-preserving program-transformation steps are performed such that initially $P = P_0$. Then, each P_{i+1} is obtained from P_i by applying some transformation T_i, i.e., $P_{i+1} = T_i(P_i)$, which preserves the semantics of the program. Finally $P' = P_n$. Transformational techniques are very powerful, the main difficulty being in automatically deciding a proper sequence of programs transformations to perform in order to obtain (an optimal) program P'.

The third and last possibility which we consider, and which we will denote the *semantic* approach, is based on the existence of an algorithm $\mathcal{S}$ which, given a program P and some knowledge ϕ, builds a semantic representation of the program $\mathcal{S}(P, \phi)$ which captures the behaviour

of P in some precise way for all contexts which satisfy ϕ. Then, there is a code generation algorithm which builds the program P_ϕ from $\mathcal{S}(P, \phi)$ in a straightforward way. Often this semantic representation can be seen as a graph. The kind of graph obtained depends on the particular semantics used by the algorithm. The "on-line" approach to partial evaluation is, in our terminology, a semantic approach since the behaviour of the program is precisely captured by the partial evaluation algorithm.

A particular algorithm for the on-line partial evaluation of logic programs is *partial deduction* [148, 110]. Though on-line partial evaluation can be considered an instance of fold/unfold transformations, the comparatively significant success of partial deduction techniques is probably due to the fact that they are often formalized as a semantic approach. I.e., an algorithm exists which can be used to build the semantic representation of the program. The existing algorithms for partial deduction [148, 57, 120] are parameterized by different control strategies. Usually, control is divided into components: "local control," which controls the unfolding for a given atom, and "global control," which ensures that the set of atoms for which a partial evaluation is to be computed remains finite. Several strategies for global and local control have been proposed which produce good-quality partial evaluations of programs [158, 139]. Regarding the correctness of partial deduction, two conditions, defined on the set of atoms to be partially evaluated, have been identified which ensure correctness of the transformation: "closedness" and "independence" [148].

13.2 Abstract Specialization through A Motivating Example

One of the distinguishing features of logic programming (LP) is that arguments to procedures can be uninstantiated variables. This, together with the search execution mechanism available (generally backtracking) makes it possible to have *multi-directional* procedures. I.e., rather than having fixed input and output arguments, execution can be "reversed". Thus, we may compute the "input" arguments from known "output" arguments.

Example 13.1 *Consider the logic program below. As usual in LP,* predicates *(procedures) are referred to in the text as* `name/arity`*, where arity is the number of arguments of the predicate. The predicate* `ground/1` *is a boolean test which succeeds if and only if its argument is bound at run-time to a term without variables, and the predicate* `is/2` *(used as an infix binary operator) computes the arithmetic value of its second (right) argument and unifies it with its first (left) argument.*

```
plus(X,Y,Z):- ground(X),ground(Y),!,Z is X + Y.
plus(X,Y,Z):- ground(Y),ground(Z),!,X is Z - Y.
plus(X,Y,Z):- ground(X),ground(Z),!,Y is Z - X.
```

The procedure `plus/3` *defines the relation such that the third argument is the addition of the first and second arguments. The procedure* `plus/3` *is multi-directional. For example, the call* `plus(1, 2, Sum)` *can be used to compute the addition of 1 and 2. Also, the call* `plus(Num, 2, 3)` *can be used to determine which is the number* `Num` *such that when added to 2 returns 3.*

Thus, the definition of `plus/2` behaves *declaratively* as long as at least two of the input arguments are ground. However, this good behavior of `plus/3` when compared to a mono-directional operation such as `is/2` is at the expense of some overhead which is incurred at run-time in order to select the appropriate clause to execute out of the three existing ones. Imagine now that at compile-time it is known that the call to `plus/3` will be of the form `plus(1, 2, Sum)`. In such case it is clear that the first clause will be selected and the execution will return the value 3 for the argument `Sum`. This is a typical example of an execution which can benefit from (traditional, "concrete") partial evaluation where ϕ is the knowledge that the initial call is `plus(X,Y,Z)` with `X=1` and `Y=2`.

In spite of the relative maturity of partial evaluation of logic programs, it is well known that the technique has certain shortcomings. Imagine we are interested in optimizing the code:

```
p(X,Y,Res):- plus(X,Y,Tmp), plus(1,Tmp,Res).
```

where `plus/3` is defined as above. By observing the program we can conclude that after the execution of the call `plus(X,Y,Tmp)` all three arguments are ground. As a result, the call `plus(1,Tmp,Res)` can be optimised to `Res is 1 + Tmp`.

Unfortunately, in traditional partial evaluation no information on the value of the argument `Tmp` is propagated to the call `plus(1,Tmp,Res)`. The intrinsic problem underlying this shortcoming of partial evaluation is that the only information which can be captured about values of arguments are *concrete* values. In the case of logic programming, values are captured by *substitutions*. This shortcoming of partial evaluation has been identified and several proposals exist which try to overcome it. Our proposal, *abstract specialization*, addresses this problem directly. Abstract specialization allows specializing calls with respect to *abstract substitutions* instead of *concrete substitutions* as in traditional partial evaluation. As will be discussed in Section 14, abstract substitutions are in this context finite representations of possibly infinite sets of data. Each such representation method is called an *abstract domain*. The kind of information which can be captured by abstract substitutions varies from one abstract domain to another. For example, we can have an abstract domain which allows capturing type information.[14] Such domain can be

[14] Alternatively we could use an abstract domain which captures groundness information natively and obtain the same optimised program.

used to determine that in the call `plus(1,Tmp,Res)` the argument `Tmp` is bound to a number. We can use this information in order to *abstractly execute* the two ground terms in the first clause of `plus/3` to the value *true*. We can even execute the `!/0` procedure call and eliminate the rest of clauses for `plus/3`[15]. We can thus optimise the original program to:

```
p(X,Y,Res):- plus(X,Y,Tmp), Res is 1 + Tmp.
```

Also, the call `plus(X,Y,Tmp)` can be optimised. Since `Tmp` is a variable which is local to the clause, it can be determined to be a free variable (and thus definitely not ground). Thus, the program can be optimised to:

```
p(X,Y,Res):- plus1(X,Y,Tmp), Res is 1 + Tmp.
plus1(X,Y,Z):- ground(X),ground(Y),!,Z is X + Y.
```

where `plus1/3` is a specialized version of `plus/3`. Generalizing from the examples above we can develop a specialization system which is able to perform the optimisations shown above. The specialization system will be able to: (1) capture more general information than traditional substitutions, i.e., it will capture abstract substitutions, (2) propagate such information in a correct way using a suitable semantics, and (3) carry out the optimisations enabled by the information available.

14 Abstract Interpretation

Static Program analysis aims at deriving at compile-time certain properties of the run-time behavior of a program. We provide some background and notation on abstract interpretation [35], which is arguably one of the most successful techniques for static program analysis.

In abstract interpretation, the execution of the program is "simulated" on an *abstract domain* (D_α) which is simpler than the actual, *concrete domain* (D). An abstract value is a finite representation of a, possibly infinite, set of actual values in the concrete domain (D). The set of all possible abstract semantic values represents an abstract domain D_α which is usually a complete lattice or cpo which is ascending chain finite. However, for this study, abstract interpretation is restricted to complete lattices over sets both for the concrete $\langle 2^D, \subseteq \rangle$ and abstract $\langle D_\alpha, \sqsubseteq \rangle$ domains.

Abstract values and sets of concrete values are related via a pair of monotonic mappings

[15]The procedure call `!/0` is used to eliminate other alternatives.

$\langle\alpha, \gamma\rangle$: *abstraction* $\alpha : 2^D \to D_\alpha$, and *concretization* $\gamma : D_\alpha \to 2^D$, such that

$$\forall x \in 2^D : \ \gamma(\alpha(x)) \supseteq x \quad \text{and} \quad \forall y \in D_\alpha : \ \alpha(\gamma(y)) = y.$$

Note that in general $\sqsubseteq$ is induced by $\subseteq$ and α (in such a way that $\forall \lambda, \lambda' \in D_\alpha : \ \lambda \sqsubseteq \lambda' \Leftrightarrow \gamma(\lambda) \subseteq \gamma(\lambda')$). Similarly, the operations of *least upper bound* ($\sqcup$) and *greatest lower bound* ($\sqcap$) mimic those of 2^D in some precise sense.

Example 14.1 (A domain for mode analysis) *Consider the following toy abstract domain D_α which captures* mode *information (i.e., the state of instantiation of program variables upon procedure call). An abstract substitution λ over a set of variables $\overline{X} = \{X_1, \ldots, X_n\}$ assigns to each variable X_i a value v in the set $\{ground, var, any\}$ where each v represents an infinite set of terms. The fact that a variable X_i is assigned an abstract value v indicates that X_i will be bound at run-time to some term belonging to v. $ground$ is the set of all terms without variables; var is the set of unbound variables (possibly aliased to other unbound variables); and any is the set of all terms. The abstract domain is complemented by the abstract substitutions $\bot$ and $\top$. As usual in abstract interpretation, $\bot$ denotes the abstract substitution such that $\gamma(\bot) = \emptyset$. The substitution $\top$ is such that $\gamma(\top) = D$. In our domain, $\top$ corresponds to assigning any to each variable in $\overline{X}$.* □

Since our discussion will concentrate on logic programs, we also recall some classical definitions in logic programming. An *atom* has the form $p(t_1, ..., t_n)$ where p is a predicate symbol and the t_i are terms. We often use $\bar{t}$ to denote a tuple of terms. A *clause* is of the form H `:-` $B_1, \ldots, B_n$ where H, the *head*, is an atom and $B_1, \ldots, B_n$, the *body*, is a possibly empty finite conjunction of atoms. Atoms in the body of a clause are often called *literals*. A *definite logic program*, or *program*, is a finite sequence of clauses.

14.1 Goal-Dependent analysis

Goal-dependent analyses are characterized by generating information which is valid only for a restricted set of calls to a predicate, as opposed to goal-independent analyses whose results are valid for any call to the predicate. Goal-dependent analyses allow obtaining results which are *specialized* (restricted) to a given context. As a result, they provide in general better (stronger) results than goal-independent analyses. In addition, goal-dependent analyses provide information on both the call and success states for each predicate, whereas goal-independent analyses in principle only provide information on success states of predicates. For these reasons, and since program specialization greatly relies on information about call states to predicates, we will restrict the discussion to goal-dependent analyses.

In order to improve the accuracy of goal-dependent analyses, some kind of description of the *initial* calls to the program should be given.[16] With this aim, we will use `entry` declarations in

[16] Predicate calls which are not initial will be called *internal*.

Property	Definition	Sufficient condition
L is abstractly executable to *true* in P	$RT(L,P) \subseteq TS(L,P)$	$\exists \lambda' \in A_{TS}(\overline{B}, D_\alpha) : \lambda_L \sqsubseteq \lambda'$
L is abstractly executable to *false* in P	$RT(L,P) \subseteq FF(L,P)$	$\exists \lambda' \in A_{FF}(\overline{B}, D_\alpha) : \lambda_L \sqsubseteq \lambda'$

Table 2: Abstract Executability

the spirit of [17]. Their role is to restrict the starting points of analysis to only those calls which satisfy a declaration of the form '`:- entry` *Pred* `:` *Call.* ' where *Call* is an abstract call substitution for *Pred*. For example, the following declaration informs the analyzer that at run-time all initial calls to the predicate `qsort/2` will have a term without variables in the first argument position:

```
:- entry qsort(A,B) : ground(A).
```

Though our framework allows having several entry declarations (for the same or different exported predicates), for the sake of clarity of the presentation we restrict ourselves to having one entry declaration only. Also, `CiaoPP` [87] supports a more general language, which includes properties defined in the source language [179]. In this setting, goal dependent abstract interpretation takes as input (1) a program P (2) an atom p, (3) an abstract substitution λ in (4) an abstract domain D_α which describes restrictions on the initial values, and computes a set of triples $Analysis(P, p, \lambda, D_\alpha) = \{\langle p_1, \lambda_1^c, \lambda_1^s\rangle, \ldots, \langle p_n, \lambda_n^c, \lambda_n^s\rangle\}$. In each triple $\langle p_i, \lambda_i^c, \lambda_i^s\rangle$, p_i is an atom and λ_i^c and λ_i^s are, respectively, the abstract call and success substitutions.[17] Due to space limitations, and given that it is now well understood, we do not describe here how we compute $Analysis(P, p, \lambda, D_\alpha)$. More details can be found in [89, 187] and their references. Given $Analysis(P, p, \lambda, D_\alpha) = \{\langle p_1, \lambda_1^c, \lambda_1^s\rangle, \ldots, \langle p_n, \lambda_n^c, \lambda_n^s\rangle\}$, correctness of abstract interpretation guarantees that the following propositions hold:

Proposition 14.2 (Correctness w.r.t. successes) *The abstract success substitutions cover all the concrete success substitutions which appear during execution, i.e.,* $\forall i = 1..n\ \forall \theta_c \in \gamma(\lambda_i^c)$ *if* $p_i\theta_c$ *succeeds in* P *with computed answer substitution* θ_s *then* $\theta_s \in \gamma(\lambda_i^s)$.

Proposition 14.3 (Correctness w.r.t. calls) *The abstract call substitutions cover all the concrete calls which appear during executions described by* $\langle p, \lambda\rangle$. *I.e., for any concrete call* c

[17] Actually, the analyzers used in practice generate information not only at the *predicate level*, as stated here for simplicity, but also at the *clause literal* level.

originated from an initial goal $p\theta$ *s.t.* $\theta \in \gamma(\lambda)$ *:* $\exists \langle p_j, \lambda_j^c, \lambda_j^s \rangle \in Analysis(P, p, \lambda, D_\alpha)$ *s.t.* $c = p_j\theta'$ *and* $\theta' \in \gamma(\lambda_j^c)$.

Proposition 14.3 is related to the closedness condition [148] required in partial deduction. A tuple $\langle p_j, \lambda_j^c, \bot \rangle$ indicates that all calls to predicate p_j with substitution $\theta \in \gamma(\lambda_j^c)$ either fail or loop, i.e., they do not produce any success substitutions. An analysis is said to be *multivariant* if more than one triple $\langle p, \lambda_1^c, \lambda_1^s \rangle$, ..., $\langle p, \lambda_n^c, \lambda_n^s \rangle$ $n > 1$ with $\lambda_i^c \neq \lambda_j^c$ for some i, j may be computed for the same predicate p.

15 Abstract Executability

The concept of *abstract executability* [77, 189] allows reducing at compile-time certain program fragments to the values *true, false,* or *error,* or to a simpler program fragment, by application of the information obtained via abstract interpretation. This allows optimizing and transforming the program (and also detecting errors at compile-time in the case of *error*).

For simplicity, we will limit herein the discussion to reducing a procedure call or program fragment L (for example, a "literal" in the case of logic programming) to either *true* or *false*. Each run-time invocation of the procedure call L will have a *local environment* which stores the particular values of each variable in L for that invocation. We will use θ to denote this environment (composed of assignments of values to variables, i.e., substitutions) and the restriction (projection) of the environment θ to the variables of a procedure call L is denoted $\theta|_L$.

We now introduce some definitions. Given a procedure call L to a predicate which performs no side-effects in a program P we define the *trivial success set* of L in P as:

$$TS(L, P) = \{\theta|_L : L\theta \text{ succeeds exactly once in } P \text{ with empty answer substitution } (\epsilon)\}$$

Similarly, given a procedure call L from a program P we define the *finite failure set* of L in P as:

$$FF(L, P) = \{\theta|_L : L\theta \text{ fails finitely in } P\}$$

Finally, given a procedure call L from a program P we define the *run-time substitution set* of L in P, denoted $RT(L, P)$, as the set of all possible substitutions (run-time environments) in the execution state just prior to executing L in any possible execution of program P.

Table 2 shows the conditions under which a procedure call L is abstractly executable to either *true* or *false*. In spite of the simplicity of the concepts, these definitions are in general not directly applicable in practice since $RT(L, P)$, $TS(L, P)$, and $FF(L, P)$ are generally not known at compile time. However, a *collecting semantics* is generally used as concrete semantics for abstract interpretation so that analysis computes for each procedure call L in the program an

abstract substitution λ_L which is a safe approximation of $RT(L,P)$, i.e. $\forall L \in P\ .\ RT(L,P) \subseteq \gamma(\lambda_L)$.

Also, under certain conditions we can compute either automatically or by hand sets of abstract values $A_{TS}(\overline{L}, D_\alpha)$ and $A_{FF}(\overline{L}, D_\alpha)$ where $\overline{L}$ stands for the *base form* of L, i.e., all the arguments of L contain distinct free variables. Intuitively, they contain abstract values in domain D_α which guarantee that the execution of $\overline{L}$ trivially succeeds (resp. finitely fails). Soundness requires that $\forall \lambda \in A_{TS}(\overline{L}, D_\alpha)\ \gamma(\lambda) \subseteq TS(\overline{L}, P)$ and $\forall \lambda \in A_{FF}(\overline{L}, D_\alpha)\ \gamma(\lambda) \subseteq FF(\overline{L}, P)$.

Even though the simple optimisations illustrated above may seem of narrow applicability, in fact for many builtin procedures such as those that check basic types or which inspect the structure of data, even these simple optimisations are indeed very relevant. Two non-trivial examples are their application to simplifying independence tests in program parallelization [189], discussed in Section 17, and the optimisation of delay conditions in logic programs with dynamic procedure call scheduling order [181], discussed in Section 18.

Also, the class of optimisations which can be performed can be made to cover traditional lower-level optimisations as well, provided the lower-level code to be optimised is "reflected" (i.e., is made explicit) at the source level or if the abstract interpretation is performed directly at the object level.

16 Abstract Multiple Specialization

The traditional approach used in analysis-based optimizing compilers is to first analyze the program and then use the information in $Analysis(P, p, \lambda, D_\alpha)$ to annotate the program with information which is then used for optimisation. Often, the underlying analysis algorithm is multi-variant. However, analysis information for the different versions of a procedure call is "flattened", i.e., "lubbed" together before being used for optimisation. Though this approach allows important optimisations, it produces optimisations which may be suboptimal when compared with the optimisations which could be achieved if separate specializations were implemented for the different versions considered by multi-variant analysis. More precisely, suppose $\{\langle p_j, \lambda_1^c, \lambda_1^s\rangle, \ldots, \langle p_j, \lambda_n^c, \lambda_n^s\rangle\}\ n > 1$ are the tuples in $Analysis(P, p, \lambda, D_\alpha)$ for predicate p_j. Generally, only one version for p_j is implemented, which is equivalent to specializing p_j w.r.t. $\lambda_1^c \sqcup \lambda_2^c, \ldots \sqcup \lambda_n^c$.

The main idea that we will exploit is to generate a different version of p_j for each tuple $\langle p_j, \lambda_i^c, \lambda_i^s\rangle$. Then, each version can be specialized w.r.t. λ_i^c regardless of the rest of the call substitutions $\lambda_j^c\ \forall j \neq i$. Hopefully, this will lead to further opportunities for optimisation in each particular version. Note that if analysis terminates the number of tuples in $Analysis(P, p, \lambda, D_\alpha)$

for each predicate must be finite, and thus the resulting program will be finite. We will refer to this kind of specialization as *abstract multiple specialization* [185, 189]. An important observation here is that abstract multiple specialization is not a *program with annotations* approach but rather a *semantic* approach in the terminology of Section 13.1.

16.1 Analysis And–Or Graphs

Traditional, goal dependent abstract interpreters for logic programs based on Bruynooghe's analysis framework [13] construct, in order to compute $Analysis(P, p, \lambda, D_\alpha)$, an and–or graph which corresponds to (or approximates) the abstract semantics of the program. We will denote by $AO(P, p, \lambda, D_\alpha)$ the and–or graph computed by the analyzer for a program P with calling pattern $\langle p, \lambda \rangle$ using the domain D_α. Such a graph can be viewed as a finite representation of the (possibly infinite) set of and–or trees explored by the (possibly infinite) concrete execution. Concrete and–or trees which are infinite can be represented finitely through a widening into a rational tree. Also, the use of abstract values instead of concrete ones allows representing infinitely many concrete execution trees with a single abstract analysis graph. Finiteness of $AO(P, p, \lambda, D_\alpha)$ (and thus termination of analysis) is achieved by considering an abstract domain D_α with certain characteristics (such as being finite, or of finite height, or without infinite ascending chains) or by the use of a *widening* operator [35].

The graph has two type of nodes: those which correspond to atoms (called *or–nodes*) and those which correspond to clauses (called *and–nodes*). Or–nodes are triples $\langle p_i, \lambda_i^c, \lambda_i^s \rangle$. As before, λ_i^c and λ_i^s are, respectively, a pair of abstract call and success substitutions for the atom p_i. For clarity, in the figures the atom p_i is superscripted with λ^c to the left and λ^s to the right of p_i respectively. For example, the or–node $\langle p(A), \{\}, \{A/a\} \rangle$ is depicted in the figure as ${}^{\{\}}p(A)^{\{A/a\}}$. And–nodes are pairs $\langle Id, H \rangle$ where $Id : inv02 - puebla.tex, v1.92004/01/1911 : 50 : 17asap - sotExp$ is a unique identifier for the node and H is the head of the clause to which the node corresponds. In the figures, they are represented as triangles and H is depicted to the right of the triangles. Note that the substitutions (atoms) labeling and–nodes are concrete whereas the substitutions labeling or–nodes are abstract. Finally, squares are used to represent the empty (true) atom. Or–nodes have arcs to and–nodes which represent the clauses with which the atom (possibly) unifies. And–nodes have arcs to or–nodes which represent the atoms in the body of the clause. Note that several instances of the same clause may exist in the analysis graph of a program. In order to avoid conflicts with variable names, clauses are standardized apart before adding to the analysis graph the nodes which correspond to such clause.

Intuitively, analysis algorithms are just graph traversal algorithms which, given $P, p, \lambda,$ and D_α, build $AO(P, p, \lambda, D_\alpha)$ by processing program clauses from left to right, adding the required

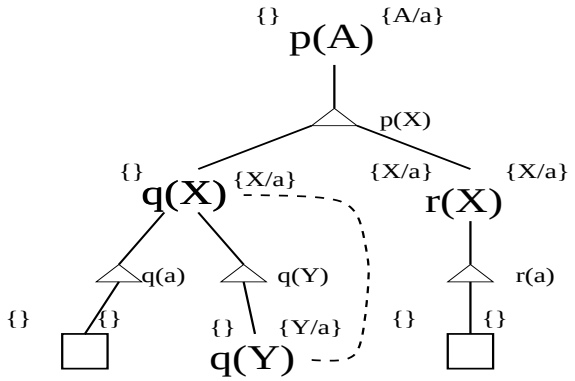


Figure 7: And–or analysis graph for a recursive program

nodes, and computing success substitutions until a global fixpoint is reached. For a given P, p, λ, and D_α there may be many different analysis graphs. However, there is a unique *least analysis graph* which gives the most precise information possible. This analysis graph corresponds to the least fixpoint of the abstract semantic equations. Each time the analysis algorithm creates a new or–node for some p_i and λ_i^c and before computing the corresponding λ_i^s, it checks whether $Analysis(P, p, \lambda, D_\alpha)$ already contains a tuple for (a variant of) p_i and λ_i^c. If that is the case, the or–node is not expanded and the already computed λ_i^s stored in $Analysis(P, p, \lambda, D_\alpha)$ is used for that or–node. This is done both for efficiency and for avoiding infinite loops when analyzing recursive predicates. As a result, several instances of the same or–node may appear in AO, but only one of them is expanded. We denote by $\mathsf{expansion}(N)$ the instance of the or–node N which is expanded. If there is no tuple for p_i and λ_i^c in $Analysis(P, p, \lambda, D_\alpha)$, the or–node is expanded, λ_i^s computed, and $\langle p_i, \lambda_i^c, \lambda_i^s \rangle$ added to $Analysis(P, p, \lambda, D_\alpha)$. Note that the success substitutions λ_i^s stored in $Analysis(P, p, \lambda, D_\alpha)$ are tentative and may be updated during analysis. Only when a global fixpoint is reached the success substitutions are safe approximations of the concrete success substitutions.

For clarity of the presentation, in the examples below we use the concrete domain as abstract domain. However, this cannot be done in general since analysis may not terminate. We will present other examples with more realistic domains later in the paper.

Example 16.2 *Consider the simple example program below taken from [120]. Figure 7 depicts a possible result of analysis for the initial call $p(A)$ with A unrestricted. The dotted arc indicates that the corresponding or–nodes have renamings of the same abstract call substitution.*

```
p(X):- q(X), r(X).
q(a).
q(X):- q(X).
r(a).
```

Algorithm 16.1 (Code Generation) *Given* $Analysis(P, p, \lambda, D_\alpha)$ *and* $AO(P, p, \lambda, D_\alpha)$ *generated by analysis for a program* P *an atom* p *with abstract substitution* $\lambda \in D_\alpha$ *do:*

- *For each tuple* $N = \langle a(\bar{t}), \lambda^c, \lambda^s \rangle \in Analysis(P, p, \lambda, D_\alpha)$ *generate a distinct predicate with name* $pred_N = \mathsf{name}(\langle a(\bar{t}), \lambda^c, \lambda^s \rangle)$.
- *Each predicate* $pred_N$ *is defined by the sequence of clauses*
  - $(pred_N(\bar{t}_1)$ `:-` $b'_1) :: \ldots :: (pred_N(\bar{t}_n)$ `:-` $b'_n)$
    where $\mathsf{expansion}(N, AO) = O_N$ *and*
    $\mathsf{children}(O_N, AO) = \langle Id_1, p_1(\bar{t}_1) \rangle :: \ldots :: \langle Id_i, p_i(\bar{t}_i) \rangle :: \ldots :: \langle Id_n, p_n(\bar{t}_n) \rangle$
- *Each body* b'_i *is defined as*
  - $b'_i = (pred_{i1}(\bar{t}_{i1}), \ldots, pred_{ik_i}(\bar{t}_{ik_i}))$
    where $pred_{ij} = \mathsf{name}(\langle a_{ij}(\bar{t}_{ij}), \lambda^c_{ij}, \lambda^s_{ij} \rangle)$, *and*
    $\mathsf{children}(\langle Id_i, p_i(\bar{t}_i) \rangle, AO) = \langle a_{i1}(\bar{t}_{i1}), \lambda^c_{i1}, \lambda^s_{i1} \rangle :: \ldots :: \langle a_{ik_i}(\bar{t}_{ik_i}), \lambda^c_{ik_i}, \lambda^s_{ik_i} \rangle$.

Figure 8: Algorithm for Code Generation

```
r(b).
```

Clearly, in the example program above the clause `r(b)` is useless and could be eliminated. Note that analysis has determined that in all successes of `q(X)`, and thus in calls to `r(X)`, the argument `X` will be bound to the value `a`. This is achieved by performing a fixpoint computation on the success values of `q(X)`. This is why in Figure 7 the or–node $\langle r(X), \{X/a\}, \{X/a\} \rangle$ only has one child (and–node).

16.2 Code Generation from an And–Or Graph

After introducing some notation, Algorithm 16.1 which generates a logic program from an analysis and–or graph is presented in Figure 8. Given a non-root node N, we denote by $parent(N, AO)$ the node $M \in AO$ such that there is an arc from M to N in AO, and $\mathsf{children}(N, AO)$ is the sequence of nodes $N_1 :: \ldots :: N_n$ $n \geq 0$ such that there is an arc from N to N' in AO iff $N' = N_i$ for some i and $\forall i, j = 0, \ldots, n$. N_i is to the left of N_j in AO iff $i < j$. Note that $\mathsf{children}(N, AO)$ may be applied both to or– and and–nodes. We assume the existence of an injective function name which (1) given $Analysis(P, p, \lambda, D_\alpha)$ returns a unique predicate name for each tuple and (2) $\mathsf{name}(\langle q(\bar{t}), \lambda^c, \lambda^s \rangle) = q$ iff $q(\bar{t}) = p$ (the exported predicate) and

$\lambda^c = \lambda$ (the restriction on initial calls), to ensure that top-level – exported – predicate names are preserved.

Let $AO(P, p, \lambda, D_\alpha)$ be an and–or graph. We denote by $P' = code\_gen(AO(P, p, \lambda, D_\alpha))$ that P' is the program obtained by applying Algorithm 16.1 to $AO(P, p, \lambda, D_\alpha)$.

Basically, the algorithm for code generation shown in Figure 8 creates a different version (predicate) name for each different (abstract) call substitution λ^c to each predicate p_i in the original program. This is easily done by associating a version to each or–node. Note that in principle such versions are identical except that atoms in clause bodies are renamed to always call the appropriate version. Correctness of this multiply specialized program is given by the correctness of the abstract interpretation procedure, as the extended program is obtained by simply materializing the (implicit) program with multiple versions from which the analysis has obtained its information.

Example 16.3 *The program generated by the code generation algorithm for the and–or graph in Figure 7 is shown below. The useless clause* `r(b)` *has been eliminated.*

```
p(X):- q(X), r(X).
q(a).
q(X):- q(X).
r(a).
```

The example above shows how the use of and–or graphs allows removing useless clauses. The example below shows how generating multiple specialized versions of a predicate can lead to optimisations which are not possible if only one version were implemented.

Example 16.4 *Consider the program P in Example 13.1. The and–or graph $AO(P, p, \top, D_\alpha)$ where D_α is a domain which captures mode information will have two or-nodes for predicate* `plus/3` *with different abstract call substitutions (we abbreviate $ground$ by g):*

$$\langle plus(X', Y', Z'), \{Z'/var\}, \{X'/g, Y'/g, Z'/g\}\rangle$$

and

$$\langle plus(X'', Y'', Z''), \{X''/g, Y''/g\}, \{X''/g, Y''/g, Z''/g\}\rangle$$

Now each of these call patterns can be optimised separately by abstractly executing the groundness tests. The final specialized program obtained is shown below:

```
p(X,Y,Res) :- plus1(X,Y,Tmp), plus2(1,Tmp,Res).
plus1(X,Y,Z) :- ground(X), ground(Y), !, Z is X+Y.
plus2(X,Y,Z) :- Z is X+Y.
```

```
mmultiply([],_,[]).
mmultiply([V0|Rest],V1,[Result|Others]) :-
        (ground(V1),
         indep([[V0,Rest],[V0,Others],[Rest,Result],[Result,Others]])
        ->   multiply(V1,V0,Result) & mmultiply(Rest,V1,Others)
        ;    multiply(V1,V0,Result), mmultiply(Rest,V1,Others)).

multiply([],_,[]).
multiply([V0|Rest],V1,[Result|Others]) :-
        (ground(V1),
         indep([[V0,Rest],[V0,Others],[Rest,Result],[Result,Others]])
        ->   vmul(V0,V1,Result) & multiply(Rest,V1,Others)
        ;    vmul(V0,V1,Result), multiply(Rest,V1,Others)).
```

Figure 9: Parallel mmatrix

Note that this program could be further improved by unfolding the call `plus2(1,Tmp,Res)`. This will be further discussed in Section 19. Also, two versions have been generated for predicate `plus/3`, namely `plus1/3` and `plus2/3`. In order to avoid code explosion our system performs a minimizing step a posteriori on the and–or graph in order to produce the minimal number of versions while maintaining all optimisations [189].

17 Program Parallelization

The final aim of parallelism is to achieve the maximum speed (effectiveness) while computing the same solution (correctness) as the sequential execution. The two main types of parallelism which can be exploited in logic programs are well known: or-parallelism and and–parallelism. In this work we concentrate on the case of and–parallelism. And-parallelism refers to the parallel execution of the literals in the body of a clause. See, for example, [82] and its references. If only *independent goals* are executed in parallel, both correctness and efficiency can be ensured [90].

17.1 The Annotation Process and Run-time Tests

The annotation (parallelization) process can be viewed as a source–to–source transformation from standard Prolog to a parallel dialect. Herein, we will use the & operator [86]. Execution of literals separated by & is performed in parallel if sufficient processors are available. Otherwise they will be executed sequentially.

The automatic parallelization process is performed as follows [164]: firstly, if requested by

the user, the Prolog program is analyzed using one or more global analyzers. Secondly, since side–effects cannot be allowed to execute freely in parallel, the original program is analyzed using the global analyzer described in [165] which propagates the side–effect characteristics of builtins determining the scope of side–effects. Finally, the *annotators* perform a source–to–source transformation of the program in which each clause is annotated with parallel expressions and conditions which encode the notion of independence used. In doing this they use the information provided by the global analyzers mentioned before.

17.2 An Example: Matrix Multiplication

A Prolog program for matrix multiplication is shown below. The declaration:

```
:-module(mmatrix,[mmultiply/3]).
```

is used by the (goal dependent) analyzer to determine that only calls to `mmultiply/3` may appear in top-level queries. In this case no information is given about the arguments in calls to the predicate `mmultiply/3` (however, this could be done using one or more `entry` declarations [17]).

```
:-module(mmatrix,[mmultiply/3]).

mmultiply([],_,[]).
mmultiply([V0|Rest], V1, [Result|Others]):-
    multiply(V1,V0,Result), mmultiply(Rest, V1, Others).
multiply([],_,[]).
multiply([V0|Rest], V1, [Result|Others]):-
    vmul(V0,V1,Result), multiply(Rest, V1, Others).
vmul([],[],0).
vmul([H1|T1], [H2|T2], Result):-
    Product is H1*H2, vmul(T1,T2, Newresult),
    Result is Product+Newresult.
```

If, for example, we want to specialize the program for the case in which the first two arguments of `mmultiply/3` are ground values and we inform the analyzer about this, the program would be parallelized without the need for any run-time tests. In our case the analyzer must in principle assume no knowledge regarding the instantiation state of the arguments at the module entry points.

Figure 9 contains the result of automatic parallelization under these assumptions. Conditions are written as `(cond -> then ; else)`, i.e., using standard Prolog syntax. The predicate `vmul/3` is not shown in Figure 9 because automatic parallelization has not detected any profitable parallelism in it (due to granularity control) and its code remains the same as in the original program.

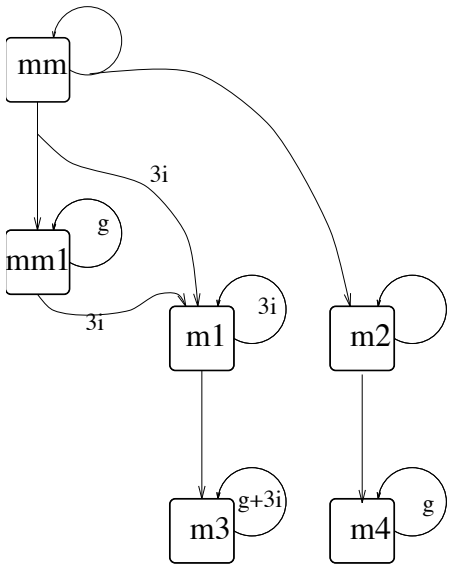


Figure 10: Call Graph of Specialized mmatrix

It is clear from Figure 9 that a good number of run-time tests has been introduced during the parallelization process. If the tests succeed the parallel code is executed. Otherwise the original sequential code is executed. The boolean test `indep(X,Y)` succeeds if and only if `X` and `Y` have no variables in common. For conciseness and efficiency, a series of tests `indep(X1,X2), ..., indep(Xn-1,Xn)` is written as `indep([[X1,X2], ..., [Xn-1,Xn]])`.

Clearly, these tests may cause considerable overhead in run-time performance, to the point of not even knowing at first sight if the parallelized program will offer speedup, i.e., if it will run faster than the sequential one. We will use abstract multiple specialization in order to reduce the run-time overhead and increase the speedup of parallel execution.

It is important to mention that abstract multiple specialization is able to automatically detect and extract some invariants in recursive loops: once a certain run-time test has succeeded it does not need to be checked in the following recursive calls [77]. Figure 10 shows the call graph of the specialized parallel program. The program itself is not shown for space limitations but can be found in [189]. In the figure, `mm` stands for `mmultiply/3` and `m` for `multiply/3`. In the and–or graph computed by analyis there are two or–nodes for predicate `mmultiply/3`, four for `multiply/3`, and eight for `vmul/3`. The minimization algorithm collapses all or–nodes for `vmul/3` into one since the different call patterns do not lead to interesting optimisations. However, two versions are generated for `mm`: `mm` and `mm1` and four for `m`. In Figure 10 edges are labeled with the number of tests which are avoided in each call to the corresponding version with respect to the non specialized program. For example, `g+3i` means that each execution of this

specialized version avoids a groundness and three independence tests. It can be seen in the figure that once the groundness test in any of `mm`, `m1`, or `m2` succeeds, it is detected as an invariant, and the more optimised versions `mm1`, `m3`, and `m4` respectively will be used in all remaining iterations.

18 Optimisation of Dynamic Scheduling

Most "second-generation" logic programming languages provide a flexible scheduling in which computation generally proceeds left-to-right, but some calls are dynamically "delayed" until their arguments are sufficiently instantiated. This general form of scheduling, often referred to as *dynamic scheduling*, which can be seen as a (restricted) class of concurrency, increases the expressive power of (constraint) logic programs. Unfortunately, it also has a significant time and space overhead.

In this section we present by means of examples two different classes of transformations. The first class simplifies the delay conditions associated with a particular literal. The second class of transformations reorders a delayed literal and moves it closer to the point where it wakes up. Both classes of transformations essentially preserve the search space and hence the operational behavior of the original program. However, reordering may change the execution order of delayed literals that are woken at exactly the same time. Note that this order is system dependent and it is rare for programmers to rely on a particular ordering.

Using the `CiaoPP` system we have built a tool which automatically optimises logic programs with delay using the above transformations. Initial experiments suggest that simplification of delay conditions is widely applicable and can significantly speed up execution, while reordering is less applicable but can also lead to substantial performance improvements.

18.1 Programs with Delaying Conditions

In dynamically scheduled languages the execution of some literal can be delayed until a particular delay condition holds. A *delay condition*, $Cond$, takes the current run-time environment and returns $true$ or $false$ indicating if evaluation can proceed or should be delayed. Typical primitive delay conditions are `ground(X)` and `nonvar(X)`. The latter holds iff `X` is bound to a non-variable term. Delay conditions can be combined to allow more complex delay behaviour. They can be conjoined, written ($Cond_1, Cond_2$), or disjoined, written ($Cond_1; Cond_2$).

A *delaying literal* is of the form $when(Cond, L)$, where $Cond$ is a delay condition and L is a literal. Evaluation of L will be delayed until $Cond$ holds for the current constraint store. Delay information can be *predicate-based* and *literal-based.* In the former, the delaying literal appears

as a declaration before the definition of the predicate, each instance of the predicate inheriting the delay condition. In the latter, the delaying literal appears in the body of some clause only affecting the literal L. It is straightforward to use predicate-based declarations to imitate literal-based delay, and vice versa. For simplicity, we will restrict ourselves to literal-based delay.

In logic programs with dynamic scheduling, a *literal* is either an atom or a delaying literal. We are assuming that all rule heads are normalized, since this simplifies the examples and corresponds to what is done in the analyzer.[18] This is not restrictive since programs can always be normalized. However, so as to preserve the behaviour of the original program under dynamic scheduling, the normalization process must ensure that head unifications are performed simultaneously, that is, grouped together in one primitive constraint.

18.2 Simplifying Dynamic Scheduling

Delay conditions may be evaluated each time a variable is touched. Simplifying such conditions can then lead to significant performance improvement. Essentially the behaviour of a delay condition is only relevant during the lifetime of the delaying literal. Hence, we can replace one delay condition by another (more efficient) condition if they are equivalent for all constraint stores that occur during the lifetime of the delaying literal.

Example 18.1 *Dynamic scheduling can be used in order to obtain much more general code. Consider for example the following program for naive reverse:*

```
:- module(nrev,[nrev/2]).
nrev([],[]).
nrev([X|Xs], Rs) :- nrev(Xs, R), app(R, [X], Rs).

app([],L,L).
app([X|Xs], Ys, [X|Zs]) :- app(Xs, Ys, Zs).
```

The `nrev/2` predicate can be used to reverse a list. For example, `nrev([1,2,3],Y)` will return `Y=[3,2,1]`. Since this program does not contain any impurities, we may in principle use it backwards, i.e., a call such as `nrev(X,[1,2,3])` should return `Y=[3,2,1]`. In fact, any Prolog system would compute that. However, if we ask for a second solution, the execution loops! One possible solution to avoid this behaviour is to reorder the two literals in the recursive clause of `nrev/2`, i.e., `nrev([X|Xs], Rs) :- app(R, [X], Rs), nrev(Xs, R).` However, now this program cannot be used forwards. This problem can be solved by means of dynamic scheduling which allows having a definition of `nrev/2` which works in both directions. Such a program is shown below:

```
nrev([],[]).
```

[18] `CiaoPP` does not need to normalize programs in order to analyze them, except for programs with dynamic scheduling.

```
nrev([X|Xs], Rs) :-
    when((nonvar(Xs);ground(R)),nrev(Xs, R)),
    when((nonvar(R);nonvar(Rs)),app(R, [X], Rs)).
app([],L,L).
app([X|Xs], Ys, [X|Zs]) :-
    when((nonvar(Xs);nonvar(Zs)),app(Xs, Ys, Zs)).
```

This has the disadvantage that dynamic scheduling may introduce important run-time overhead. However, we can use abstract specialization in order to optimise the above code for the required usage. In fact, our prototype specializer for dynamic scheduling [181] is able to optimise the program back to the original code without delays shown in Example 18.1 if it can infer that at the call the first argument is definitely ground. Also, it will reorder the two literals in the recursive clause of append if analysis guarantees that calls have a free variable in the first argument and the second argument is ground.

18.3 Reordering Delaying Literals

In spite of the apparent simplicity of the specialization of dynamic scheduling, it is indeed rather involved. First, the analysis has to be able to handle logic programs with dynamic scheduling. Doing so accurately is a complex task. Second, the purpose of specialization is not that the final program can be executed without delays but rather that the operational semantics, i.e., the search space, of the program is maintained.

Example 18.2 *In order to illustrate this we show the following example in which a naive algorithm for sorting lists is presented. It is based on the specification of the sorting algorithm: the resulting list must be a permutation of the input list and be sorted.*

```
naive_sort(List, Sorted) :-
   when(nonvar(Sorted),sorted_list(Sorted)),
   permute(List, Sorted).

sorted_list([]).
sorted_list([Fst|Oths]) :-
   when(nonvar(Oths),sorted_list1(Fst, Oths)).

sorted_list1(_, []).
sorted_list1(Fst, [Snd|Rest]) :-
   when((ground(Fst),ground(Snd)),Fst =< Snd),
   when(nonvar(Rest),sorted_list1(Snd, Rest)).

permute([],[]).
permute(List,Result):-
   when((nonvar(List);nonvar(Oths)),
```

```
                    delete(Elem, List, Oths)),
   Result = [Elem|Perm1],
   permute(Oths, Perm1).

delete(Elem, [Elem|Oths], Oths).
delete(Elem, List, Oths):-
   head(List,Oths) = head([Fst|TM],[Fst|R]),
   when((nonvar(TM);nonvar(R)),delete(Elem, TM, R)).
```

Thanks to the use of dynamic scheduling the code above has the following desirable features: (1) it can be used in order to sort a list; (2) if the second argument is ground, it can be used in order to generate all the possible lists (permutations) of a given sorted list; (3) though it is not a fast sorting algorithm, it behaves relatively well for small lists due to co-routining: generation of the permutation is interleaved with tests of its sortedness as new items are added to the partial solution, i.e., it is a *test while generate* algorithm rather than a *generate and test* one.

Of course, another alternative would have been to write by hand a program which checks sortedness of partial solutions explicitly. This has the disadvantage that it separates the code apart from its specification and that the obvious resulting code is once again not reversible.

Example 18.3 *In a call such as* `naive_sort([1,2,3],L)`, *the literal:*

`when(nonvar(Sorted),sorted_list(Sorted))`

will delay at the execution of predicate `naive_sort/2` *whereas it will definitely not delay after the execution of the literal* `permute(List, Sorted)`. *We may thus be tempted to reorder it across the following literal in the clause, obtaining:*

```
naive_sort(List, Sorted) :-
     permute(List, Sorted), sorted_list(Sorted).
```

which no longer needs dynamic scheduling. However, this resulting program would definitely be much less efficient than the original one since this changes the co-routining behaviour and thus the search space, and we end up in the generate and test algorithm. □

Though our specializer reordered the literals in the naive reverse example, it does not in this one. This is because the specializer only reorders a delaying literal L_i until after literal L_{i+1} if either (1) L_i is guaranteed not to wake up during the execution of L_{i+i} or (2) if it does, it can only wake up in program points of L_{i+1} which are *final*. More details can be found in [181]. The program obtained by our specializer when the first argument is ground is shown below:

```
naive_sort(List,Sorted) :-
    when(nonvar(Sorted),sorted_list(Sorted)),
    permute(List,Sorted).

sorted_list([]).
sorted_list([Fst|Oths]) :-
```

```
    when(nonvar(Oths),sorted_list1(Fst, Oths)).

sorted_list1(_, []).
sorted_list1(Fst, [Snd|Rest]) :-
    when((ground(Fst),ground(Snd)),Fst =< Snd),
    when(nonvar(Rest),sorted_list1(Snd, Rest)).

permute([],[]).
permute(List,Result) :-
    delete(Elem,List,Oths),
    Result=[Elem|Perm1],
    permute(Oths,Perm1).

delete(Elem, [Elem|Oths], Oths).
delete(Elem, List, Oths):-
    head(List,Oths) = head([Fst|TM],[Fst|R]),
    delete(Elem,TM,R).
```

18.4 Automating the Optimisation

In order to perform the optimisations discussed, the abstract interpretation framework used has to handle dynamic scheduling. Different analysis frameworks have been proposed for this. In our prototype we use the approach of [40]. For reordering, the analyzer needs to provide, in addition to a description of calling contexts, a description of the set of waking up literals at each program point.

The experimental results in [181] demonstrate that both simplification and reordering can lead to an order of magnitude performance improvement, and that they give reasonable speedups in most benchmarks. This is important because dynamic scheduling looks set to become increasingly prevalent in (constraint) logic programming languages because of its importance in implementing constraint solvers and controlling search as well as for implementing concurrency. In all these contexts, delay declarations are automatically introduced by the compiler. This has the advantage that it avoids the tedious and error prone task of having to do it by hand. Also, they are a clear target for abstract specialization.

19 Integration with Partial Evaluation

Most of the practical algorithms for program specialization use, to a greater or lesser degree, information generated by static program analysis. As already mentioned, one of the most widely used techniques for static analysis is abstract interpretation [35]. In fact, some of the relations

between abstract interpretation and partial evaluation have been identified before [62, 77, 56, 34, 185, 143, 103, 183, 124, 192, 37].

However, the role of analysis is so fundamental that it is natural to consider whether partial evaluation could be achieved directly by a generic, top-down abstract interpretation system.

19.1 And–Or Graphs Vs. SLD Trees

Almost all existing approaches to the (on-line) partial evaluation of logic programs use the same operational semantics, i.e *SLD resolution*, for both program execution and partial evaluation. Different alternative derivations of SLD resolution which may occur during execution constitute different branches in the *SLD tree*. See for example [147]. In partial deduction a slight modification to this semantics is required in order to allow incomplete derivations and thus incomplete SLD trees.

However, it is known [143] that the propagation of success information during partial evaluation is not optimal compared to that potentially achievable by abstract interpretation. The higher accuracy of abstract interpretation has already been hinted in Example 16.2.

We now show a further example of the power of abstract interpretation. This time, rather than the concrete domain we will use the abstract domain *eterms* [213] currently implemented in the `CiaoPP` system, and which is based on the concept of regular types [65]. Note that in this example the concrete domain cannot be used straight away, since the set of values which need to be represented is infinite.

Example 19.1 *Consider the following program and the initial call* `r(X)`

```
r(X) :- q(X),p(X).
q(a).
q(f(X)) :- q(X).
p(a).
p(f(X)) :- p(X).
p(g(X)) :- p(X).
```

It can be observed that the third clause for `p` *can be eliminated in the specialized program, since the call substitution for* `p(X)` *(i.e., the success substitution for* `q(X)`*) is of the form* `X=a` *or* `X=f(a)` *or* `X=f(...f(a)...)`*. Thus, the clause* `p(g(X)) :- p(X).` *is useless. Our implementation of the abstract domain* eterms *is able to determine that the value of* `X` *in any call to* `p(X)` *is described by the regular type* `rt` *whose definition as a regular unary Prolog program follows:*

```
rt(a).
rt(f(A)) :- rt(A).
```

Our specializer is in fact able to use this information in order to remove the useless clause mentioned above. Note that standard partial evaluation algorithms based on unfolding will not be

able to eliminate the third clause for `p`, since an atom of the form `p(X)` will be produced, no matter what local and global control is used.[19]

In addition to allowing the elimination of useless clauses, our specialization system is able to perform more aggressive optimisations, as shown in the example below.

Example 19.2 *Consider the following definition of the* `flatten_and_sort/2` *predicate.*

```
flatten_and_sort(Struct,Sorted_List):-
        sorted_int_list(Struct),
        Sorted_List=Struct.
flatten_and_sort(Struct,Sorted_List):-
        int_list(Struct),
        sort(Struct,Sorted_List).
flatten_and_sort(Struct,Sorted_List):-
        list_of_int_lists(Struct),
        flatten_list(Struct,Unsorted_List),
        sort(Unsorted_List,Sorted_List).
flatten_and_sort(Struct,Sorted_List):-
        tree(Struct),
        flatten_tree(Struct,Unsorted_List),
        sort(Unsorted_List,Sorted_List).
```

The argument `Struct` is a data structure which can be: a sorted list of integers, a list of integers, a list of lists of integers, or a tree which stores an integer in each non-leaf node. The predicate first determines which of the four possibilities mentioned above is the case and then, if needed, it uses the appropriate procedure for flattening before sorting the list of arguments, which is the output of the procedure. Clearly, if the input data structure is a list of integers there is no need for flattening the list. Furthermore, if it is already sorted, there is no need to sort it either. Though we could define a flatten predicate which is able to flatten both lists and binary trees, it is often the case that distinct predicates for flattening lists and for flattening trees already exist (in different libraries).

We show below the Prolog definition of the properties `sorted_int_list/1`, `int_list/1`, and `list_of_int_lists/1`. It can be observed that the last two predicates are indeed unary logic programs which correspond to deterministic regular types. This is indicated to `CiaoPP` with the declaration `regtype`.

```
sorted_int_list([]).
sorted_int_list([N]):-  int(N).
sorted_int_list([A,B|R]):- int(A), int(B),
        A =< B, sorted_int_list([B|R]).

:- regtype int_list/1.
```

[19] Conjunctive partial deduction [144] can solve this problem in a completely different way.

```
int_list([]).
int_list([H|L]):- int(H), int_list(L).

:- regtype list_of_int_lists/1.
list_of_int_lists([]).
list_of_int_lists([H|L]):-
        int_list(H), list_of_int_lists(L).

:- regtype tree/1.
tree(void).
tree(t(L,N,R)):- int(N), tree(L), tree(R).
```

The `regtype` declaration is checked by `CiaoPP` against the code defining the property. If the code does not correspond to a deterministic regular type, an error message is issued. If it is, this information can be used by the specializer in order to be able to abstractly execute to the value true the whole execution of the predicate. The sufficient conditions for this are (1) the predicate does not perform any side-effects, and (2) the calling abstract substitution must be equal or more particular than the success substitution for the predicate. Note that abstractly executing a predicate call to false using regular types does not need the `regtype` declaration. Any call to a predicate p can be abstractly executed to false if (1) execution of p is guaranteed not to perform any side-effects (2) the call substitution is incompatible with the success substitution of p or equivalently, the success substitution using goal-dependent analysis for p and λ_p^c is the empty substitution $\bot$. This is further discussed in Section 19.3. For example, if we call `sorted_int_list(Struct)` with `Struct` bound to a binary tree, the system can determine that this call is incompatible with the success type of `sorted_int_list`, which for the regular type analysis is approximated by `int_list`.

For example, the above program when specialized using the *eterms* domain for the call `main/0`, defined as:

```
main:-int_list(L),append(L,[3],L1),flatten_and_sort(L1,_).
```

optimises the definitions of `flatten_and_sort/2` and `int_list/2` as shown below.

```
   flatten_and_sort(Struct,Sorted_List) :-
     sorted_int_list(Struct), Sorted_List = Struct.
flatten_and_sort(Struct,Sorted_List) :-
    sort(Struct,Sorted_List).
sorted_int_list([]).
sorted_int_list([N]).
sorted_int_list([A,B|R]) :- A=<B, sorted_int_list([B|R]).
```

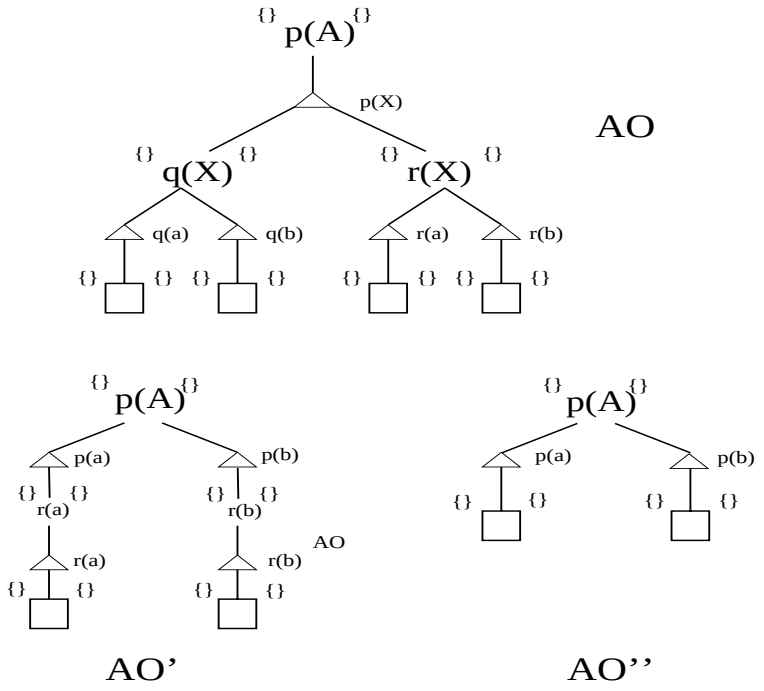


Figure 11: Example Node Unfoldings

Since analysis using *eterms* infers that the call to `flatten_and_sort/2` has got a non-empty list of integers as first argument, the specializer is able to abstractly execute the tests for `list_of_int_lists/1` and `tree/1` to false, since they are incompatible with their calling types. In addition, the `list/1` test in the second clause for `flatten_and_sort/2` has been abstractly executed to true, the same as the `integer/1` tests in `sorted_int_list/1`. This is an example in which abstract execution allows "executing" at compile-time a test whose execution would require traversing the data structure at run-time.

The examples above show that and–or graphs allow a level of success information propagation not possible in traditional partial evaluation. This observation already provides motivation for studying the integration of full partial evaluation in an analysis/specialization framework based on abstract interpretation.

19.2 Partial Evaluation using And–Or Graphs

We now discuss how the global and local control aspects of on-line partial evaluation appear in the setting of abstract interpretation algorithms.

19.2.1 Global Control in Abstract Interpretation

Effectiveness of traditional partial deduction greatly depends on the set of atoms $\mathbf{A} = \{A_1, \ldots, A_n\}$ for which (specialized) code is to be generated. This set is mainly determined by the global control used. However, in abstract specialization the role of the atoms in $\mathbf{A}$ is played by the set of or–nodes $Analysis(P, p, \lambda, D_\alpha)$. The choice of abstract domain and widening operators (if any) will determine the number of or–nodes (equivalently, $\mathbf{A}$). The finer-grained the abstract domain is, the larger the set $\mathbf{A}$ will be. In conclusion, the role of so-called global control in partial evaluation is played in abstract interpretation by our particular choice of abstract domain and widening operators (which are strictly required for ensuring termination when the abstract domain contains ascending chains which are infinite – as is the case for the concrete domain and for domains based on regular types).

Note that the specialization framework we propose is very general. Depending on the kind of optimisations we are interested in performing, different domains (and widening operators) should be used and thus different $\mathbf{A}$ sets would be obtained.

19.2.2 Local Control in Abstract Interpretation

Local control in partial evaluation determines how each atom in $\mathbf{A}$ should be unfolded. However, in traditional abstract interpretation frameworks each or–node is related by just one (abstract) unfolding step to its children. This corresponds to a trivial local control (unfolding rule) in partial evaluation.

Note that if we use abstract domains for analysis which allow propagating enough information about the success of an or–node, it is possible to perform useful specialization on other or-nodes. This requires that the lub operator not lose "much" information, for example by allowing sets of abstract substitutions. The advantage of this method is that no modification of the abstract interpretation framework is required. An example of this has been shown in Example 19.1. Such specialization is not possible by methods based on unfolding (unfolding is a standard program transformation technique in which an atom in the body of a clause, i.e., a call to a procedure, is conceptually replaced by the code of such procedure).

Another approach to overcoming this limitation of abstract interpretation is the use of *node-unfolding* [192]. Node-unfolding is a *graph* transformation technique which given an and–or graph AO and an or–node N in AO builds a new and–or graph AO'. Such graph transformation mimics the effect of traditional unfolding.

Example 19.3 *Consider the program below. The analysis graph generated without performing any node-unfolding is shown in Figure 11 as AO, using the concrete domain as abstract domain*

and the most specific generalization *(msg) as lub operator for summarizing different success substitutions into one. As discussed in Section 19.2.3 below, the msg is a rather crude lub operator. However, we use it for the sake of clarity of the example.*

```
p(X):- q(X), r(X).
q(a).
q(b).
r(a).
r(b).
```

AO' is an analysis graph for the same program but this time the or–node $\langle q(X), \{\}, \{\}\rangle$ has been unfolded. Finally, graph AO'' in the figure is the result of applying node-unfolding twice to AO', once w.r.t. $\langle p(a), \{\}, \{\}\rangle$ and another one w.r.t $\langle p(b), \{\}, \{\}\rangle$. The code generated by $code\_gen(AO'')$ is the program:

```
p(a).
p(b).
```

An important question is the moment at which node-unfolding is performed, i.e., during or after building AO. The simplest possibility is to perform node-unfolding of an or–node prior to computing its success substitution. This corresponds to what is done in partial deduction: local control is performed first and then atoms are passed to global control. It allows performing node-unfolding after computing the success-substitution of an or-node, even after computing the final and–or graph. This allows having more information prior to deciding whether to unfold a node or not. Thus, we consider it a more challenging approach. The main difficulty lies in being able to efficiently rebuild the analysis and–or graph so as to reach a fixpoint after the graph is modified by node-unfolding. We believe that this cost can be kept quite reasonable by the use of incremental analysis techniques such as those presented in [89, 187].

19.2.3 Abstract Domains and Widenings for Partial Evaluation

We now address the features which an abstract domain (and associated widening operators) should have in order to be appropriate for performing partial evaluation within the abstract specialization framework. They should (1) simulate the effect of unfolding, which is how bindings are propagated in partial evaluation. The abstract domain has to be capable of tracking such bindings. This suggests that domains based on term structure are required. In addition, the domain (2) needs to capture *disjunctive information*. This makes it possible to distinguish, in a single abstract substitution, several bindings resulting from different branches of computation. A term domain whose least upper bound is based on the *msg* (most specific generalization), for instance, will rapidly lose information about multiple answers since all substitutions are combined into one binding.

We now discuss two classes of domains which have the above mentioned features. One

class is based on sets of depth-k substitutions with set union as the least upper bound operator. However, uniform depth bounds are usually either too imprecise (if k is too small) or generate much redundancy if larger values of k are chosen. One way to eliminate the depth-bound k in the abstract domain is to depend on a suitable widening operator which will guarantee that the set of or–nodes remains finite. Many techniques have been developed for global control of partial evaluation. Such techniques make use of advanced data structures such as *characteristic trees* [61], [118] (related to *neighborhoods* [208]), *trace-terms* [68], and *global trees* [158], and combinations of them [139]. Thus, it seems possible to adapt these techniques to the case of abstract interpretation and formalize them as widening operators.

The second class of domains are those based on regular-types [65, 83, 213] and seem very good candidates, their main drawback being that inter-argument dependencies are lost. Independently of our work in `CiaoPP`, recently there has been a lot of interest in the application of regular types for improving partial evaluation [70, 130]. The use of non-deterministic regular types [71] presents an interesting trade-off since on one hand they allow improved accuracy but on the other they require a higher computational cost and their applicability to program specialization should be further explored.

19.3 Code Generation using Success Substitutions

One important feature of abstract specialization not available in partial evaluation is that for each or-node, in addition to a call substitution, there is also an abstract substitution which describes the success of the call. If the properties captured by the abstract domain are downwards closed (as is the case with variable bindings), it is natural to consider specialization w.r.t. success substitutions rather than call substitutions (only). We first recall some notation from [192].

Definition 19.4 (partial concretization) *A function* $part\_conc : D_\alpha \rightarrow D$ *is a* partial concretization *iff* $\forall \lambda \in D_\alpha \; \forall \theta' \in \gamma(\lambda) \; \exists \theta''$ *s.t.* $\theta' = part\_conc(\lambda)\theta''$.

$part\_conc(\lambda)$ can be regarded as containing (part of) the definite information about concrete bindings that the abstract substitution λ captures. Note that different partial concretizations of an abstract substitution λ with different accuracy may be considered. For example if the abstract domain is a depth-k abstraction and $\lambda = \{X/f(f(Y)) or X/f(a)\}$, a most accurate $part\_conc(\lambda)$ is $\{X/f(Z)\}$. Note also that $part\_conc(\lambda) = \epsilon$ where ϵ is the empty substitution, is a trivially correct partial concretization of any λ.

It is straightforward to modify Algorithm 16.1 in order to exploit answer substitutions as well. Such algorithm can be found in [192]. Specialization w.r.t. answers will in general provide further specialized (and optimised) programs as in general the success substitution (which

describes answers) computed by abstract interpretation is more informative (restricted) than the call substitution. However, this cannot be done for example if the program contains calls to extra-logical predicates such as `var/1`.

Specializing w.r.t answer substitutions enables optimisations which are not possible to achieve by finite unfolding. For example, abstract interpretation can detect both finite and infinite failure of a predicate p. In both cases, the abstract success substitution for p will be $\bot$. If p does not perform side effects, the definition of p generated by our specializer is `p(`$\bar{t}$`):- fail.`, as it is known to produce no answers. Even if the success substitution λ^s for $\langle p, \lambda^c, \lambda^s \rangle$ is not $\bot$, individual clauses for p whose success substitution is $\bot$ (useless clauses) for the considered λ^c are removed from the final program.

Note that the specialized program may fail finitely while the original one loops. We believe this kind of optimisations are desirable in most cases. However, optimisation w.r.t. answers is optional in our system.

20 Related Work

Abstract specialization is a framework which can be used successfully in different contexts. We have discussed its application to program parallelization and optimisation of dynamic scheduling. The framework is generic in that it can be instantiated with different abstract domains which provide different kinds of information according to the optimisations which we aim at performing. If the abstract domain captures term structure then it is possible to obtain information which can then be used to perform optimisations which are very related to those which take place during partial evaluation.

The integration of partial evaluation and abstract interpretation has been attempted before, both from the partial evaluation and the abstract interpretation perspectives. Some preliminary studies are [62, 56] in which an integration is attempted from the point of view of partial evaluation. Another integration in the context of functional programs is presented in [34]. On the other hand, the drawbacks of traditional partial evaluation techniques for propagating success information are identified in [143] and some of the possible advantages of a full integration of partial evaluation and abstract interpretation are presented in [103].

From an abstract interpretation perspective, the integration has also received considerable attention. The first complete framework for multiple specialization based on abstract interpretation is presented in [216]. The first implementation and experimental evaluation appears in [185]. However, these systems do not perform unfolding.

To the best of our knowledge, the first relatively satisfactory framework for the integration of

abstract interpretation and partial evaluation is [183, 192].

A completely different framework for the integration of partial deduction and abstract interpretation is presented in [124]. In this formulation a top-down specialization algorithm is presented which assumes the existence of an *abstract unfolding* and an *abstract resolution* operation and which generalizes existing algorithms for partial evaluation. Such framework provides interesting insights on the problems involved together with correctness conditions which can be used to prove that a given specialization framework, which possibly uses abstract interpretation, is correct. One important difference is that in our approach a single (and already existing) top-down abstract interpretation algorithm augmented with an unfolding rule performs propagation of both the call and success patterns in an integrated fashion, whereas in [124] the success propagation used is added in an ad hoc way and is not multivariant, and thus less precise.

Another difference between the two approaches is that [124] is capable of dealing with conjunctions and not only atoms.

The need for more general information than the concrete substitutions handled by partial evaluation has been identified repeatedly in previous work, such as [34, 173]. Though the aims of abstract specialization and those of [173] are quite similar, the means proposed to achieve them are completely different. Also, abstract interpretation is not used and it sticks to the more traditional SLD semantics.

More recently, [37] presents a very general view which integrates program transformation and abstract interpretation. This result allows formalizing partial evaluation as an abstract interpretation (as done by abstract specialization). This new formalization of program transformation may enable other novel program optimisation techniques.

21 Conclusions

Abstract specialization can be seen as a semantic approach much in the same way as existing frameworks for partial deduction [148, 110, 57, 120] and also as other attempts at the integration of partial evaluation and abstract interpretation of logic programs [124, 70, 130]. One of the main differences between abstract specialization and the aforementioned techniques is the underlying semantics. Abstract specialization is based on and–or trees whereas the rest are based on SLD trees. Though SLD-trees have the conceptual advantage that the semantics used for program specialization is almost identical to that used during program execution, our approach has other practical and conceptual advantages. For example, optimisations based on and–or trees can be done to preserve number and order of solutions, an issue often overlooked by traditional partial deduction systems. Furthermore, they allow performing optimisations not achievable by means

of unfolding, including the detection of infinite failure.

A pragmatic motivation for this work is the availability of off-the-shelf generic abstract interpretation engines such as the one in `CiaoPP` [87][20] which greatly facilitate the efficient implementation of analyses. Such analysis can deal with all features of real programs [17] in an accurate way, including builtins, libraries and modules [190]. But, more generally, we argue that *the existence of such an abstract interpreter in advanced optimizing compilers is likely*, and thus using the analyzer itself to perform partial evaluation can result in a great simplification of the architecture of the compiler.

[20] More information on `Ciao` and `CiaoPP` is available at `www.clip.dia.fi.upm.es`

Part III
More Precise Yet Efficient Type Inference for Logic Programs

Type analyses of logic programs which aim at inferring the types of the program being analyzed are presented in a unified abstract interpretation-based framework. This covers most classical abstract interpretation-based type analyzers for logic programs, built on either top-down or bottom-up interpretation of the program. In this setting, we discuss the widening operator, arguably a crucial one. We present a new widening which is more precise than those previously proposed. Practical results with our analysis domain are also presented, showing that it also allows for efficient analysis.

Furthermore, we introduce an abstract domain consisting of atomic formulas constrained by linear arithmetic constraints (or convex hulls). This domain is used in an algorithm for specialization of constraint logic programs. The algorithm incorporates in a single phase both top-down goal directed propagation and bottom-up answer propagation, and uses a widening on the convex hull domain to ensure termination. We give examples to show the precision gained by this approach over other methods in the literature for specializing constraint logic programs. The specialization method can also be used for ordinary logic programs containing arithmetic, as well as constraint logic programs. Assignments, inequalities and equalities with arithmetic expressions can be interpreted as constraints during specialization, thus increasing the amount of specialization that can be achieved.

22 Background

In type analyses, the widening operation has much influence in the results. If the widening is too aggressive in making approximations then the analysis results may be too imprecise. On the other hand, if it is not sufficiently aggressive then the analysis may become too inefficient.

Widening operators are aimed at identifying the recursive structure of the types being inferred. All widening operators already proposed in the literature are based on locating type nodes with the same functors, which are possible sources of recursion. However, they disregard whether such nodes come in fact from a recursive structure in the program or not. This may originate an unnecessary loss of precision, since the widening result may then impose a recursive structure on the resulting type in argument positions where the concrete program is in fact not recursive. We propose a widening operator to try to remedy this problem.

This part of the deliverable is organised as follows: We first revisit regular types (Section 23) and, in particular, deterministic ones. Then, we focus on deterministic types for ease of presentation; however, there is nothing in our widening which prevents it to be applicable also to non-deterministic types. The abstract interpretation framework is set up in Section 24. Section 25 reviews previous widenings in the literature, and Section 26 presents ours. In Section 27 experimental results based on our widening operator are presented. A constraint domain based on linear arithmetic equalities and inequalities is reviewed in Section 28.1. The structure of the specialization algorithm is presented (Section 29), along with examples illustrating its key aspects. In Section 30 more examples of specialization using the domain of linear arithmetic constraints are given. Comparisons with related work are provided in Section 31 while, finally, some remarks and pointers for future work are considered in Section 32.

23 Regular Types

A *regular type* [39] is a type representing a class of terms that can be described by a regular term grammar. A *regular term grammar,* or grammar for short, describes a set of finite terms constructed from a finite alphabet $\mathcal{F}$ of *ranked function symbols* or *functors*. A grammar $G = (S, \mathcal{T}, \mathcal{F}, \mathcal{R})$ consists of a set of non-terminal symbols $\mathcal{T}$, one distinguished symbol $S \in \mathcal{T}$, and a finite set $\mathcal{R}$ of productions $T \longrightarrow rhs$, where $T \in \mathcal{T}$ is a non-terminal and the right hand side rhs is either a non-terminal or a term $f(T_1, \ldots, T_n)$ constructed from an n-ary function symbol $f \in \mathcal{F}$ and n non-terminals.

The non-terminals $\mathcal{T}$ are *types* describing (ground) terms built from the functors in $\mathcal{F}$. The concretization $\gamma(T)$ of a non-terminal T is the set of terms derivable from its productions, that is,

$$\begin{aligned} \gamma(T) &= \bigcup_{(T \longrightarrow rhs) \in \mathcal{R}} \gamma(rhs) \\ \gamma(f(T_1, \ldots, T_n)) &= \{f(t_1, \ldots, t_n) \mid t_i \in \gamma(T_i)\} \end{aligned}$$

The types of interest are each defined by one grammar: each T_i is defined by a grammar $(T_i, \mathcal{T}_i, \mathcal{F}, \mathcal{R}_i)$, so that for any two types of interest T_1 and T_2 on $\mathcal{F}$, $\mathcal{T}_1 \cap \mathcal{T}_2 = \emptyset$. Sometimes, we will be interested in types defined by non-terminals of a grammar $(T, \mathcal{T}, \mathcal{F}, \mathcal{R})$ other than the distinguished non-terminal T. This is formalized by defining a type $T_i \in \mathcal{T}$ as the grammar

$$(T_i, \{T \in \mathcal{T} \mid T_i \xrightarrow{reach}{}^*_{\mathcal{R}} T\}, \mathcal{F}, \{(T \longrightarrow rhs) \in \mathcal{R} \mid T_i \xrightarrow{reach}{}^*_{\mathcal{R}} T\}) \tag{9}$$

where all the non-terminals are renamed apart, $\xrightarrow{reach}{}^*_{\mathcal{R}}$ is the reflexive and transitive closure of $\xrightarrow{reach}_{\mathcal{R}}$ and

$$T_i \xrightarrow{reach}_{\mathcal{R}} T_j \text{ iff } T_i \longrightarrow_{\mathcal{R}} T_j \text{ or } T_i \longrightarrow_{\mathcal{R}} f(\ldots, T_j, \ldots).$$

A grammar is in *normal form* if none of the right hand sides are non-terminals. A particular class of grammars are deterministic ones. A grammar is *deterministic* if it is in normal form and for each non-terminal T the function symbols are all distinct in the right hand sides of the productions for T.

Deterministic grammars are less expressive than non-deterministic ones. Deterministic grammars can only express sets of terms which are *tuple-distributive*; informally speaking, which are "closed under exchange of arguments". I.e., if the set contains two terms of the same functor, then it also contains terms with the same principal functor obtained by exchanging subterms of the previous two terms in the same argument positions. Basically, no dependencies between arguments of a term can be expressed with deterministic grammars.

Example 23.1 Consider the type T denoting the set $\{f(a,b), f(c,d)\}$, which is non-deterministic,

$$\begin{array}{llll} T \longrightarrow f(A,B) & A \longrightarrow a & C \longrightarrow c \\ T \longrightarrow f(C,D) & B \longrightarrow b & D \longrightarrow d \end{array}$$

A deterministic type T' with a concretization which included $\gamma(T)$ would also have to include $\{f(c,b), f(a,d)\}$, that is,

$$\begin{array}{llll} T' \longrightarrow f(AC,BD) & AC \longrightarrow a & BD \longrightarrow b \\ & AC \longrightarrow c & BD \longrightarrow d \end{array}$$

To facilitate the presentation non-terminals with a single production will often be "inlined" and multiple right hand sides combined so that T above will be written $T \longrightarrow f(a,b) \mid f(c,d)$ and T' as

$$T' \longrightarrow f(AC,BD) \qquad AC \longrightarrow a \mid c \qquad BD \longrightarrow b \mid d$$

To be able to describe terms containing numbers and variables we introduce two distinguished symbols **num** and **any**, plus an additional $\perp$. The concretization of **num** is the set of all numbers, the concretization of **any** is the set of all terms (including variables), and the concretization of $\perp$ is the empty set of terms. These symbols are non-terminals but they are considered terminals to the effect of regarding a grammar as deterministic.

Let $\mathcal{G}$ be the set of all grammars, if T_1, T_2 belong to $\mathcal{G}$, the relation $T_1 \equiv T_2 \Leftrightarrow \gamma(T_1) = \gamma(T_2)$ is an equivalence relation. The quotient set $\mathcal{G}/\equiv$ is a complete lattice with top element **any** and bottom element $\perp$ based on the relation of *containment*, or type *inclusion*: for every $\overline{T_1}, \overline{T_2} \in \mathcal{G}/\equiv$, $\overline{T_1} \sqsubseteq \overline{T_2} \Leftrightarrow \gamma(T_1) \subseteq \gamma(T_2)$. We will denote $\overline{T_i}$ simply by T_i.

The least upper bound is given by type *union*, $(T_1 \sqcup T_2)$, and the greatest lower bound by type *intersection*, $(T_1 \sqcap T_2)$ [39]. It can be shown that intersection describes term unification:

$$t_1^* \subseteq \gamma(T_1) \wedge t_2^* \subseteq \gamma(T_2) \wedge t_1\theta = t_2\theta \Rightarrow (t_1\theta)^* \subseteq \gamma(T_1 \sqcap T_2)$$

where t^* denotes the set of ground terms which are instances of the term t.

24 Abstract Domain for Type Inference

In an abstract interpretation-based type analysis, a type is used as an abstract description of a set of terms. Given variables of interest $\{x_1, \ldots, x_n\}$, any substitution $\theta = \{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$ can be approximated by an *abstract substitution* $\{x_1 \leftarrow T_{x_1}, \ldots, x_n \leftarrow T_{x_n}\}$ where $t_i \in \gamma(T_{x_i})$ and each type $T_{x_i} \in \mathcal{G}/\equiv$. We will write abstract substitutions as tuples $\langle T_1, \ldots, T_n \rangle$, and sometimes also abbreviate a tuple simply as T^n.

Concretization is lifted up to abstract substitutions straightforwardly,

$$\gamma(\langle T_1, \ldots, T_n \rangle) \;=\; \{\, \{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\} \mid t_i \in \gamma(T_i) \,\}$$

as well as the equivalence relation $\equiv$. Additionally, we consider a distinguished abstract substitution $\bot$ as a representative of any $\langle T_1, \ldots, T_n \rangle$ such that there is $T_i = \bot$. Of course, $\gamma(\bot) = \emptyset$.

An ordering on the domain is obtained as the natural element-wise extension of the ordering on types:

$$\begin{array}{rcl} \bot & \sqsubseteq & T^n \\ \langle T_1, \ldots, T_n \rangle & \not\sqsubseteq & \bot \\ \langle T_1, \ldots, T_n \rangle & \sqsubseteq & \langle T'_1, \ldots, T'_n \rangle \quad \Longleftrightarrow \quad \forall_{1 \le i \le n} T_i \sqsubseteq T'_i \end{array}$$

The domain is a lattice with bottom element $\bot$ and top element $\langle T_1, \ldots, T_n \rangle$ such that $T_1 = \ldots = T_n =$ **any**. The greatest lower bound and least upper bound domain operations are lifted also element-wise, as follows,

$$\begin{array}{rcl} \bot \sqcup T^n = T^n \sqcup \bot & = & T^n \\ \langle T_1, \ldots, T_n \rangle \sqcup \langle T'_1, \ldots, T'_n \rangle & = & \langle T_1 \sqcup T'_1, \ldots, T_n \sqcup T'_n \rangle \\ \bot \sqcap T^n = T^n \sqcap \bot & = & \bot \\ \langle T_1, \ldots, T_n \rangle \sqcap \langle T'_1, \ldots, T'_n \rangle & = & \langle T_1 \sqcap T'_1, \ldots, T_n \sqcap T'_n \rangle \end{array}$$

Using the adjoint α of γ as abstraction function, it can be shown that $(2^{\Theta}, \alpha, \Omega, \gamma)$ is a Galois insertion, where Θ is the domain of concrete substitutions and Ω that of abstract substitutions.

The following abstract unification operator can be shown to approximate the concrete one. Let $x = t$ be a concrete unification equation, with x a variable,t any term, and T^n the current abstract substitution, and let y_j, $j = 1, \ldots, m$ be the variables of t, the new abstract substitution is:

$$amgu(T^n, x = t) = T^n[T_x/T'_x, T_{y_1}/T'_{y_1}, \ldots, T_{y_m}/T'_{y_m}] \tag{10}$$

with each T replaced by T' in the tuple, $T'_x = T_x \sqcap t\mu$, $\mu = \{y_1 \leftarrow T_{y_1}, \ldots, y_m \leftarrow T_{y_m}\}$, and $solve(t, T'_x) = \{y_1 = T'_{y_1}, \ldots, y_m = T'_{y_m}\}$, a set of equations that define the types of the

variables of a term $t \in \gamma(T'_x)$, obtained as:

$$solve(t,T) = \begin{cases} \{t = T\} & \text{if} \quad t \text{ is a variable} \\ \bigcup\limits_{T \longrightarrow f(T_1,\ldots,T_n)} \bigcup\limits_{i=1,\ldots,n} solve(t_i, T_i) & \text{if} \quad t \text{ is } f(t_1,\ldots,t_n) \end{cases}$$

In this abstract interpretation-based setting, analysis with a monotonic semantic function can be easily shown correct. However, it is not guaranteed to terminate, since Ω has infinite ascending chains. To guarantee termination, a widening operator is required.

Example 24.1 Consider the following program which defines the regular type lists of lists of numbers:

```
list_of_lists([]).                  num_list([]).
list_of_lists([L|Ls]):-             num_list([N|Xs]):-
        num_list(L),                        number(N),
        list_of_lists(Ls).                  num_list(Xs).
```

For the argument of `num_list`, without a widening operator, an analysis would obtain the following first three approximations:

$$T_0 \longrightarrow [] \qquad T_1 \longrightarrow [] \mid .(\mathbf{num}, T_0) \qquad T_2 \longrightarrow [] \mid .(\mathbf{num}, T_1)$$

where each T_i represents a list of i numbers. Analysis will never terminate, since it would keep on obtaining a new type representing a list with one more number. A widening operator would be required that over-approximates some type T_l to something like

$$T_l \longrightarrow [] \mid .(\mathbf{num}, T_l)$$

which is the expected type, and allows termination of the analysis.

25 Widenings

The widening operation is required to guarantee that an analysis terminates when the abstract domain has infinite ascending chains as is the case of regular types.

Functor Widening This is probably the simplest widening operator which still keeps information from the recursive structure of the program that "produces" the corresponding terms. The idea behind it is to create a type and a production for each functor symbol in the original type. All arguments of the function symbols are replaced with the new types [159].

Example 25.1 Consider predicate `list_of_lists` of Example 24.24.1, its argument should ideally have the following type:

$$T_{ll} \longrightarrow [] \mid .(T_l, T_{ll}) \qquad T_l \longrightarrow [] \mid .(\mathbf{num}, T_l)$$

but the functor widening will yield

$$T \longrightarrow [] \mid \mathbf{num} \mid .(T, T)$$

Type Jungle Widening A type jungle is a grammar where each functor always has the same arguments. It was originally proposed as a finite type domain [146] , since in a domain where all grammars are of the type jungle class all ascending chains are finite. However, it can be used as a subdomain to provide a widening operator.

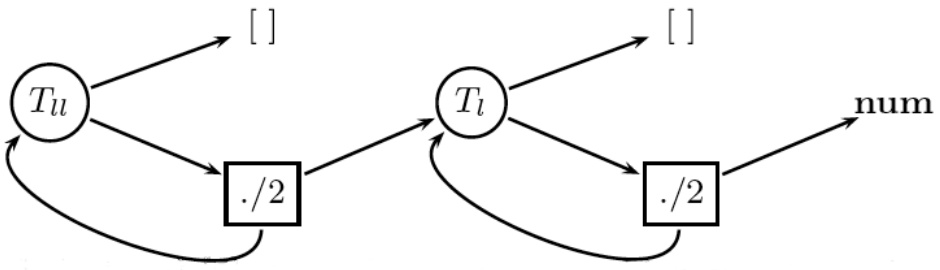
Example 25.2 Applying this widening to the previous type T_{ll}, the following will be obtained:

$$T \longrightarrow [] \mid .(T_1, T) \qquad T_1 \longrightarrow [] \mid \mathbf{num} \mid .(T_1, T)$$

Note that this widening is strictly more precise than the functor widening. In the example, the new type captures the upper level of lists, but it loses precision when describing the type of the list elements. This is due to the restriction of forcing functors to always have the same arguments.

Shortening A grammar can be seen as a graph where the nodes correspond to the non-terminals (or-nodes) and to the right hand sides of productions (and-nodes), and the edges correspond to the production relation or the relation between a functor and its arguments in a right hand side of a production. Given an or-node, its *principal functors* are the functors appearing in its children nodes.

Example 25.3 The type T_{ll} of the previous examples can be seen as the graph:



Gallagher and de Waal [65] defined a widening which avoids having two or-nodes, which have the same principal functors, connected by a path. If two such nodes exist, they are replaced by their least upper bound.

Example 25.4 In the above example graph, nodes T_{ll} and T_l have the same principal functors ([] and .) so that they are replaced, yielding:

$$T \longrightarrow [] \mid .(T_1, T) \qquad T_1 \longrightarrow [] \mid \mathbf{num} \mid .(\mathbf{num}, T)$$

Note the precision improvement with respect to the result in the previous example. Note also that still the result is imprecise.

Restricted Shortening Saglam and Gallagher [196] propose a more precise variant of the previous widening. Shortening is restricted so that two or-nodes T and T' which are connected by a path from T to T' and have the same principal functors are replaced only if $T' \sqsubseteq T$. If this is the case, only T' needs be replaced, since the least upper bound is T.

Example 25.5 Continuing previous examples, since nodes T_{ll} and T_l have the same principal functors but $T_l \not\sqsubseteq T_{ll}$, the widening operation will make no change. In this case, the most precise type is achieved.

Note, however, that restricted shortening does not guarantee termination in general (and thus, it is not, strictly speaking, a widening). There are cases in which analysis may not terminate using only this widening operator [159].

Depth Widening Janssens and Bruynooghe [98] proposed a type analysis in which the widening effect is achieved by a "pruning" of the type depth up to a certain bound. A parameter k establishes the maximum number of occurrences of a functor in-depth in a type. The idea is similar to the well-known depth-k abstraction for term structure analysis. The resulting type analysis uses normal restricted type graphs, which are basically deterministic types satisfying the depth limit. Obviously, precision of this analysis depends on the value of the parameter k.

Example 25.6 The widening of our previous type T_{ll} with k=1 will yield the same result than the functor widening (Example 25.25.1), whereas with k=2 will yield the same result as restricted shortening (Example 25.25.5).

Topological Clash Widening Van Hentenryck et al. [212] proposed the first widening operator that takes into account two consecutive approximations to the type being inferred. After merging the two —i.e., calculating their least upper bound, the result is compared with the previous approximation to try to "guess" where the type is growing. This is done by locating *topological clashes*: functors that differ or appear at different depth in each type graph. The clashes are resolved by replacing them with the recently calculated least upper bound.

Example 25.7 Consider the program:

```
sorted([]).
sorted([_X]).
sorted([X,Y|L]):- X =< Y, sorted([Y|L]).
```

and the moment during analysis when the final widening is performed. The resulting type for the argument of `sorted/1` is the one on the left below for the first two clauses, and the one on the right for the last one:

$$T_0 \longrightarrow [] \mid .(\mathbf{any}, [])$$

$$T_1 \longrightarrow .(\mathbf{num}, .(\mathbf{num}, T_l))$$
$$T_l \longrightarrow [] \mid .(\mathbf{num}, T_l)$$

Their least upper bound is T_u on the left below, which exhibits a clash with T_0 in the second argument of functor $./2$. Thus, the result of widening is T_s:

$$T_u \longrightarrow [] \mid .(\mathbf{any}, T_l) \qquad T_s \longrightarrow [] \mid .(\mathbf{any}, T_s)$$

All widening operators are based on locating recursive structures in the type definitions where there are nodes with the same functors. This may originate an unnecessary loss of precision, since the widening may impose a recursive structure on the resulting type in argument positions where the concrete program is in fact not recursive. In the following section we present a new widening operator that tries to remedy this problem.

26 Structural Type Widening

In this section we define an extended domain for type analysis which incorporates a widening operator aimed at improving the precision of the analysis. The domain is defined so as to keep track of information on the program structure, so that recursion on the types produced by the analysis is imposed by the widening operator only in the cases where it corresponds to a recursive structure in the program being analyzed. To this end, type names will be used.

A *type name* is roughly a (distinguished) non-terminal that represents a type produced during the analysis. Type names are created for each variable in each argument of each variant of

each program atom for each predicate (note how this is different from, for example, set-based analyses [22], where variants are not taken into account).

Type names provide information on how types are being formed from other types during analysis. This makes it possible to precisely identify places where to impose recursion on the types: in a subterm of the type which happens to refer to the name of that type. To this end, type names contain references to the position of its constituent types. To determine positions, selectors are used, as defined below.

Definition 26.1[selector] Define t/s, the subterm of a concrete term t referenced by a *selector* s, inductively as follows. The empty selector ϵ refers to the term t, that is, $t/\epsilon = t$. If $t/s = t'$, t' is a compound term $f(t_1, \ldots, t_i, \ldots, t_n)$ (where f is an n-ary function symbol) then $t/s \cdot (f.i) = t_i$, $1 \leq i \leq n$.

For every two selectors s, p, if $t/s = t'$ and if t'/p exists then $t/s \cdot p = t'/p$. The initial ϵ of a non-empty selector will often be omitted, so $\epsilon \cdot p$ will be written simply as p.

We define a set of type names $\mathcal{N}$ such that $\mathcal{N} \cap \mathcal{G} = \emptyset$ and a set $2^{\mathcal{N}\times\mathcal{G}}$ of relations $\mathcal{X} \in 2^{\mathcal{N}\times\mathcal{G}}$ between type names and types, of the form $\mathcal{X} \subseteq \mathcal{N} \times \mathcal{G}$.

Definition 26.2[label] Let $\mathcal{X}$ a relation between type names and types. Given a type name N, a selector s, and a type name N', a tuple $\langle s, N' \rangle$ is a *label* of N iff $(N, T) \in \mathcal{X}$, $(N', T') \in \mathcal{X}$, and $T' \sqsubseteq T/s$.

Labels of a type name N indicate subterms of the type T defining N where other type names occur.

Example 26.3 Let a relation $\mathcal{X}$ such that $\{(A, T_1), (B, T_2)\} \subseteq \mathcal{X}$, and let grammars $(T_1, \mathcal{T}_1, \mathcal{F}, \mathcal{R}_1)$ and $(T_2, \mathcal{T}_2, \mathcal{F}, \mathcal{R}_2)$, such that the only rule for T_1 is $(T_1 \longrightarrow f(b)) \in \mathcal{R}_1$ and $(T_2 \longrightarrow g(c, T_3)) \in \mathcal{R}_2$, $(T_3 \longrightarrow b \mid f(b)) \in \mathcal{R}_2$. Consider a label $\langle (g.2), A \rangle$ of B. We have that $T_1 \sqsubseteq T_2/(g.2) = T_3$.

Definition 26.4[type descriptor] Let $\mathcal{G}$ a set of types (regular term grammars), $\mathcal{N}$ a set of type names, and $\mathcal{X} \subseteq \mathcal{N} \times \mathcal{G}$. A *type descriptor* is a tuple (N, E, T) where $N \in \mathcal{N}$, $T \in \mathcal{G}$, $(N, T) \in \mathcal{X}$, and E is a set of labels of N.

In the new domain, type descriptors will be used instead of types. Let $\mathcal{D}$ be the set of all type descriptors from given sets of types $\mathcal{G}$ and of type names $\mathcal{N}$. Concretization is defined as $\gamma((N, E, T)) = \gamma(T)$. The domain ordering and operations on $\mathcal{D}$ are the same as on $\mathcal{G}$ except for type names. In this case, they have to take into account the possible labels of the type name.[21]

[21]Note that these operations do not manipulate the type names: they are assigned independently during analysis. In particular, the name N of the type resulting from union and intersection is always a new name.

Inclusion $(N_1, E_1, T_1) \sqsubseteq (N_2, E_2, T_2) \Leftrightarrow T_1 \sqsubseteq T_2 \wedge E_1 \subseteq E_2$.

Union $(N, E, T) = (N_1, E_1, T_1) \sqcup (N_2, E_2, T_2) \Leftrightarrow T = T_1 \sqcup T_2 \wedge E = E_1 \cup E_2$.

Intersection $(N, E, T) = (N_1, E_1, T_1) \sqcap (N_2, E_2, T_2) \Leftrightarrow T = T_1 \sqcap T_2 \wedge E = E_1 \cup E_2$.

Again, we may be interested in types defined by non-terminals other than the distinguished non-terminal T of a grammar $(T, \mathcal{T}, \mathcal{F}, \mathcal{R})$. A type descriptor (N_i, E_i, T_i), where $T_i \in \mathcal{T}$, is formally defined from (N, E, T) as follows: T_i is the grammar of Equation 9, N_i is a new type name, and

$$E_i = \{\langle p, N' \rangle \mid \langle s \cdot p, N' \rangle \in E \wedge T/s = T_i\}.$$

Abstract substitutions for variables of interest $\{x_1, \ldots, x_n\}$ are now defined as tuples of the form $\langle (N_1, E_1, T_{x_1}), \ldots, (N_n, E_n, T_{x_n}) \rangle$. Concretization and the domain ordering and operations are lifted to abstract substitutions element-wise, in the same way as in Section 24, including the widening operator defined below. If now Ω is the domain of type descriptors, it can be shown that $(2^{\Theta}, \alpha, \Omega, \gamma)$ is a Galois insertion, where α is the adjoin of γ. Abstract unification is defined as in Equation 10, but using type descriptors instead of types. During unification, all type names in the "input" abstract substitution T^n to $amgu$ are preserved; in the labels, the selectors for those names are changed so as to refer to the resulting type graph instead of to that of T^n.

Definition 26.5[structural widening] The widening between an approximation T_2 to type name N and a previous approximation T_1 to N is $(N, E_1, T_1) \bigtriangledown (N, E_2, T_2) = (N, E_1 \cup E_2, T)$, such that T is defined by $(T, \mathcal{T}, \mathcal{F}, \mathcal{R})$ where $\mathcal{T} = \{T_i \mid T \longrightarrow^*_{\mathcal{R}} T_i\}$, and $\mathcal{R}$ is obtained by the following algorithm:

T' := $T_1 \sqcup T_2$ defined by $(T', \mathcal{T}', \mathcal{F}, \mathcal{R}')$
$\mathcal{S}$:= $\{s \mid (s, N) \in E_1 \cup E_2\}$
$Seen := \emptyset$
`for each` $(T' \longrightarrow f(A_1, \ldots, A_n)) \in \mathcal{R}'$ `add to` $\mathcal{R}$ `production`
    $T \longrightarrow f($`widen`$(A_1, \mathcal{R}', (f.1)), \ldots,$ `widen`$(A_n, \mathcal{R}', (f.n)))$

`widen`$(N, \mathcal{R}', Sel)$:
    `if` N = **any** `return` **any**
    `if` $\exists M \langle N, M \rangle \in Seen$ `return` M
    `let` M `a new non-terminal`
    $Seen$ `:=` $Seen \cup \{\langle N, M \rangle\}$

```
for each (N ⟶ f(A_1,...,A_n)) ∈ R' add to R production
      M ⟶ f(widen(A_1,R',Sel·(f.1)),...,widen(A_n,R',Sel·(f.n)))
if Sel ∈ S then
   add to R production M ⟶ T
return M
```

Structural widening basically identifies subterms of the new type $T_1 \sqcup T_2$ where a reference to the type N being widened appears, and makes this "self-reference" explicit in the definition of the new type. Note that the widening operation starts with the least upper bound and, basically, adds new grammar rules to that type. Therefore, the result is always a correct approximation of such an upper bound. This justifies its correctness. Moreover, this approach based on type names is potentially more precise than any of the previous widening operators discussed, as the following examples show:

Example 26.6 Consider program `sorted` in Example 25.25.7. A top-down analysis with topological clash was roughly described there. Let us now look at analysis using restricted shortening. The resulting type happens to be the same one.

Analysis of program atom `sorted([Y|L])` approximates variable `Y` always as $\mathbf{num}$, both in the calls and in the successes. The first two success approximations for variable `L` are $[]$ and $.(\mathbf{num}, [])$. Their lub (and widening) is:

$$T_1 \quad \longrightarrow \quad [] \mid .(\mathbf{num}, [])$$

The next approximation to the type of `L` is $.(\mathbf{num}, T_1)$. Its lub with T_1 is $T_2 \longrightarrow [] \mid .(\mathbf{num}, T_1)$, and since T_2 and T_1 have the same functors, and T_1 is included in T_2, the widening of T_2 is:

$$T_3 \longrightarrow [] \mid .(\mathbf{num}, T_3)$$

i.e., list of numbers. The next approximation to the type of `L` is $.(\mathbf{num}, T_3)$ (i.e., a list with at least one number). It is included in T_3, so fixpoint is reached.

The success of principal goal `sorted(X)` is approximated after analyzing the two non-recursive clauses by $T_4 \longrightarrow [] \mid .(\mathbf{any}, [])$. Analysis of the third clause yields $.(\mathbf{num}, .(\mathbf{num}, T_3))$. Its lub with T_4 is $T_5 \longrightarrow [] \mid .(\mathbf{any}, T_3)$. The widening of T_5 finds that T_5 and T_3 have the same functors and $T_3 \sqsubseteq T_5$, since **num** $\sqsubseteq$ **any**. Thus, the result of widening is:

$$T_6 \longrightarrow [] \mid .(\mathbf{any}, T_6)$$

i.e., list of terms. This is the final result after one more iteration. Note that the information about successes where the tail of lists of length greater than one is a list of numbers is lost.

Let us now consider structural widening. Analysis of atom `sorted([Y|L])` always approximates the type of `Y` by $(N_{13}, \emptyset, \mathbf{num})$. For variable `L` the two first approximations are $(N_{14}, \emptyset, [])$ and $(N_{14}, E_{14}, .(\mathbf{num}, []))$, where the set of labels is:

$$E_{14} = \{\ (\texttt{'.'}.1, N_{13}),\ (\texttt{'.'}.2, N_{14})\ \}$$

The result of widening is (N_{14}, E_{14}, T_1), where T_1 is defined as:

$$T_1 \ \longrightarrow\ [] \mid .(\mathbf{num}, T_1)$$

i.e., list of numbers. This is the final result after one more iteration.

The success of principal goal `sorted(X)` is approximated after analyzing the two non-recursive clauses by $(N_3, \emptyset, T_2)$ where $T_2 \longrightarrow [] \mid .(\mathbf{any}, [])$. Analysis of the third clause yields $(N_3, E_3, .(\mathbf{num}, .(\mathbf{num}, T_1)))$, where

$$E_3 \ =\ \{\ (\texttt{'.'}.2 \cdot \texttt{'.'}.1, N_{13}),\ (\texttt{'.'}.2 \cdot \texttt{'.'}.2, N_{14})\ \}$$

Its widening with the previous approximation T_2 is (N_3, E_3, T_3), where

$$T_3 \ \longrightarrow\ [] \mid .(\mathbf{any}, T_1)$$

which amounts to their lub, since the widening operator does not produce any change, because N_3 is not among its own labels. Therefore, the final result, after one more iteration, is T_3, where indeed lists of length greater than one have a tail which is a list of numbers.

However, structural widening does not guarantee termination. It is effective as long as the new approximation is built from the previous approximation of the type being inferred. This case is identified, in essence, by locating a reference to the type name of the previous approximation within the definition of the new one. However, there are contrived cases in which a type is constructed during analysis which loses the reference to the previous approximation. In these cases, a more restrictive widening has to be applied to guarantee termination.

Example 26.7 Consider the program:

```
main:- p(a).      p(a).                    q(a,f(a)).
                  p(X):- q(X,Y), p(Y).     q(f(Z),f(L)):- q(Z,L).
```

The calling substitution for atom `p(Y)` is the sequence

$$T_1 \ \longrightarrow\ f(a) \qquad T_2 \ \longrightarrow\ f(f(a)) \qquad T_3 \ \longrightarrow\ f(f(f(a))) \qquad \ldots$$

whereas the type $T \longrightarrow f(a) \mid f(T)$ correctly describes such calls. However, the analysis is not able to infer such a type.

The problem in the above example is that none of the approximations T_i contains a reference to the previous approximation. This is originated in the program fact for predicate `q/2` which causes the loss of the reference to the previous approximation because of the double occurrence of constant `a`.

In our analysis, termination is guaranteed by a bound on the number of times the widening operation can be applied to a type name. A counter is associated to each type name, so that when the bound is reached a more restrictive widening that guarantees termination is applied.

27 Type Inference Analysis Results

We have implemented analyses based on most of the widenings discussed in this paper, including structural widening. The implementation is in Prolog and has been incorporated to the CiaoPP system [85, 84], which uses the top-down analysis algorithm of PLAI. The analysis of [65], based on regular approximations, which uses a bottom-up algorithm, is also incorporated into the system. This analysis uses shortening. We want to compare the top-down and bottom-up approaches with the same widening and similar implementation technology,[22] as well as the precision and efficiency, within the same analysis framework, of the widening operators previously discussed.

We have used two sets of benchmark programs: the one used in the PLAI framework and that used in the GAIA [23] framework. A summary of the benchmarking follows. The analysis times in miliseconds are shown in Table 3. The first column (`rul`) is for the regular approximation analysis and the other three for the PLAI-based analyses: column `short` for shortening, column `clash` for topological clash, and column `struct` for structural widening.

Table 4 shows results in terms of precision. The precision of `struct` is never improved by any of the others. The improved precision of `struct` has been measured as follows. The left subcolumns under `rul`, `short`, and `clash` show the number of types with a more precise definition inferred by `struct`. The right subcolumns show the number of types where the previous ones appear (and are thus, also, more precise). The former are types directly inferred from program predicates; the latter are types which are defined from the former, due to the data flow in the program.

The following conclusions can be drawn from the tables. First, the regular approximation approach seems to behave better in terms of efficiency than the program interpretation approach, at least for the bigger programs. This conclusion, however, has to be taken with some care, since

[22] Similar in the programming technique. Of course, the regular approximation method is rather different from the method of program interpretation on an abstract domain: Evaluating this difference is part of the aim of the comparison.

Program	rul	short	clash	struct
aiakl	568	469	529	900
bid	1480	2209	2529	4730
boyer	3450	3890	4989	9629
browse	758	380	389	539
cs_o	3840	1889	2689	2580
cs_r	18549	10720	24479	19560
disj_r	4468	1819	6399	2440
gabriel	1549	1430	1870	1760
grammar	330	160	160	190
hanoiapp	620	719	1889	1150
kalah_r	1520	79	79	89
mmatrix	310	190	209	119
occur	380	219	330	289
palin	590	840	980	850
pg	839	2020	2980	3990
plan	1138	819	960	1009
progeom	979	1840	2530	3640
qsort	310	590	659	680
qsortapp	369	1000	2898	1210
queens	329	179	190	180
query	720	360	370	410
serialize	478	810	969	899
witt	2929	4890	1399	1169
zebra	560	3490	14958	12830

Table 3: Timing results

the current implementation of `rul` performs some caching of the type grammars that the PLAI-based analysis does not. This should be subject of a more thorough evaluation, which is out of the scope of this paper. The fact that it improves in bigger programs seems to suggest that the effect of this caching is most surely not negligible.

Regarding the analyses based on program interpretation, it can be concluded that the better the precision the worse the efficiency: `short` takes less than `clash`, and this one takes less than `struct`; this one is more precise than `clash`, which is more precise than `short`. This conclusion seems evident at first sight, but it is not: in analysis, an improvement in precision can very well trigger an improvement in efficiency. This can also be seen in the tables in some cases, the most significant probably being `zebra`. Overall, one can arguably conclude that the efficiency loss found is not a high price in exchange for the gain in precision.

Program	rul		short		clash	
aiakl	1	1	1	1		
bid	9	12	9	12		
cs_o	4	18	4	18	2	9
cs_r	4	28	4	28	2	19
disj_r	6	13	6	13		
mmatrix	2	2	2	2		
occur	1	1	1	1		
palin	2	4	2	4		
pg	1	1	1	1		
qsort	1	1	1	1		
serialize	2	4	2	4		
zebra	3	3	3	3	1	1

Table 4: Precision results

We have also carried out another test. For practical purposes, the CiaoPP system includes a back-end to the analysis that simplifies the types inferred, in the sense that equivalent types are identified, so that they are then reduced to a single type. This facilitates the interpretation of the output. It is the case that the structural widening includes certain amount of type simplification, so that the analysis creates less different types which are in fact equivalent. For this reason, we have included the same tests as above, but adding now the times taken in the back-end simplification phase.

The times including the simplification are shown in table 5. The columns read as before. It can be seen that in this case structural widening outperforms all of the other analyses, except, in some cases, `rul`. It can also be observed that `rul` behaves usually better than `short` also when simplification is included. This seems to suggest that incorporating our widening into the regular approximation approach would probably give the best results in practice.[23]

28 Convex Hull Abstractions in Specialization of CLP Programs

Program specialization is sometimes regarded as being achieved in three phases: *pre-processing of the program*, *analysis* and *program generation*. During pre-processing the input program may be subject to some minor syntactic analyses or changes, ready for the analysis phase. The analysis computes some data-flow and control-flow information from the program and the spe-

[23]This, however, may not be trivial. It is subject for future work.

Program	rul	short	clash	struct
aiakl	697	3009	3738	1409
bid	2899	31278	35949	15259
boyer	19620	201169	206917	92117
browse	987	2848	2987	1698
cs_o	11958	17389	32959	4878
cs_r	50760	303430	238788	30169
disj_r	6508	18598	26077	6408
gabriel	2098	13388	22379	5208
grammar	759	3169	3169	1279
hanoiapp	840	3988	13738	3378
kalah_r	2069	1187	1188	888
mmatrix	757	1769	2078	488
occur	530	1647	2628	767
palin	997	8520	11878	2180
pg	1349	15380	22870	7370
plan	1587	6167	6559	2288
progeom	1358	12800	17598	6679
qsort	520	3439	4168	1409
qsortapp	569	7789	9669	2900
queens	457	1128	1138	429
query	1627	22458	22788	11818
serialize	937	8429	11957	2217
witt	3438	188419	42699	25709
zebra	717	55100	189949	44540

Table 5: Timing results (including simplification)

cialization query. Finally, at program generation time the result of the analysis is used to produce a residual program reflecting the result of the analysis. In off-line specialization the three phases are consecutive, whereas in on-line specialization and driving the analysis and program generation phases are merged or interleaved.

The use of abstract interpretation techniques to assist program specialization is well-established [63, 57, 104, 123, 193] and goes back to the invention of binding time analysis to compute static-dynamic annotations [100]. More complex and expressive abstract domains have been used such as regular tree structures [162, 64, 75, 132].

In this paper we focus on an abstract domain based on arithmetic constraints. Abstract interpretations based on arithmetic constraints have already been developed [38, 7, 197]. We show how the analysis phase of a specialization algorithm can benefit from advances made in that field.

We introduce an abstract domain consisting of atomic formulas constrained by linear arithmetic constraints (or convex hulls [38]). The operations on this domain are developed from a standard constraint solver.

We then employ this domain within a generic algorithm for specialization of (constraint) logic programs [75]. The algorithm combines analysis over an abstract domain with partial evaluation. Its distinguishing feature is the analysis of the success constraints (or answer constraints) as well as the call constraints in a computation. This allows us to go beyond the capability of another recent approach to use a linear constraint domain in constraint logic program specialization [51].

The specialization method can also be used for ordinary logic programs containing arithmetic, as well as constraint logic programs. We can reason in constraint terms about the arithmetic expressions that occur in logic programs, treating them as constraints (for instance `X is Expr` is treated as $\{$`X = Expr`$\}$). In addition, the algorithm provides a contribution to the growing field of using specialization for model checking infinite state systems [141].

28.1 A Constraint Domain

Approximation in program analysis is ubiquitous, and so is the concept of a domain of properties. The analysis phase of program specialization is no exception.

Linear Arithmetic Constraints Our constraint domain will be based on linear arithmetic constraints, that is, conjunctions of equalities and inequalities between linear arithmetic expressions. The special constraints true and false are also included. This domain has been used in the design of analysers and for model checking infinite state systems. Here we use it for specialization of (constraint) logic programs.

Let Lin be the theory of linear constraints over the real numbers. Let C and D be two linear constraints. We write $C \sqsubseteq D$ iff $\mathsf{Lin} \models \forall(C \rightarrow D)$. C and D are *equivalent*, written $C \equiv D$, iff $C \sqsubseteq D$ and $D \sqsubseteq C$. Let C be a constraint and V be a set of variables. Then $\mathsf{project}_V(C)$ is the projection of constraint C onto the variables V; the defining property of projection is that $\mathsf{Lin} \models \forall V(\exists V'.C \leftrightarrow \mathsf{project}_V(C))$, where $V' = \mathsf{vars}(C) \setminus V$. Given an expression e let us denote $\mathsf{vars}(e)$ as the set of variables occurring in e. If $\mathsf{vars}(e) = V$, we sometimes refer to $\mathsf{project}_e(C)$ rather than $\mathsf{project}_V(C)$ when speaking of the projection of C onto the variables of e.

Arithmetic constraints can be presented in their simplified form, removing redundant constraints. Constraint simplification serves as a satisfiability check: the result of simplifying a constraint is false if and only if the constraint is unsatisfiable. If a constraint C is satisfiable, we write $\mathsf{sat}(C)$. Because we used the CLP facilities of SICStus Prolog all these operations (projection, simplification and checking for equivalence) are provided for the domain of linear

constraints over rationals and reals. We refer the interested reader to a survey on CLP [96] for a thorough discussion on the subject.

Intuitively, a constraint represents a convex polyhedron in cartesian space, namely the set of points that satisfy the constraint. Let S be a set of linear arithmetic constraints. The *convex hull* of S, written $\mathsf{convhull}(S)$, is the least constraint (with respect to the $\sqsubseteq$ ordering on constraints) such that $\forall S_i \in S.S_i \sqsubseteq \mathsf{convhull}(S)$. So $\mathsf{convhull}(S)$ is the smallest polyhedron that encloses all members of S. Further details and algorithms for computing the convex hull can be found in the literature [38].

Constrained Atoms and Conjunctions Now we must define our abstract domain. It consists of equivalence classes of *c-atoms*, which are constrained atoms. Each c-atom is composed of two parts, an atom and a linear arithmetic constraint.

Definition 28.1 [c-atoms and c-conjunctions] A c-conjunction is a pair $\langle B, C\rangle$; B denotes a conjunction of atomic formulas (atoms) and C a conjunction of arithmetic constraints, where $\mathsf{vars}(C) \subseteq \mathsf{vars}(B)$. If B consists of a single atom the pair is called a c-atom.

(Note that c-conjunctions are defined as well as c-atoms, since they occur in our algorithm. However, the domain is constructed only of c-atoms).

Given any arithmetic constraint C and atom A, we can form a c-atom $\langle A, C'\rangle$, where $C' = \mathsf{project}_A(C)$. Any atom A can be converted to a c-atom $\langle A', C\rangle$ by replacing each non-variable arithmetic expression occurring in A by a fresh variable[24], obtaining A'. Those expressions which were replaced together with the variables that replace them are added as equality constraints to the constraint part C of the c-atom. For example, the c-atom obtained from $p(f(3), Y+1)$ is $\langle p(f(X_1), X_2), (X_1 = 3, X_2 = Y+1)\rangle$.

A c-atom represents a set of concrete atoms. We define the *concretization* function γ as follows.

Definition 28.2 [γ]

Let $\mathcal{A} = \langle A, C\rangle$ be a c-atom. Define the concretization function γ as follows.

$$\gamma(\mathcal{A}) = \left\{ A\theta \;\middle|\; \begin{array}{l} \theta \text{ is a substitution} \wedge \\ \forall\varphi.\mathsf{sat}(C\theta\varphi) \end{array} \right\}$$

γ is extended to sets of c-atoms: $\gamma(S) = \bigcup\{\gamma(\mathcal{A}) \mid \mathcal{A} \in S\}$.

[24] By parsing the arguments the desired terms can be selected.

There is a partial order on c-atoms defined by $\mathcal{A}_1 \sqsubseteq \mathcal{A}_2$ if and only if $\gamma(\mathcal{A}_1) \subseteq \gamma(\mathcal{A}_2)$. Two c-atoms $\mathcal{A}_1$ and $\mathcal{A}_2$ are equivalent, written $\mathcal{A}_1 \equiv \mathcal{A}_2$ if and only if $\mathcal{A}_1 \sqsubseteq \mathcal{A}_2$ and $\mathcal{A}_2 \sqsubseteq \mathcal{A}_1$. Equivalence can also be checked using syntactic comparison of the atoms combined with constraint solving, using the following lemma.

Lemma 28.3 Let $\mathcal{A}_1 = \langle A_1, C_1 \rangle$ and $\mathcal{A}_2 = \langle A_2, C_2 \rangle$ be two c-atoms. Let $\langle \bar{A}_1, \bar{C}_1 \rangle$ and $\langle \bar{A}_2, \bar{C}_2 \rangle$ be the c-atoms obtained by removing repeated variables from A_1 and A_2 and adding constraints to C_1 and C_2 in the following manner. If a variable X occurs more than once in the atom, then one occurrence is replaced by a fresh variable W and the constraint $X = W$ is added to the corresponding constraint part.

Then $\mathcal{A}_1 \equiv \mathcal{A}_2$ if and only if there is a renaming substitution θ such that $\bar{A}_1\theta = \bar{A}_2$ and $\bar{C}_1\theta \equiv \bar{C}_2$.

Now we are in a position to define the domain and the operations on the elements of our domain. The relation $\equiv$ on c-atoms is an equivalence relation. The abstract domain consists of equivalence classes of c-atoms. For practical purposes we consider the domain as consisting of *canonical* constrained atoms, which are standard representative c-atoms, one for each equivalence class. These are obtained by renaming variables using a fixed set of variables, and representing the constraint part in a standard form. Hence we speak of the domain operations as being on c-atoms, whereas technically they are operations on equivalence classes of c-atoms.

Next we define the upper bound of c-atoms which combines the *most specific generalization operator (msg)* [194] on terms and the *convex hull* [38] on arithmetic constraints. The idea is to compute the *msg* of the atoms, and then to rename the constraint parts suitably, relating the variables in the original constraints to those in the *msg*, before applying the convex hull operation.

The following notation is used in the definition. Let θ be a substitution whose range only contains variables; the domain and range of θ are $\mathsf{dom}(\theta)$ and $\mathsf{ran}(\theta)$ respectively. $\mathsf{alias}(\theta)$ is the conjunction of equalities $X = Y$ such that there exist bindings X/Z and Y/Z in θ, for some variables X, Y and Z. Let $\bar{\theta}$ be any substitution such that $\mathsf{dom}(\bar{\theta}) = \mathsf{ran}(\theta)$ and $X\bar{\theta}\theta = X$ for all $X \in \mathsf{ran}(\theta)$. (That is, $\bar{\theta} = \varphi^{-1}$ where φ is some bijective subset of θ with the same range as θ).

The following definition is a reformulation of the corresponding definition given previously [197].

Definition 28.4 [Upper bound of c-atoms, $\sqcup$] Let $\mathcal{A}_1 = \langle A_1, C_1 \rangle$ and $\mathcal{A}_2 = \langle A_2, C_2 \rangle$ be c-atoms. Their upper bound $\mathcal{A}_1 \sqcup \mathcal{A}_2$ is c-atom $\mathcal{A}_3 = \langle A_3, C_3 \rangle$ defined as follows.

1. $A_3 = \mathsf{msg}(A_1, A_2)$, where $\mathsf{vars}(A_3)$ is disjoint from $\mathsf{vars}(A_1) \cup \mathsf{vars}(A_2)$.

2. Let $\theta_i = \{X/U \mid X/U \in \mathsf{mgu}(A_i, A_3), U \text{ is a variable}\}$, for $i = 1, 2$. Then $C_3 = \mathsf{project}_{A_3}(\mathsf{convhull}(\{\mathsf{alias}(\theta_i) \cup C_i\bar{\theta}_i \mid i = 1, 2\}))$.

$\sqcup$ is commutative and associative, and we can thus denote by $\sqcup(S)$ the upper bound of the elements of a set of c-atoms S.

Example 28.5 Let $\mathcal{A}_1 = \langle p(X, X), X > 0\rangle$ and $A_2 = \langle p(U, V), -U = V\rangle$. Then $\mathcal{A}_1 \sqcup \mathcal{A}_2 = \langle p(Z_1, Z_2), Z_2 \geq -Z_1\rangle$. Here, $\mathsf{mgu}(p(X, X), p(U, V)) = p(Z_1, Z_2)$, $\theta_1 = \{Z_1/X, Z_2/X\}$, $\theta_2 = \{Z_1/U, Z_2/V\}$, $\mathsf{alias}(\theta_1) = \{Z_1 = Z_2\}$, $\mathsf{alias}(\theta_2) = \emptyset$, $\bar{\theta}_1 = \{X/Z_1\}$ (or $\{X/Z_2\}$) and $\bar{\theta}_2 = \{U/Z_1, V/Z_2\}$. Hence we compute the convex hull of the set $\{(Z_1 = Z_2, Z_1 > 0), (-Z_1 = Z_2)\}$, which is $Z_2 \geq -Z_1$.

Like most analysis algorithms, our approach computes a monotonically increasing sequence of abstract descriptions, terminating when the sequence stabilizes at a fixed point. Because infinite ascending chains may arise during specialization it is not enough to have an upper bound operator, in order to reach a fixpoint. An operator called *widening* may be interleaved with the upper bound to accelerate the convergence to a fixpoint and ensure termination of an analysis based on this domain. When widening we assume that the c-atoms can be renamed so that their atomic parts are identical, and the widening is defined solely in terms of widening of arithmetic constraints, ∇_c [38]. This is justified since there are no infinite ascending chains of atoms with strictly increasing generality. Hence the atom part of the c-atoms does not require widening.

Definition 28.6 [Widening of c-atoms, ∇] Given two c-atoms $\mathcal{A}_1 = \langle A_1, C_1\rangle$ and $\mathcal{A}_2 = \langle A_2, C_2\rangle$, where A_1 and A_2 are variants, say $A_2 = A_1\rho$. The widening of $\mathcal{A}_1$ and $\mathcal{A}_2$, denoted as $\mathcal{A}_1 \nabla \mathcal{A}_2$ is c-atom $\mathcal{A}_3 = \langle A_2, C_3\rangle$ where $C_3 = C_1\rho \, \nabla_c \, C_2$.

For instance, the widening of $\langle p(X), X \geq 0, X \leq 1\rangle$ and $\langle p(Y), Y \geq 0, Y \leq 2\rangle$ is $\langle p(Y), Y \geq 0\rangle$.

29 An Algorithm for Specialization with Constraints

In this section we describe an algorithm for specialization, incorporating operations on the domain of convex hulls. The algorithm is based on one presented previously [75], where we used a domain of regular trees in place of convex hulls, and the operations named ω, calls and answers are taken from there. The operations ω and aunf^* (which is used in the definition of calls) were taken from Leuschel's top-down abstract specialization framework [123]. The answer propagation aspects of our algorithm are different from Leuschel's answer propagation method, though.

There is no counterpart of the answers operation in Leuschel's framework. The differences between the approaches were discussed in our previous work [75].

The structure of the algorithm given in Figure 12 is independent of any particular domain of descriptions such as regular types or convex hulls. The operations concerning convex hulls appear only within the domain-specific operations calls, ω, ∇ and answers.

INPUT: a program P and a c-atom $\mathcal{A}$
OUTPUT: two sets of c-atoms (calls and answers)

begin
    S_0 := $\{\mathcal{A}\}$
    T_0 := $\{\}$
    i := 0
    repeat
        S_{i+1} := $\omega(\mathsf{calls}(S_i, T_i), S_i)$
        T_{i+1} := $T_i \nabla \mathsf{answers}(S_i, T_i)$
        i := $i + 1$
    until $S_i = S_{i-1}$ and $T_i = T_{i-1}$
end

Figure 12: Partial Evaluation Algorithm with Answer Propagation

29.1 Generation of Calls and Answers

The idea of the algorithm is to accumulate two sets of c-atoms. One set represents the set of *calls* that arise during the computation of the given initial c-atom $\mathcal{A}$. The other set represents the set of *answers* for calls.

At the start, the set of calls S_0 contains only the initial goal c-atom, and the set of answers T_0 is empty. Each iteration of the algorithm extends the current sets S_i and T_i of calls and answers. The diagram in Figure 13 illustrates the process of extending the sets. All existing calls $\mathcal{A} = \langle A, C\rangle \in S_i$ are unfolded according to some unfolding rule. This yields a number of *resultants* of the form $(A, C)\theta \leftarrow B_1, \ldots, B_l, C'$, where $A\theta \leftarrow B_1, \ldots, B_l$ is a result of unfolding A and C' is the accumulated constraint; that is, C' is the conjunction of $C\theta$ and the other constraints introduced during unfolding. If $\mathsf{sat}(C')$ is false then the resultant is discarded. The unfolding process is performed in the algorithm by the operation aunf^*, defined as follows.

Definition 29.1 [$\mathsf{aunf}, \mathsf{aunf}^*$**]** Let P be a definite constraint program and $\mathcal{A} = \langle A, C\rangle$ a c-atom. Let $\{A\theta_1 \leftarrow L_1, C_1, \ldots, A\theta_n \leftarrow L_n, C_n\}$ be some partial evaluation [149] of A in P, where

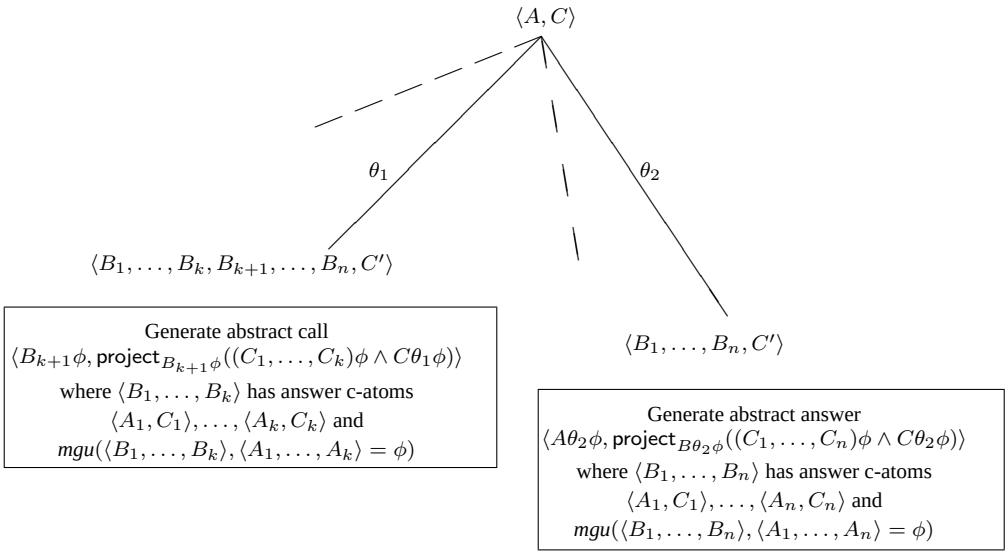


Figure 13: The generation of calls and answers

$C_i, L_i (1 \leq i \leq n)$ are the constraint and non-constraint parts respectively of each resultant body. Then define

$$\mathsf{aunf}(\mathcal{A}) = \left\{ A\theta_i \leftarrow L_i, (C_i \wedge C\theta_i) \;\middle|\; 1 \leq i \leq n, \mathsf{sat}(C_i \wedge C\theta_i) \right\}.$$

Let S be a set of c-atoms. We define $\mathsf{aunf}^*(S)$ as:

$$\mathsf{aunf}^*(S) = \left\{ (L, project_L(C')) \;\middle|\; \begin{array}{l} \langle A, C\rangle \in S \\ A\theta \leftarrow L, C' \in \mathsf{aunf}(\mathcal{A}) \end{array} \right\}$$

In the following examples, assume that the unfolding rule selects the leftmost atom provided that it matches at most one clause (after discarding matches that result in an unsatisfiable constraint), otherwise selects no atom.

Example 29.2 Consider the following simple program P.

```
s(X,Y,Z) <-
  p(X,Y,Z), q(X,Y,Z)
p(0,0,0) <-
p(X,Y,Z) <-
  {X=X1+1, Y=Y1+1, Z=Z1+1},
  p(X1,Y1,Z1)
```

```
q(0,Z,Z) <-
  {Z > 0}
q(X,Y,Z) <-
  {X = X1+1, Z=Z1+1},
  q(X1,Y,Z1)
```

Let S be $\{\langle \mathtt{s(X,Y,Z)}, \mathtt{X} > 2\rangle\}$. Then $\mathsf{aunf}^*(S) = \{(\mathtt{p(X1,Y,Z1)}, \mathtt{q(X,Y,Z)}, (\mathtt{X} > 2, \mathtt{X} = \mathtt{X1} + 3, \mathtt{Z} = \mathtt{Z1} + 3)\}$. The unfolding rule results in four steps: the unfolding of the atom `s(X,Y,Z)` followed by three unfoldings of `p`, since the initial constraint $\mathtt{X} > 2$ implies that the base case `p(0,0,0)` cannot be matched so long as the first argument of `p` is greater than zero.

Note that the range of the function aunf^* is the set of c-conjunctions. The current answers from T_i are then applied, from left to right, to the c-conjunctions generated by aunf^*. If there is some prefix $B_1 \ldots, B_k$ ($k < l$) in a c-conjunction, having a solution in T_i, then a call to an instance of B_{k+1} is generated. More precisely, we define a function calls as follows. We first define the notion of a "solution" of a conjunction with respect to a set of c-atoms.

Definition 29.3 [solution of a conjunction] Let $(B_1, \ldots, B_l)$ be a conjunction of atoms and T be a set of c-atoms. Then $\langle \varphi, \bar{C} \rangle$ is a solution for $(B_1, \ldots, B_l)$ in T if there is a sequence of c-atoms $\langle \mathcal{A}_1, \ldots, \mathcal{A}_l \rangle$ where $\mathcal{A}_j = \langle A_j, C_j \rangle \in T$, $1 \leq j \leq l$, such that $\mathsf{mgu}((B_1, \ldots, B_l), (A_1, \ldots, A_l)) = \varphi$, and $\mathsf{sat}(\bar{C})$ (where $\bar{C} = (C_1 \wedge \cdots \wedge C_l)\varphi$).

Definition 29.4 [calls] Let S_i be a set of call c-atoms and T_i be a set of answer c-atoms. Define $\mathsf{calls}(S_i, T_i)$ to be the set of c-atoms $\langle B_{k+1}\varphi, \mathsf{project}_{B_{k+1}\varphi}(\bar{C} \wedge C'\varphi) \rangle$ where

1. $\langle B_1, \ldots, B_l, C' \rangle \in \mathsf{aunf}^*(S_i)$, and
2. there is a conjunction $(B_1, \ldots, B_k)$ ($k < l$) which has a solution $\langle \varphi, \bar{C} \rangle$ in T_i, and $\mathsf{sat}(\bar{C} \wedge C'\varphi)$.

Example 29.5 Let P be the program from Example 29.2 and let S be $\{\langle \mathtt{s}(\mathtt{X}, \mathtt{Y}, \mathtt{Z}), \mathtt{X} > 2 \rangle\}$. Let $T = \{\langle \mathtt{p}(\mathtt{X1}, \mathtt{Y1}, \mathtt{Z1}), \mathtt{X1} = 0, \mathtt{Y1} = 0, \mathtt{Z1} = 0 \rangle\}$. Then:

$$\mathsf{calls}(S, T) = \{\langle \mathtt{p}(\mathtt{X1}, \mathtt{Y}, \mathtt{Z1}), \mathsf{true} \rangle, \langle \mathtt{q}(\mathtt{X}, \mathtt{Y}, \mathtt{Z}), \mathtt{X} = 3, \mathtt{Y} = 3, \mathtt{Z} = 3 \rangle\}$$

Note that the call to q arises from applying the solution for p and simplifying the accumulated constraints.

An answer is derived by finding a resultant $A\theta \leftarrow B_1, \ldots, B_k, C'$ whose body has a solution in the current set of answers. The function answers is defined as follows.

Definition 29.6 [answers] Let S_i be a set of call c-atoms and T_i be a set of c-atoms. Define $\mathsf{answers}(S_i, T_i)$ to be the set of answer c-atoms $\langle A\theta\varphi, \mathsf{project}_{A\theta\varphi}(\bar{C} \wedge C'\varphi) \rangle$ where

1. $\mathcal{A} = \langle A, C \rangle \in S_i$, and
2. $A\theta \leftarrow B_1, \ldots, B_l, C' \in \mathsf{aunf}(\mathcal{A})$, and
3. $(B_1, \ldots, B_l)$ has a solution $\langle \varphi, \bar{C} \rangle$ in T_i, and $\mathsf{sat}(\bar{C} \wedge C'\varphi)$.

Example 29.7 Let P be the program from Example 29.2 and let S be $\{\langle \mathtt{p}(\mathtt{X}, \mathtt{Y}, \mathtt{Z}), \mathsf{true} \rangle\}$. Let $T = \{\langle \mathtt{p}(\mathtt{X1}, \mathtt{Y1}, \mathtt{Z1}), \mathtt{X1} = 0, \mathtt{Y1} = 0, \mathtt{Z1} = 0 \rangle\}$. Then $\mathsf{answers}(S, T) = \{\langle \mathtt{p}(\mathtt{X}, \mathtt{Y}, \mathtt{Z}), \mathtt{X} = 1, \mathtt{Y} = 1, \mathtt{Z} = 1 \rangle\}$.

An important feature of the algorithm is that no call to a body atom is generated until the conjunction of atoms to its left has an answer. One effect of this is to increase specialization because the propagation of answers for some atom restricts the calls to its right. Secondly, answers can only be generated for called atoms, and no answer to an atom is generated until there is an answer to the whole body of some resultant for that atom. There can exist abstract calls that have no corresponding answers; these represent concrete calls that either fail or loop. In fact, infinitely failed computations are not distinguished from finitely failed computations, with the result that programs that produce infinitely failing computations can be specialized to ones that fail finitely. The examples later in this section illustrate this point.

29.2 Approximation Using Convex Hulls and Widening

Call and answer c-atoms derived using the calls and answers functions are added to the sets S_i and T_i respectively. There is usually an infinite number of c-atoms that can be generated in this way. The purpose of the ω and ∇ functions in the algorithm is to force termination. The ω function computes a safe approximation of the calls and answers, using the *convex hull* and *widening* operations, both of which are standard in analyses based on linear arithmetic constraints.

On each iteration, the sets of call c-atoms are partitioned into sets of "similar" c-atoms. The notion of "similar" is heuristic: the only requirements are that the definition of similarity should yield a finite partition, and that all c-atoms in one subset should have the same predicate name. In our implementation we partitioned based on the *trace terms* or "unfolding patterns" of the c-atoms [72]. We assume a function that partitions a set S of c-atoms into a finite set $\{S_1, \ldots, S_m\}$ of disjoint subsets of S, and computes the upper bound of each subset. The function $\mathsf{partition}(S)$ is defined as $\mathsf{partition}(S) = \{\sqcup(S_1), \ldots, \sqcup(S_m)\}$. It is desirable though not essential that $\sqcup(S)$ belongs to the same set as S.

Even if the partition is finite, a widening is required to enforce termination. The widening step is defined between the sets of c-atoms on two successive iterations of the algorithm. Let S, S' be two sets of c-atoms, where we assume that both S and S' are the result of a partition operation. Define $S' \nabla S$ to be

$$\left\{ \mathcal{A}' \nabla \mathcal{A} \;\middle|\; \begin{array}{l} \mathcal{A}' \in S', \mathcal{A} \in S, \\ \mathcal{A}', \mathcal{A} \text{ are in the same set} \end{array} \right\} \bigcup \left\{ \mathcal{A} \;\middle|\; \begin{array}{l} \mathcal{A} \in S, \\ \nexists \mathcal{A}' \in S' \text{ in the same set as } \mathcal{A} \end{array} \right\}$$

Finally the operation ω can be defined as $\omega(S, S') = S' \nabla \mathsf{partition}(S)$. This ensures termination if the number of sets returned by partition is bounded. The definition states that each element $\mathcal{A}$ of S is replaced by the result of widening $\mathcal{A}$ with the element from S' from the same set, if such an element exists.

29.3 Generation of the Specialized Program

After termination of the algorithm, the specialized program is produced from the final sets of calls and answers S and T respectively. It consists of the following set of clauses.

$$\left\{ \mathsf{rename}(A\theta\varphi \leftarrow L\varphi, C'\varphi) \;\middle|\; \begin{array}{l} \mathcal{A} = \langle A, C\rangle \in S, \\ A\theta \leftarrow L, C' \in \mathsf{aunf}(\mathcal{A}), \\ L \text{ has solution } \langle \varphi, \bar{C}\rangle \text{ in } T, \\ \mathsf{sat}(\bar{C} \wedge C'\varphi) \end{array} \right\}$$

That is, each of the calls is unfolded, and the answers are applied to the bodies of the resultants. Note that we do not add the solution constraints $\bar{C}$ to the generated clause, so as not to introduce redundant constraints. The rename function is a standard renaming to ensure independence of different specialized versions of the same predicate, as used in most logic program specialization systems (see for example [57] for a description of the technique).

Example 29.8 Consider again the example from Example 29.2. We specialize this program with respect to the c-atom $\langle \mathtt{s}(\mathtt{X},\mathtt{Y},\mathtt{Z}), \mathsf{true}\rangle$ assuming the usual left-to-right computation rule. Note that the concrete goal `s(X,Y,Z)` does not have any solutions, although with the standard computation rule the computation is infinite.

After the first few iterations of the algorithm the answer for `p(X,Y,Z)` is computed, after widening the successive answers `p(0,0,0), p(1,1,1), p(2,2,2), ...`
This in turn generates a call to `q(X,Y,Z)`. The c-atom describing the answers for `p(X,Y,Z)` is $\langle \mathtt{p}(\mathtt{X},\mathtt{X},\mathtt{X}), \mathtt{X} \geq 0\rangle$ and thus the call $\langle \mathtt{q}(\mathtt{X},\mathtt{X},\mathtt{X}), \mathtt{X} \geq 0\rangle$ generated. Further iterations of the algorithm show that this call to `q` has no answers. Concretely, the call would generate an infinite failed computation. When the algorithm terminates, the complete set of calls obtained is:

$$\{\langle \mathtt{s}(\mathtt{X},\mathtt{Y},\mathtt{Z}), \mathsf{true}\rangle, \langle \mathtt{p}(\mathtt{X},\mathtt{Y},\mathtt{Z}), \mathsf{true}\rangle, \langle \mathtt{q}(\mathtt{X},\mathtt{X},\mathtt{X}), \mathtt{X} \geq 0\rangle\}$$

The set of answers is $\{\langle \mathtt{p}(\mathtt{X},\mathtt{X},\mathtt{X}), \mathtt{X} \geq 0\rangle\}$. Thus we can see that there are some calls (namely, to `q` and `s`) that have no answers.

To generate the specialized program from this set of calls and answers, we generate resultants for the calls, and apply the answers to the bodies. Since there is no answer for `q(X,Y,Z)` in the resultant for `s(X,Y,Z)`, `s(X,Y,Z)` fails and the specialized program is empty. The specialized program thus consists only of the resultants `p(0,0,0)` and `p(X,X,X) <- {X = Y+1}, p(Y,Y,Y)`. The failure of the original goal is immediately apparent since there are no clauses for predicate `s`.

Example 29.9 More insight into the nature of the approximation can be gained by considering the same program as in the previous example, except that the body goals are reversed in the clause for `s`. In this case `q(X,Y,Z)` is called first, and the answers for `q` constrain the calls to `p`. The call $\langle \mathtt{q}(\mathtt{X},\mathtt{Y},\mathtt{Z}), \mathtt{true}\rangle$ results in the abstract answer c-atom $\langle \mathtt{q}(\mathtt{X},\mathtt{Y},\mathtt{Z}), \mathtt{X} \geq 0, \mathtt{Y} \geq 0, \mathtt{Z} = \mathtt{X} + \mathtt{Y}\rangle$. Again, widening is essential to derive this answer. Note that the solution `q(0,0,0)` is included as a result of the convex hull approximation, even though this is not a concrete solution.

This answer is then propagated to the call to `p`, hence there is a call c-atom $\langle \mathtt{p}(\mathtt{X},\mathtt{Y},\mathtt{Z}), \mathtt{X} \geq 0, \mathtt{Y} \geq 0, \mathtt{Z} = \mathtt{X} + \mathtt{Y}\rangle$. Specialization of this call to `p` gives the abstract answer $\langle \mathtt{p}(\mathtt{X},\mathtt{X},\mathtt{X}), \mathtt{X} \geq 0\rangle$.

The specialized program corresponding to this set of calls and answers is the following.

```
s(0,0,0) <-
  q(0,0,0), p(0,0,0).
p(0,0,0) <-
p(X,Y,Z) <-
  {X=X1+1, Y=Y1+1, Z=Z1+1},
  p(X1,X1,X1)
```

```
q(0,Z,Z) <-
  {Z > 0}
q(X,Y,Z) <-
  {X = X1+1, Z=Z1+1},
  q(X1,Y,Z1)
```

The instance of the clause for `s` is obtained by conjoining the answers for the body goals `q(X,Y,Z), p(X,Y,Z)`, that is, $\mathtt{X} \geq 0, \mathtt{X} = \mathtt{Y}, \mathtt{X} = \mathtt{Z}, \mathtt{Y} \geq 0, \mathtt{Z} = \mathtt{X} + \mathtt{Y}$, which simplifies to the constraint $\mathtt{X} = 0, \mathtt{Y} = 0, \mathtt{Z} = 0$. The above program does not make the failure of `s(X,Y,Z)` explicit; a non-trivial post-processing such as another run of the specialization algorithm would be needed to discover the failure of the call `q(0,0,0)`. The general point here is that the convex hull approximation loses the information that `q(0,0,0)` is not a solution for `q(X,Y,Z)`.

The two examples taken together show that the direction of propagation of answers affects precision. It would be possible to design an algorithm incorporating more sophisticated propagation, but post-processing or re-specialization is a practical alternative for experimental studies.

Note that the above presentation of the algorithm is naive in the sense that the sets of calls and answers need not be totally recomputed on each iteration. We use standard techniques to optimize the algorithm, focusing on the "new" calls and answers on each iteration. We can also use the recursive structure of the target program to optimize the iterative structure of the algorithm. Instead of one global fixpoint computation, we compute a series of fixpoints, one for each group of mutually recursive predicates.

29.4 Correctness of the Specialization

A program that has been specialized with respect to a c-atom $\mathcal{A} = \langle A, C\rangle$ produces the same answers as the original program for any terminating computation for any query in $\gamma(\mathcal{A})$. Note that the proposition below states nothing about the preservation of looping computations in the

original program. A goal that loops in the original program can finitely fail in the specialized program.

Proposition 29.10 Let P be a definite CLP program and $\mathcal{A}$ a c-atom. Let P' be the specialized program derived by the algorithm described above, with initial c-atom $\mathcal{A}$. Let S and T be the sets of call and answer c-atoms returned by the algorithm. Then for any goal $G = \leftarrow B_1, \ldots, B_k$ such that $i = 1 \cdots k$ and $B_i \in \gamma(\mathcal{A}')$ for some $\mathcal{A}' \in S$, $P \cup \{G\}$ has an answer ρ if and only if $P' \cup \{G\}$ has an answer ρ. Also, if $P \cup \{G\}$ fails finitely then $P' \cup \{G\}$ fails finitely.

Proof Suppose there is a terminating (possibly failed) derivation of $P' \cup \{G\}$. We argue by induction on the length of the derivation. If the derivation has length 0, then G fails immediately. We know that there is some $\mathcal{A}' = \langle A', C' \rangle$ in S such that $B_1 \in \gamma(\mathcal{A}')$, since the first call c-atom is $\langle B_1, \mathsf{true} \rangle$, and so S should contain an element $\mathcal{A}'$ such that $\langle B_1, \mathsf{true} \rangle \sqsubseteq \mathcal{A}'$. So a failure means that (i) there are no resultants for A', or (ii) that no resultant body has an answer, or (iii) that there is a resultant $A'\theta \leftarrow L$ with an answer φ for L given by the set of answer c-atoms, but B_1 does not unify with $A'\theta\varphi$. In the case of (i) there is a finitely failed computation tree of $P \cup \{G\}$. In the case of (ii) or (iii) there is either a finitely failed computation tree of $P \cup \{G\}$, or the computation tree for $P \cup \{G\}$ is infinitely failed.

If the derivation has length 1, with answer substitution ρ, then $G = \leftarrow B_1$ and there is some unit clause in P' whose head unifies with B_1 with substitution ρ. Now, unit clauses in P' may come from two sources: either they are already in P or they are the result of successfully unfolding the body of a non-unit clause, also in P. Hence by definition of the residual program construction the mgus are equivalent modulo variable renaming.

If all derivations of length at most m in P' have a corresponding derivation in P, then we show that all derivations of length $m + 1$ in P' do as well. Suppose the first clause used in the derivation is $A'\theta \leftarrow L$, $\mathsf{mgu}(B_1, A'\theta) = \varphi$ and $(L, B_2, \ldots, B_k)\varphi$ has a derivation in P' of length at most m. By the induction hypothesis there is a corresponding derivation for $(L, B_2, \ldots, B_k)\varphi$ in P. Then clearly there is a derivation in P corresponding to the $m + 1$ step derivation in P', obtained by concatenating the steps corresponding to the clause $A'\theta \leftarrow L$.

The above argument establishes soundness. For completeness, a sketch of a proof is provided. For each terminating derivation of $P \cup \{G\}$ we can construct a terminating derivation in $P' \cup \{G\}$. The clauses in P' that are needed to construct such a derivation exist by virtue of the closedness of the sets of calls and answers. That is $S = \omega(\mathsf{calls}(S, T), S)$ and $T = T \nabla \mathsf{answers}(S, T)$. Furthermore, the answers produced by successful derivations in P can be reproduced by derivations in P' by virtue of the correctness of the unfolding function aunf, and the procedure for computing the solution of a conjunction with respect to a set of answer c-atoms. □

```
sat(_,true) <-
sat(_,false) <- fail
sat(E,P) <- prop(E,P)
sat(E,and(F,G)) <-
        sat(E,F),
        sat(E,G)
sat(E,or(_F,G)) <-
        sat(E,G)
sat(E,or(F,_G)) <-
        sat(E,F)
sat(E,not(F)) <-
        not(sat(E,F))
sat(E,en(F)) <-
        trans(_Act,E,Ei),
        sat(Ei,F)
sat(E,an(F)) <-
        not(sat(E,en(not(F))))
sat(E,eu(F,G)) <-
        sat_eu(E,F,G)
```

```
sat(E,au(F,G)) <-
      sat(E,not(eu(not(G),
                   and(not(F),not(G))))),
      sat_noteg(E,not(G))
sat(E,ef(F)) <-
      sat(E,eu(true,F))
sat(E,af(F)) <-
      sat_noteg(E,not(F))
sat(E,eg(F)) <-
      not(sat_noteg(E,F))
sat(E,ag(F)) <-
      sat(E,not(ef(not(F))))
sat_eu(E,_F,G) <-
      sat(E,G)
sat_eu(E,F,G) <-
      sat(E,F),
      trans(_Act,E,Ei),
      sat_eu(Ei,F,G)
sat_noteg(E,F) <-
      sat(E,not(F))
sat_noteg(E,F) <-
      not((trans(_Act,E,Ei),
           not(sat_noteg(Ei,F))))
```

Figure 14: CTL metainterpreter

30 Examples

We implemented the algorithm described in the previous section, using the SICStus Prolog linear arithmetic constraint solver. Next we present some examples where on-line specialization as presented here is used for verifying some formulas in CTL [27].

Specialization can be seen as an approach to model-checking infinite systems [141, 52] and in this context our more powerful specialization techniques are highly relevant. We used the CTL metainterpreter shown in Fig. 14 (also used by M. Leuschel et al. [141]).

The set of transitions[25](predicate `trans/3` in the figure) of the system to be verified in the form of a (C)LP program is appended to this metainterpreter. Also, the property (predicate `prop/2` in the CTL metainterpreter) with respect to which verification is to be carried out should be specified. Finally, the specialization query provides the initial state and the CTL formula which is to be verified for the given system and initial state.

[25]A transition system may be that of a Kripke structure or a Petri Net, for instance.

Specialization Strategy Before applying the convex hull specialization, we performed a trivial top-down specialization with respect to the given goal. The main effect of this stage was to unfold the calls to the transition relation `trans/3`. In principle, this unfolding could be performed during the execution of the main specialization algorithm. However, the overall process is faster and easier to control when doing the specialization in two stages.

Example 30.1 Consider for instance the following transition system, where `trans(t,[X,Y],[Z,W])` holds iff state `[Z,W]` may be obtained from state `[X,Y]` using transition `t`.

```
trans(t1,[P1,P2],[X,P3]) <-
        X is 0,
        P1>=1,
        P2>=0,
        P3>=0,
        P3 is P2+1
```

```
trans(t2,[P1,P3],[P4,P2]) <-
        P1>=0,
        P2>=0,
        P4 is P1+2,
        P3 is P2+1
```

The encoding of an unsafe state property `[X,Y]` with `X>=3` is added as another clause in the CTL metainterpreter.

```
prop([X,Y],p(unsafe)) <- X>=3
```

The specialization query, taking into consideration the initial state `[X,Y]` with `X=1,Y=0`, for CTL formula `ef(p(unsafe))`[26] is `<- sat([1,0],ef(p(unsafe)))`. As a result of specializing the CTL metainterpreter with a description (transition system) of the system and a state property with respect to the query above, we obtained the empty program. This is equivalent to saying that there is no residual program in which state `[1,0]` may reach state `[X,Y]` with `X>=3`. Had we obtained a residual program we would have interpreted the residual program as the set of traces which lead from the initial state to the unsafe state, as above.

This behaviour may be regarded as that of a model checker, hence we argue that our specializer may be used as a model checker for some infinite state systems. The only requirement is that those systems may be expressed as definite (constraint) logic programs and the CTL formulas does not use negation.

Example 30.2 Figure 15 depicts a Petri net modeling one process with its critical section (`cs`) and a semaphore (`sema`) controlling access to it. The definition of predicate `trans/3` corresponding to the transition relation of the Petri net above, follows.

[26] Meaning that there exists a state in the future such that state property `unsafe` holds.

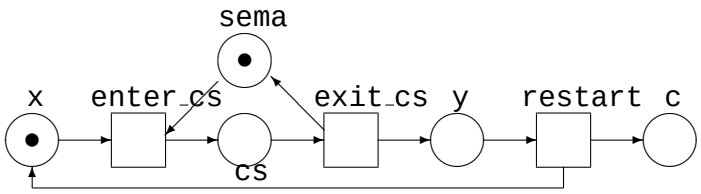


Figure 15: Petri Net with one semaphore

```
trans(enter_cs,[X,Sema,Cs,Y,C],[X1,Sema1,Cs1,Y,C]) <-
        X>=1, X1 is X-1,
        Sema>=1, Sema1 is Sema-1,
        Cs>=0, Cs1 is Cs+1
trans(exit_cs,[X,Sema,Cs,Y,C],[X,Sema1,Cs1,Y1,C]) <-
        Sema>=0, Sema1 is Sema+1,
        Cs>=1, Cs1 is Cs-1, Y>=0, Y1 is Y+1
trans(restart,[X,Sema,Cs,Y,C],[X1,Sema,Cs,Y1,C1]) <-
        X>=0, X1 is X+1,
        Y>=1, Y1 is Y-1,
        C>=0, C1 is C+1
```

Next, we may specify with the following clause the unsafe property of more than two processes being in their critical section (`cs`) at the same time:

```
prop([_X,_Sema,Cs,_Y,_C],p(unsafe)) <-  Cs>=2.
```

Now, for the specialization query, with constraint[27] `X>=1`:

```
<- sat([X,1,0,0,0],ef(p(unsafe)))
```

we obtained the empty program, thus denoting that there is no path from the initial state (`[X,1,0,0,0]`), with `X>=1` leading to a state where property `p(unsafe)` holds.

Example 30.3 Another way of specifying concurrent systems was proposed by U. A. Shankar [200]. Delzanno and Podelski [46], in turn, propose a systematic method to translate such specifications into CLP programs. Our translation is similar to theirs, differing only in the form of the clauses produced, mainly due to the meaning of the predicate employed.

Figure 16 below contains a specification of the bakery algorithm for two processes using the technique above cited.

Such a specification may be readily translated into the following definition of the `trans` predicate:

[27] For every token in the place with name `X` we associate a process, thus the constraint `X>=1`.

Control variables $p_1, p_2 : \{$*think, wait, use*$\}$

Data variables $turn_1, turn_2$: int

Initial condition $p_1 = think \wedge p_2 = think \wedge turn_1 = turn_2 = 0$

Events	**cond** $p_1 = think$	**action** $p_1' = wait \wedge turn_1' = turn_2 + 1$
	cond $p_1 = wait \wedge turn_1 < turn_2$	**action** $p_1' = use$
	cond $p_1 = wait \wedge turn_2 = 0$	**action** $p_1' = use$
	cond $p_1 = use$	**action** $p_1' = think \wedge turn_1' = 0$
	... symmetrically for Process 2	

Figure 16: The bakery algorithm

```
trans(f,[think,A,P2,B],[wait,A1,P2,B]) <-  A>=0, A1 is B+1
trans(f,[P1,A,think,B],[P1,A,wait,B1]) <-  B>=0, B1 is A+1
trans(s,[wait,A,P2,B],[use,A,P2,B]) <-  A>=0, A<B
trans(s,[P1,A,wait,B],[P1,A,use,B]) <-  B>=0, B<A
trans(s,[wait,A,P2,B],[use,A,P2,B]) <-  B=0
trans(s,[P1,A,wait,B],[P1,A,use,B]) <-  A=0
trans(t,[use,A,P2,B],[think,A1,P2,B]) <-  A>=0, A1=0
trans(t,[P1,A,use,B],[P1,A,think,B1]) <-  B>=0, B1=0
```

Consequently, an unsafe property for the previous system would be a state where the two processes are in their critical section (denoted as `use`) at the same time. This property is denoted as the clause:

```
prop([use,A,use,B],p(unsafe)) <-
```

Furthermore, verifying that there is no state of the above mentioned system where such an unsafe state holds amounts to obtaining an empty program for the following query:

```
<-sat([think,0,think,0],ef(p(unsafe)))
```

where the variables denoting the turn of each process, namely `A,B`, are initially constrained by `A=B=0`. As a result of the specialization we obtained the empty program thus verifying that there is no unsafe state in any path beginning from the initial state described in figure 16.

In a similar way we verified some correctness property [141] of the producers and consumers algorithm [5] for one producer, one consumer and a buffer of size one. The authors [141] could not successfully specialize this last example.

Assessment Here we have shown some applications of our specialization strategy to infinite state model checking. Compared to other approaches using specialization for the same purpose, we believe our approach sheds some insight into the field. The example of the bakery protocol was also verified by Fioravanti et al. [52]. As opposed to their approach we show the actual specialization strategy and its use in other related examples. We depart from a general CTL metainterpreter whereas Fioravanti et al. present a somehow specialized version of a CTL metainterpreter.

For the other examples of this section M. Leuschel et al. [141] have a four stage model checker, as opposed to ours which is just one specialization step. That is, M. Leuschel et al. first pass through an off-line specializer, then one or more specialization passes of their on-line specializer and finally one pass through a most-specific-version analyser.

Admittedly examples 30.2 and 30.3 in this section do not propagate answers, and require a simple unfolding prior to specialization with answers. By contrast, example 30.1 and the producer-consumer of [141] do not need any prior unfolding and have some limited answer propagation. That is, specialization with answers could be applied directly to the metainterpreter (together with the transition definition and the property), to yield the expected verification results. The running example of Section 29 does indeed need and use answer propagation.

31 Related Work

Despite the fact that unfold-fold approaches to program transformation and program specialization based on a fixpoint calculation are not directly comparable, there are some unfold-fold methods related to our techniques.

In [153] the authors propose the use of convex-hull analysis to enable optimization/specialization of CLP programs. Their removal, refinement and reordering may be rendered as transformation rules. The fairness of comparing our technique with theirs is dubious because theirs is used for compilation and ours for specialization, and potentially the former is a special case of the latter one. A weak form of their method was later dubbed by Fioravanti et al. [51] as contextual specialization.

Peralta and Gallagher [170] use arithmetic constraints (convex hulls) to specialize CLP programs, especially an interpreter for imperative programs.

Their specialization abstract domain is the same as that one used here, but the specializer only propagates information top-down and cannot achieve the effects of answer propagation.

Fioravanti et al. [51] (without reference to [92, 170]) argue an automatic specialization method based on folding and unfolding among other transformation techniques. They use a domain of atomic formulas constrained by arithmetic expressions with upper bound based on widening alone, rather than the combination of convex hull and widening which is known to give better approximations. The aspects of their method concerned with specialization resemble a top-down on-line specializer with a subsequent "contextual specialization", and thus does not in general achieve the effects of answer propagation.

Another application of specialization using abstract interpretation over polyhedral descriptions followed by a contextual specialization was given by Howe *et al.* [92]. This approach is similar in being based on abstract interpretation over a domain of polyhedra. Its bottom-up analysis of answers is not as powerful as ours, which combines top-down and bottom-up propagation.

Conjunctive Partial Deduction (CPD) [201] aims to solve the answer propagation problem in a different way. The approach is to preserve shared information between subgoals by specializing conjunctions rather than atoms. It is not yet clear whether CPD or answer propagation via atoms, or some combination of both, will be most effective. In the extreme case of CPD, no resolvent is ever split, and no answer propagation is needed. However in general resolvents can be of unbounded size, some splitting is therefore needed, and answer propagation is required to preserve shared information between conjunctions.

32 Final Remarks

We have presented a new widening operator on regular types within an abstract interpretation-based characterization of type inference. The idea behind it is similar to set-based analyses [53, 22] in that we assign and fix type names, but it is applied here with more generality. The most comparable aproach among the set-based analyses would be [71]. It can be seen as a generalization of the idea of "guessing" the growth of the types during analysis which is behind [212]. Instead of guessing, our technique determines exactly where the type is growing. The resulting widening operator has been presented on deterministic regular types. However, its extension to non-deterministic regular types should be straightforward.

Our operator is more precise than previous approaches, but it is still efficient. This has been shown with (preliminary) practical results. However, it does not guarantee termination. We are currently working on the non-termination problem. A moded type domain will help in this. The

idea is to enhance abstract unification so that it is able to identify the "transference" of type names from the input to the output types, so that the names are not dropped. This will remedy the problem of Example 26.7 and, hopefully, allow us to prove termination of analyses with the proposed widening operator.

We have also presented an algorithm for specialization of definite[28] (C)LP programs. Its main novelty is the propagation of calls and answers described by atoms whose arguments are described by convex hulls. The use of answer propagation with an expressive domain like convex hulls gives increased specialization. By interpreting Prolog arithmetic as constraints we can also apply the algorithm to "non-constraint" programs.

Future Work This work has revealed two issues that may be worth investigating for practical purposes: the impact on the efficiency of analysis of the different implementation techniques for different analysis methods, and of the simplification of types.

Because negation in CTL is interpreted as negation in (constraint) logic programs, this restricts us to model checking of safety properties, as opposed to liveness properties. Extending the presented techniques to include negation is the focus of our current research.

Scalability of our specialization method is one avenue into which we plan to extend the current proposal, thus making our specialization techniques applicable to larger systems.

Also, in order to improve precision of our specialization with answers, more sophisticated domains are sought.

[28] At the moment we can only specialize definite (constraint) logic programs.

Part IV
Inductive Theorem Proving by Program Specialisation: Generating proofs for ISABELLE using ECCE

In this part we discuss the similarities between program specialisation and inductive theorem proving, and then show how program specialisation can be used to perform inductive theorem proving. We then study this relationship in more detail for the particular problem of verifying infinite state systems in order to establish a clear link between program specialisation and inductive theorem proving. Indeed, ECCE is a program specialisation system which can be used to automatically generate abstractions for the model checking of infinite state systems. We show that to verify the abstractions generated by ECCE we may employ the proof assistant ISABELLE. Thereby ECCE is used to generate the specification, hypotheses and proof script in ISABELLE's theory format. Then, in many cases, ISABELLE can automatically execute these proof scripts and thereby verify the soundness of ECCE's abstraction. In this work we focus on the specification and verification of Petri nets.

33 Background

The relation between program specialisation and theorem proving has already been raised several times in the literature [209, 78, 210, 174]. In this paper we will examine in closer detail at the relationship between partial deduction and inductive theorem proving.

Partial Deduction The heart of any technique for *partial deduction* is a program analysis phase. Given a program P and an (atomic) goal $\leftarrow A$, one aims to analyse the computation-flow of P for all instances $\leftarrow A\theta$ of $\leftarrow A$. Based on the results of this analysis, new program clauses are synthesised.

In partial deduction, such an analysis is based on the construction of finite and usually incomplete[29], SLD(NF)-trees. More specifically, following the foundations for partial deduction developed in [152] (see also [127] for an up-to-date overview), one constructs

- a finite set of atoms $S = \{A_1, \ldots, A_n\}$, and
- a finite (possibly incomplete) SLD(NF)-tree τ_i for each $(P \cup \{\leftarrow A_i\})$,

[29] As usual in partial deduction, we assume that the notion of an SLD-tree is generalised [152] to allow it to be incomplete: at any point we may decide not to select any atom and terminate a derivation.

such that:

1) the atom A in the initial goal $\leftarrow A$ is an instance of some A_i in S, and

2) for each goal $\leftarrow B_1, \ldots, B_k$ labelling a leaf of some SLD(NF)-tree τ_l, each B_i is an instance of some A_j in S.

The conditions 1) and 2) ensure that *together* the SLD(NF)-trees $\tau_1, \ldots, \tau_n$ form a *complete description* of all possible computations that can occur for all concrete instances $\leftarrow A\theta$ of the goal of interest. At the same time, the point is to propagate the available input data in $\leftarrow A$ as much as possible through these trees, in order to obtain sufficient accuracy. The outcome of the analysis is precisely the set of SLD(NF)-trees $\{\tau_1, \ldots, \tau_n\}$: a complete, and hopefully as precise as possible, description of the computation-flow. Finally, a code generation phase produces a *resultant clause* for each non-failing branch of each tree, which synthesises the computation in that branch. The approach has been generalised to specialising a set of *conjunctions* rather than just atoms in [42]. An overview of control techniques that are used in partial deduction, such as determinacy, homeomorphic embedding, and characteristic trees, can be found in [127].

Let us illustrate conjunctive partial deduction on the following simple program.

```
even(0).
even(s(X)) :- odd(X).                odd(s(X)) :- even(X).
```

Conjunctive partial deduction can specialise this program for the query $\leftarrow even(X) \wedge odd(X)$ by constructing the incomplete SLD-tree for it depicted in Fig. 17. The set S mentioned above would simply be $S = \{even(X) \wedge odd(X)\}$. The specialised program we obtain, is:

```
even_odd(s(X)) :- even_odd(X).
```

It is immediately obvious that $even\_odd(X)$ will never succeed, and hence that no number is even and odd at the same time. The partial evaluator ECCE [140, 42] will basically produce the same result (slightly more involved as it does not re-order atoms by default) and can also automatically infer the failure of $even\_odd(X)$ by applying its bottom up more specific program construction phase [155] in the post-processing.

Inductive Theorem Proving Now, the above result corresponds to an inductive proof showing that no number can be both even and odd. The left branch of Fig. 17 corresponds to examining the base case $X = 0$, while the right branch corresponds to the induction step whereby $even(s(Y)), odd(s(Y))$ is rewritten into the equivalent $odd(Y), even(Y)$ so that the induction hypothesis can be applied.

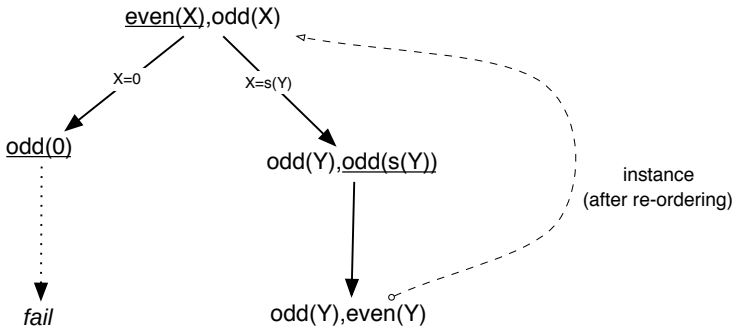


Figure 17: Specialisation of even-odd

In a sense the conjunctive partial deduction has identified a working induction schema and the bottom-up propagation [155] has performed the induction proper. This highlights a similarity between partial deduction and *inductive theorem proving*. Indeed, in the induction step of an inductive proof one tries to transform the induction assumption(s) for $n+1$ using basic inference rules so as to be able to apply the induction hypothese(s) and complete the proof. In partial deduction, one tries to transform the atoms in A (or conjunctions for conjunctive partial deduction) by unfolding so as to be able to "fold" back all leaves. The set of atoms A thus plays the role of the induction hypotheses and resolution the role of classical theorem proving. In summary,

- there is a striking similarity between the control problems of partial deduction and inductive theorem proving. The problem of ensuring A-closedness is basically the same as finding induction hypotheses where the induction "goes through'.' Many control techniques have been developed in either camp (e.g., [18] for inductive theorem proving) and cross-fertilisation might be possible.

- if basic resolution steps correspond to logical inference rules one may be able to perform inductive theorem proving directly by partial deduction. For example, ECCE can fully automatically prove associativity of addition [121] (see also[143])..

  The only difference is that resolution is not guaranteed to decrease the induction parameter, so this is only guaranteed to work if the predicates to be specialised are inductively defined.

In the next sections we show how ECCE can be used to perform inductive theorem proving as applied to verification tasks and how the induction schemas produced by ECCE can be automatically translated for the proof assistant Isabelle [168].

34 Infinite Model Checking by Program Specialisation

In recent work it has been shown that logic programming based methods in general, and partial deduction in particular, can be applied to model checking [26] of infinite state systems. As this problem can also be tackled by inductive theorem proving [168] we choose this as the basis of a more formal comparison. Indeed, one of the key issues of model checking of infinite systems is *abstraction* [28]. Abstraction allows to approximate an infinite system by a finite one, and if proper care is taken the results obtained for the finite abstraction will be valid for the infinite system. This is related to finding proper induction schemas for inductive theorem proving, which in turn is related to the control problem of partial deduction.

In earlier work we have tried to solve the abstraction problem by applying existing techniques for the *automatic* control of *(logic) program specialisation*, [126] and modelling the system to be verified as a logic program by means of an interpreter [80, 142]. Thereby, the interpreter describes how the states of the system change by executing transitions. By applying partial deduction to the interpreter we expect a finite abstraction of the possibly infinite state space of the system to be generated. This abstraction may then be used to verify system properties of interest. This approach proved to be quite powerful as it was possible to obtain decision procedures for the coverability problem, if "typical" specialisation algorithms, as for example implemented in the ECCE system [140, 119], are applied to logic programs that encode Petri nets [137]. It is even possible to precisely mimic well known Petri net algorithms (by Karp–Miller [108] and by Finkel [48]) when the program specialisation techniques are slightly weakened. The results of [137] refer to *forward* algorithms only, i.e. algorithms which construct, beginning from some initial state, an abstract representation of the whole reachability tree of a Petri net. However, for some classes of systems such exhaustive algorithms are not necessary or even not precise enough to decide coverability [1, 49, 50]. In such cases partial deduction may often be successfully applied as well [136], thereby mimicking well known *backward algorithms* [49].

Technically, the dynamic system specified in the input for the partial deduction algorithm can also be viewed as an inductive system describing the set of finite behaviours, i.e. the set of finite *paths*. Thereby, the set of initial states form the inductive base and each transition represents an inductive step. For the output of the partial deduction algorithm to be a sound abstraction each of the states reachable by a path must be contained in a state representation of the output. It is desirable to verify this property if we cannot ensure that the partial deduction algorithm is correctly implemented. The goal of this work is to show that such proofs can be generated and executed automatically. To this end we employ the partial deduction system ECCE for the automatic generation of the theory and the proof scripts. The proof assistant ISABELLE [169] is used to execute the proof scripts.

If we can use ISABELLE to verify the soundness of the output of the partial deduction method we may also ask whether it is possible to generate the hypotheses automatically and thereby use ISABELLE directly as a model checker of infinite systems. To this end, similar to the partial deduction system, ISABELLE needs to perform some kind of abstraction while searching for a proof of some dynamic property such as safety.

In this paper we focus on the specification and verification of Petri nets. This is due to their simple representation as a logic program as well as in a ISABELLE theory. The following section describes how we can specify Petri nets in ISABELLE. Then we discuss how such specifications are generated using ECCE, and how ECCE output can be translated into ISABELLE. In Section 37 we demonstrate how proof scripts can be used in ISABELLE for automatic theorem proving. In Section 38 we demonstrate the complete verification process using an example specification. The above mentioned automatic generation of hypotheses and some efficiency issues are discussed in Section 39. The last section gives a conclusion and proposes some further work. All relevant source code of the ECCE system can be found in the technical report [116].

35 Specification of Petri nets in ISABELLE

The proof assistant ISABELLE [168] has been developed as a generic system for implementing logical formalisms. Instead of developing an all new logic for our purposes we will use the specification and verification methods realised by the implementation of Higher Order Logic (HOL) in ISABELLE. HOL allows to express most mathematical concepts and, in contrast to, for example, First Order Logic, it allows the specification of and the reasoning about inductively defined sets. This latter feature is crucial for our purposes. Hence, strictly speaking, we will develop specifications in ISABELLE/HOL. Furthermore, the current ISABELLE system provides the language ISAR for the specification of theories and the development of proof scripts. In this work we will use ISAR instead of ISABELLE's implementation language ML since ISAR is much easier to use as it hides most implementation details of ISABELLE. However, the possibilities to develop proof tactics using ISAR only are very limited. Consequently we conjecture that for efficient automatic theorem proving the use of ISAR allone is insufficient (see also Section 40).

ISABELLE allows specifications as part of *theories*. A theory can be thought of as a collection of *declarations*, *definitions*, and *proofs*. ISABELLE/HOL is a typed logical language where the *base types* resemble those of functional programming languages such as ML. To specify new types ISABELLE provides *type constructors*, *function types*, and *type variables*. We will introdce the particular concepts as we will use them and refer for additional information to the

Isabelle/Isar Reference Manual[30].

Terms are formed by applying functions to arguments, e.g. if f is a function of type $\tau_1 \Rightarrow \tau_2$ and t a term of type τ_1 then ft is a term of type τ_2.

Formulas are terms of base type `bool`. Accordingly, the usual logical operators are defined as functions whose arguments and domain are of type `bool`.

We specify the Petri net theory `PN` as a successor of the theory `Main` which is provided by ISABELLE/HOL. `Main` contains a number of basic declarations, definitions, and lemmas concerning often required basic concepts such as lists and sets. Thereby, every part of the theory `Main` becomes automatically visible in `PN`:

```
theory PN = Main:
```

To simplify the specification and to increase readability of the theory we define the type `state` which corresponds to a notion in Petri net theory: A *state* or *marking* is a vector of natural numbers representing the number of *tokens* on the *places* of a Petri net. The number of dimensions of the vector corresponds to the number of places of the particular net. In ISABELLE we use the type constructor $\times$ to define the type `state` as a product over the base type `nat`:

```
types
 state = "nat × nat ×...× nat"
```

Based on the type `state` we declare the functions `paths`, `trans`, and `start`. The function `start` represents the *initial state* of the Petri net. Note that since we allow parameters in the definition of `state` it actually may represent a set of initial states. The function `trans` describes how the firing of a *transition* can change the state of a Petri net. The additional parameter of type `nat` is used to refer to a particular transition of the net. The set of finite possible sequences of transitons starting in the initial state is represented by `paths`. Note that the declaration of `trans` and *paths* is independent of the particular considered Petri net.

```
consts
 start ::  "nat ⇒ ...⇒ nat ⇒ state"
 trans ::  "(state × state × nat) set"
 paths ::  "(state list) set"
```

By assigning a unique number the transition names are defined as a of enumeration type. Consequently, for each transition t we include a declaration of the following form:

```
consts
 t ::  "nat"
```

[30]Lawrence C. Paulson. The Isabelle Reference Manual. http://isabelle. in.tum.de/doc/ref.pdf.

The initial state `start` is defined by a term *term* of type state:

```
defs
 start_def [simp]:  "start list of variables ≡ term"
```

The optional `[simp]` controls the strategy of ISABELLE's built-in simplifier to apply this definition whenever possible. For our purposes *term* will be always a tuple of terms built using the unary successor function `Suc`, 0, and variables appearing in the *list of variables* (the number of variables in this list must correspond to the number of parameters in the declaration of `start`.

The transition function is defined as a set of transitions of the Petri net. Thereby each transition is represented as a tuple `(x,y,n)`, where x and y are tuples of terms built by `Suc` and variables of the corresponding *list of variables*. The term n is the name of the transition.

```
defs
 trans_def:  "trans ≡ {(x,y,n).
                    (∃ list_1 of variables.  (x,y,n)= transition_1
                    ∨ (∃ list_2 of variables.  (x,y,n)= transition_2
                    ⋮
                    ∨ (∃ list_n of variables.  (x,y,n)= transition_n}"
```

One of the important features of ISABELLE/HOL is the possibility of inductive definitions. We define `paths` inductively using the following two rules:

```
inductive paths
intros
 zero:  "[(start list of variables)] ∈ paths"
 step:  "⟦(y,z,n) ∈ trans; y#l ∈ paths⟧ ⟹ z#(y#l) ∈ paths"
```

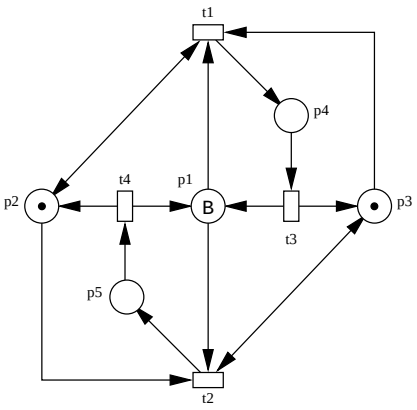
The first rule defines all initial states to be paths. The second rule allows the construction of new paths by extending an arbitrary path by a new state if there exists a transition from the state at the head of the path to the new state.

Finally, each transition t is defined as follows, where n is a unique natural number:

```
defs
 t_def [simp]:  "t ≡ n"
```

The following example shows the the specification of a Petri net according to this scheme.

Example 35.1 We encode the Petri net depicted below in ISABELLE/HOL. The initial state is defined by one token on each of the places $p2$ and $p3$, and the parameter `A` representing an arbitrary number of tokens on place $p1$ ($p1$, $p2$, $p3$ correspond to the first, second, and third dimension, respectively, of the state vector.




```
theory PN = Main:
types
 state = "nat × nat × nat × nat × nat"
consts
 start ::  "nat ⇒ state"
 trans ::  "(state × state × nat) set"
 paths ::  "(state list) set"
 t1 ::  "nat"
 t2 ::  "nat"
 t3 ::  "nat"
 t4 ::  "nat"
defs
 start_def [simp]:  "start ≡ (B,(Suc 0),(Suc 0),0,0)"
 trans_def:  "trans ≡ {(x,y,n).
                  (∃ E D C B A. (x,y,n)=(((Suc A),(Suc B),(Suc C),D,E),
                                        (A,(Suc B),C,(Suc D),E),t1))
                  ∨ (∃ E D C B A. (x,y,n)=(((Suc A),(Suc B),(Suc C),D,E),
                                          (A,B,(Suc C),D,(Suc E)),t2))
                  ∨ (∃ E D C B A. (x,y,n)=((A,B,C,(Suc D),E),
                                          ((Suc A),B,(Suc C),D,E),t3))
                  ∨ (∃ E D C B A. (x,y,n)=((A,B,C,D,(Suc E)),
                                          ((Suc A),(Suc B),C,D,E),t4))}"
 t1_def [simp]:  "t1 ≡ 0"
 t2_def [simp]:  "t2 ≡ 1"
 t3_def [simp]:  "t3 ≡ 2"
 t4_def [simp]:  "t4 ≡ 3"
inductive paths
intros
 zero:  "[(start B)] ∈ paths"
 step:  "⟦(y,z,n) ∈ trans; y#l ∈ paths⟧ ⟹ z#(y#l) ∈ paths"
```

□

36 Generating ISABELLE theories using ECCE

Since we aim to verify the partial deduction results of ECCE, we have integrated the generation of the ISABELLE theory directly into ECCE. The generated ISABELLE theory consists of three parts:

1. the specification of the Petri net,
2. the specification of the coverability graph as generated by ECCE,
3. the lemma to be verified together with a proof script.

In this section we deal with the first two parts while the third part is discussed in Section 37.

36.1 Generating Petri net specifications from logic programs

The ISABELLE theory generator integrated in ECCE assumes that the transitions of a Petri net are specified by a set of clauses of a ternary predicate. The first parameter represents a transition name, the second represents the set of states where the transition can be applied, and the third how the state changes if the transition is executed. Technically, the second and the third parameter of each clause are lists of the length corresponding to the number of places. Relying on unification, conditions and changes can be easily expressed. For example, the condition that at least two tokens are on place $p3$ in a Petri net with five places is expressed by the term `[X0,X1,s(s(X2)),X3,X4]` (thereby `s` can be interpreted as the successor function on natural numbers). Similarly, the state change can be expressed: the removal of one token on place $p3$ and generation of two tokens on $p1$ is represented as `[s(s(X0)),X1,s(X2),X3,X4]`.

The initial state is simply represented as a single clause where the last parameter must be a list of the length corresponding to the number of places. Each element of the list can be constructed using `0`, the unary function `s`, and variables.

Example 36.1 The following logic program encodes the Petri net of Example 1.

```
trans(t1,[s(X0),s(X1),s(X2),X3,X4],[X0,s(X1),X2,s(X3),X4]).
trans(t2,[s(X0),s(X1),s(X2),X3,X4],[X0,X1,s(X2),X3,s(X4)]).
trans(t3,[X0,X1,X2,s(X3),X4],[s(X0),X1,s(X2),X3,X4]).
trans(t4,[X0,X1,X2,X3,s(X4)],[s(X0),s(X1),X2,X3,X4]).

start([B,s(0),s(0),0,0]).
```

□

The implementation of the theory generator is part of the file "code_generator.pro" and can be found in [116]. The generation is initiated by a call to the clause `print_specialised_program_isa`. In a user dialog the name of the file containing the Petri net specification, and the names of the predicates representing transitions and initial state, respectively are determined. The ISABELLE specification is generated by the subsequent calls of `print_isa_header`, `print_isa_type_decl`, `print_isa_path_decl(Data)`, and `print_isa_path_def(Data)` in the body of `print_specialised_program_isa`. For example, the ISABELLE theory of Example 1 has been generated from the logic program of Example 2.

36.2 Generating specifications of the coverability graph from logic programs

To use partial deduction techniques for model checking we need to specify also the verification task as a logic program. To this end we may implement the satisfiability relation of some temporal logic as a logic program. However, the generation of a coverability graph (by partial deduction or other techniques) is not effective for all tasks that can be expressed with a powerful temporal logic. However, one of the tasks where it is effective is the checking of *safety properties*. To express safety properties we only require the definition of the *EU* operator of the temporal logic *CTL*:

```
infinite_model_check(basic_safety,Formula) :- start(_,S),
     Formula = sat(S,eu(true,p(unsafe))).

sat(E,p(P)) :- prop(E,P).
sat(E,eu(F,G)) :-  sat_eu(E,F,G).
sat_eu(E,_F,G) :-  sat(E,G).
sat_eu(E,F,G) :-  sat(E,F), trans(_Act,E,E2), sat_eu(E2,F,G).
```

Depending on the safety property we are interested in we define when a state is considered to be unsafe. For example the clause `prop([X0,X1,X2,s(X3),s(X4)],unsafe)` defines a state of a Petri net to be unsafe when there exist at least one token on each of the places $p4$ and $p5$.

Note that simply calling the clause `infinite_model_check(basic_safety,Formula)` in Prolog would force the system to explore an infinite derivation. Due to the potentially infinite state space of a Petri net also methods like tabeling would be in general insufficient to deal with this problem.

Before we apply the partial deduction system ECCE we will first perform a preliminary compilation of the particular Petri net and task. Thereby we will get rid of some of the interpretation

overhead and achieve a more straightforward equivalence between markings of the Petri net and atoms encountered during the partial deduction phase. We will use the LOGEN offline partial deduction system [134] to that effect (but any other scheme which has a similar effect can be used). This system allows one to annotate calls in the original program as either reducible (executed by LOGEN) or non-reducible (not executed and thus kept in the specialised program).[31] In our case we will annotate all calls to `trans` and `start` as reducible. After that, the LOGEN system will (efficiently) produce a compiled version: As can be seen in Example 3, the compilation gives us a predicate `sat_eu__2` with one argument each for the transition name and the result, plus one argument per Petri net place. Observe that LOGEN (and ECCE as well) adds two underscores and a unique identifier to existing predicate names. `sat_eu__2` contains one clause per transition of the Petri net plus one fact (for the marking reached). The initial marking is encoded in the one clause for `ssat__0` which calls `sat__1`.

Example 36.2 Applying LOGEN to the Petri net specification of Example 2 and the above task implementation generates the following clauses:

```
sat_eu__2(B,C,D,s(E),s(F)).
sat_eu__2(s(G),s(H),s(I),J,K) :- sat_eu__2(G,s(H),I,s(J),K).
sat_eu__2(s(L),s(M),s(N),O,P) :- sat_eu__2(L,M,s(N),O,s(P)).
sat_eu__2(Q,R,S,s(T),U) :- sat_eu__2(s(Q),R,s(S),T,U).
sat_eu__2(V,W,X,Y,s(Z)) :- sat_eu__2(s(V),s(W),X,Y,Z).
sat__1(B,C,D,E,F) :- sat_eu__2(B,C,D,E,F).
ssat__0 :- sat__1(B,s(0),s(0),0,0).
```

□

After this precompilation we can apply ECCE to the resulting program. To this end we aim to specialise the predicate `ssat__0`. The result of applying ECCE to the program of Example 3 is given in Example 4:

Example 36.3

```
ssat__0 :-  ssat__0__1.
   /* ssat__0__1 --> [ssat__0] */
ssat__0__1 :- sat__1__2(A).
   /* sat__1__2(A) --> [sat__1(A,s(0),s(0),0,0)] */
sat__1__2(A) :- sat_eu__2__3(A).
   /* sat_eu__2__3(A) --> [sat_eu__2(A,s(0),s(0),0,0)] */
sat_eu__2__3(s(A)) :- sat_eu__2__4(A).
```

[31] LOGEN is offline: the control decisions have been taken beforehand (and are encoded in the annotations).

```
sat_eu__2__3(s(A)) :-sat_eu__2__5(A).
   /* sat_eu__2__4(A) --> [sat_eu__2(A,s(0),0,s(0),0)] */
sat_eu__2__4(A) :- sat_eu__2__3(s(A)).
   /* sat_eu__2__5(A) --> [sat_eu__2(A,0,s(0),0,s(0))] */
sat_eu__2__5(A) :- sat_eu__2__3(s(A)).
```

□

From the output of ECCE we generate an ISABELLE theory representing the generated coverability relation. Independent of the particular domain this relation is declared as a set of pairs of states:

```
consts
 coverrel::  "(state × state) set"
```

For each predicate name of a clause in the specialised program, which represents a set of states we add a declaration of the form:

```
consts
 name ::  nat ⇒ ...⇒ nat ⇒ state"
```

Thereby the number of parameters of type `nat` corresponds to the number of variables in the head of the clause. The definitions have the form:

```
defs
 name_def:   "name  list of variables  ≡  term"
```

For our purposes *term* will be always a tuple of terms built using the unary successor function Suc, 0, and variables appearing in the *list of variables* (the number of variables in this list must correspond to the number of parameters in the declaration of *name*).

Finally, the coverability relation is defined as a set of pairs of states. In the specialised program every clause of the form $name_m(args_m)$ `:-` $name_n(args_n)$ corresponds to such a pair. Formally, in the ISABELLE theory each pair is represented as a tuple `(x,y)`, where x and y are tuples of terms built by Suc and variables of the corresponding *list of variables*:

```
defs
 coverrel_def:  "coverrel ≡
                        {(x,y).   ∃ list1 of variables.   x= state11  ∧  y= state12}
                      ∪ {(x,y).   ∃ list2 of variables.   x= state21  ∧  y= state22}
                      ⋮
                      ∪ {(x,y).   ∃ listm of variables.   x= statem1  ∧  y= statem2}"
```

The theory generator (cf. [116]) produces automatically the specification of the coverability relation from the specialised program. To this end the predicate names characterising the coverability relation in the specialised program are determined by a user dialog (only the unspecialised names have to be provided, e.g. in the above example `sat__1` and `sat_eu__2`). In the body of `print_specialised_program_isa` the calls to `print_isa_cover_decl` and `print_isa_cover_def` generate the necessary declarations and definitions, respectively.

Example 36.4 The following theory was generated by the theory generator [116] from the program of Example 4:

```
consts
 coverrel::  "(state × state) set"
 sat__1__2 ::  "nat ⇒ state"
 sat_eu__2__3 ::  "nat ⇒ state"
 sat_eu__2__4 ::  "nat ⇒ state"
 sat_eu__2__5 ::  "nat ⇒ state"
defs
 sat__1__2_def:  "sat__1__2 A ≡ (A,(Suc 0),(Suc 0),0,0)"
 sat_eu__2__3_def:  "sat_eu__2__3 A ≡ (A,(Suc 0),(Suc 0),0,0)"
 sat_eu__2__4_def:  "sat_eu__2__4 A ≡ (A,(Suc 0),0,(Suc 0),0)"
 sat_eu__2__5_def:  "sat_eu__2__5 A ≡ (A,0,(Suc 0),0,(Suc 0))"
 coverrel_def:  "coverrel ≡ {(x,y).  ∃ A. x=(sat__1__2 A)
                                     ∧ y=(sat_eu__2__3 A)}
                          ∪ {(x,y).  ∃ A. x=(sat_eu__2__3 (Suc A))
                                     ∧ y=(sat_eu__2__4 A)}
                          ∪ {(x,y).  ∃ A. x=(sat_eu__2__3 (Suc A))
                                     ∧ y=(sat_eu__2__5 A)}
                          ∪ {(x,y).  ∃ A. x=(sat_eu__2__4 A)
                                     ∧ y=(sat_eu__2__3 (Suc A))}
                          ∪ {(x,y).  ∃ A. x=(sat_eu__2__5 A)
                                     ∧ y=(sat_eu__2__3 (Suc A))}"
```

□

37 Proof Scripts

In this section we demonstrate how we can prove theorems using ISABELLE/ISARand how we can write proof scripts for automatic execution. Thereby we focus only on some of the "execution style" proof commands of ISABELLE/Isar. These commands can be considered to be the classical way of writing ISABELLE proofs although the actual ISABELLE proof methods are wrapped

within the ISAR language. Note however that ISAR allows also a more "mathematical style" notation of proofs than the one we use here (see the *Isabelle/Isar Reference Manual* for details).

Furthermore we discuss only the proof methods we are going to apply in order to solve the verification task of ECCE. Keep in mind that ISABELLE/ISAR provides a much wider range of methods.

The proof mode of ISABELLE/ISAR is initiated by executing a `lemma`. When entering the proof mode ISABELLE/ISAR generates a single pending subgoal consisting of the lemma to be proven. The list of subgoals can be altered, mainly by executing *proof methods*. Proof methods are executed using the proof command `apply`. Thereby the list of subgoals defines the *proof state*. The proof mode can be left by executing `done` in the case that there are no pending subgoals (the proof state is the empty list of subgoals, in which case ISABELLE/ISAR prints `No subgoals!`).

Note that all proof methods described below only transform the first subgoal of the proof state. For finding a proof this may be inconvenient. Therefore, ISABELLE/ISAR provides commands to change the order of the subgoals. However, our aim in this paper is the automatic execution of proof scripts, not their interactive development.

37.1 Rewriting

To rewrite a subgoal using existing definitions and lemmas automatically we may execute ISABELLE's simplifier: `apply(simp)`. For the simplifier to automatically attempt to use new defintions and lemmas they have to be accompanied by the option `[simp]`. Such defined simplification rules are then applied from left to right. However, we have to take care if we define simplification rules in such a way as they may slow the simplifier down considerably or even cause it to loop. Instead of defining a general simplification rule we may also use the simplifier to only apply certain, explicitely stated definitions. E.g., the execution `apply(simp only:` *r*`_def)` causes to rewrite using the definition of `r` only.

37.2 Introduction and Elimination

Based on reasoning using *natural deduction* there are two types of rules for each logical symbol, such as $\vee$: *introduction rules* which allow us to infer formulas containing the symbol (e.g. $\vee$), and *elimination rules* which allow us to deduce consequences of a formula containing the symbol (e.g. $\vee$).

In ISABELLE an introduction rule is usually applied by `apply(rule` *r*`)`. Assume *r* being a rule of the form:

$$\frac{P_1, \ldots, P_n}{Q}$$

where Q is a formula containing the introduced logical symbol while the formulas P_1,...,P_n in the premise do not. Then, if *r* is applied as introduction rule the current first subgoal is unified with Q and replaced by the properly instantiated P_1,...,P_n.

An elimination rule is usually applied using `apply(erule r)`. Assume *r* being a rule of the above form and the current first subgoal of the form $A_1, \ldots, A_m \Longrightarrow S$. Then, if *r* is applied as elimination rule S is unified with Q and some A_i is unified with P_1. The old subgoal is replace by $n-1$ new subgoals of the form $A_1, \ldots, A_{i-1}, A_{i+1}, \ldots, A_m \Longrightarrow P_k$ with $2 \leq k \leq n$.

In our verification proofs we will use explicitely only the elimination rules `disjE` for disjunction and `paths.induct` for induction over the length of paths.

37.3 Automatic Reasoners

Most classical reasoning of even simple lemmas can require the application of a vast amount of rules. To simplify this task ISABELLE provides a number of automatic reasoners. Here we will make use of `blast` which is the most powerful of ISABELLE's reasoners. Additionally, we will employ `clarify` which performs obvious transformations which do not require to split the subgoal or render it unprovable. The method `clarify` and the explicit application of the elimination rule `disjE` (see above))) was necessary to tune the proof process. This tuning was necessary to complete the verification proofs of even very small Petri nets using the available computing resources.

Additionally to the two classical reasoners we also employ the simplifier `simp` as an automatic proof tool as it can also handle some arithmetics. Furthermore, for some cases in our verification task `simp` succeeded faster than `blast` if it was able to eliminate a subgoal at all.

37.4 Scripts

To improve the handling of large proofs and to allow a higher flexibility of a proof proof scripts can be extended by the following operators:

- *method*$_1$`,` `...` `,` *method*$_n$: a list of methods represents their sequential execution;
- `(`*method*`)`: mainly used to define the scope of another operator;
- *method*`?`: executes *method* only if it does not fail,

- *method*$_1$ | . . . | *method*$_n$: attempts to execute *method*$_k$ only if each *method*$_i$ with $i < k$ failed;

- *method*+: *method* is repeatedly executed until it fails.

For our verification task the lemma and proof script are generated automatically by the theory generator [116] (by calls to `print_isa_lemma` and `print_isa_proofscript` in the body of `print_specialised_program_isa`). The execution of the script in the example below is illustrated in the next section.

Example 37.1 The following lemma and script corresponds to the one automatically generated by ECCE for the Petri net specification of Example 1:

```
lemma "l ∈ paths ⟹ ∃ y.  ((hd l),y) ∈ coverrel"

 apply(erule paths.induct)
  apply(simp only: start_def
                   coverrel_def)
  apply(simp only: sat__1__2_def
                   sat_eu__2__3_def
                   sat_eu__2__4_def
                   sat_eu__2__5_def)
  apply(simp)
  apply(blast)
 apply(simp only:trans_def)
 apply(clarify)
 apply(((erule disjE)?,
          simp only: coverrel_def,
          simp,
          ((erule disjE)?,
             simp only: sat__1__2_def
                        sat_eu__2__3_def
                        sat_eu__2__4_def
                        sat_eu__2__5_def,
             simp|blast)+)+)
```

□

38 Verifying ECCE

In this section we illustrate the automatic verification of the ECCE output by the ISABELLE system. To this end the theory, lemma and proof script as generated by ECCE for the Petri net of Example 1, are executed (the complete input consists of the ISABELLE specifications of Example 1, Example 5, and lemma and proof script of Example 6). Full details can be found in

the technical report [116]. After this, we can also apply the steps required to prove the lemma for transition `t1` in a similar fashion to the remaining transitions. The following proof script attempts precisely this. Again, the elimination rule `disjE` is not applicable for the last transition. Hence, we perform a test using `?` before applying this method in the first line.

```
apply(((erule disjE)?,
       simp only: coverrel_def,
       simp,
       ((erule disjE)?,
         simp only: sat__1__2_def
                    sat_eu__2__3_def
                    sat_eu__2__4_def
                    sat_eu__2__5_def,
         simp|blast)+)+)
```

For our example all cases could be verified, hence ISABELLE answers:

```
No subgoals!
```

□

Consequently, the coverability relation generated by ECCE for the Petri net of Example 1 covers indeed all states reachable by any path (under the condition that the theory generated by the automatic theory generator as implemented in ECCE is correct).

39 Automatic Generation of Hypotheses

Instead of defining the coverability as a relation as illustrated in Subsection 36.2 we may view the coverability graph as an inductive definition of a set of states which covers the actual state space of the Petri net. For our example a corresponding ISABELLE/ISAR definition could look as follows:

```
consts
 coverstates::  "state set"
inductive coverstates
intros
 zero :  "(sat__1__2 A) ∈ coverstates"
 step1 :  "⟦∃ A. (sat_eu__2__3 (Suc A)) ∈ coverstates⟧ ⟹
                                          (sat_eu__2__4 A) ∈ coverstates"
 step2 :  "⟦∃ A. (sat_eu__2__3 (Suc A)) ∈ coverstates⟧ ⟹
                                          (sat_eu__2__5 A) ∈ coverstates"
 step3 :  "⟦∃ A. (sat_eu__2__4 (Suc A)) ∈ coverstates⟧ ⟹
                                          (sat_eu__2__3 A) ∈ coverstates"
 step4 :  "⟦∃ A. (sat_eu__2__5 (Suc A)) ∈ coverstates⟧ ⟹
                                          (sat_eu__2__3 A) ∈ coverstates"
```

Similarly, instead of using the concept of paths, we may directly specify the set of reachable states inductively in ISABELLE/ISAR. For our example the following specification would fit the purpose:

```
consts
 reachstates::  "state set"
inductive reachstates
intros
 zero :  "(start B) ∈ reachstates"
 step1 :  "⟦∃ A B C D E. ((Suc A),(Suc B),(Suc C),D,E) ∈ reachstates⟧ ⟹
                                    (A,(Suc B),C,(Suc D),E) ∈ reachstates"
 step2 :  "⟦∃ A B C D E. ((Suc A),(Suc B),(Suc C),D,E) ∈ reachstates⟧ ⟹
                                    (A,B,(Suc C),D,(Suc E)) ∈ reachstates"
 step3 :  "⟦∃ A B C D E. (A,B,C,(Suc D),E) ∈ reachstates⟧ ⟹
                                    ((Suc A),B,(Suc C),D,E) ∈ reachstates"
 step4 :  "⟦∃ A B C D E. (A,B,C,D,(Suc E)) ∈ reachstates⟧ ⟹
                                    ((Suc A),(Suc B),C,D,E) ∈ reachstates"
```

Then, the lemma to be verified to show the soundness of the coverability relation is

```
lemma "x ∈ reachstates ⟹ x ∈ coverstates"
```

However, lets assume that the specification of `coverstates` is unknown and has to be generated by ISABELLE. To this end we may attempt to prove the following lemma:

```
lemma "∃ coverstates.  x ∈ reachstates ⟹ x ∈ coverstates"
```

Thereby it is not important to find a proof, since there are many sets which fulfill this criterion (e.g. the (minimal) set `reachstates` and the (maximal) set of all states). Instead it is important to find a proof, which generates the induction steps of the above specification of `coverstates` as (or as parts of) subgoals. In other words, the question is whether ISABELLE's proof methods can imitate the behaviour of ECCE (or other model checkers for Petri nets).

The most important elements of ECCE's partial deduction method to generate the coverability graph are: *coverability test*, *unfolding*, *whistling*, *abstraction*. The coverability test can easily be defined in ISABELLE/ISAR, e.g.:

```
⟦ x∈ state; y∈ state; x≤y⟧ ⟹ covers(y,x)
```

where $\leq$ is defined as an order on the set of states. We may also check whether a set of states is covered by another set of states, e.g.:

```
∀ B. ∃ A. covers((0,0,0,A,0),(0,0,0,(Suc B),0))
```

Similarly, we may define *whistling* for two states (state sets) or even for the states on a path (a whistle blows if a newly encontered state is (in some sense) bigger than any of its predecessors on the path, thereby it indicates a potentially infinite growth).

The *unfolding* corresponds in ISABELLE simply to the rewriting of a subgoal using a definition, in case of Petri nets the definition of the transition function.

The most difficult element to imitate seems to be the *abstraction*. Given a certain subgoal ISABELLE's proof method has to replace this subgoal by a more general one. E.g., if unfolding

of a transition has led to a subgoal containing the state `(0,0,0,(Suc 0),0)` and the whistle has blown due to a preceding state of the form `(0,0,0,0,0)`, then we have to replace the subgoal by a new one containing a state of the form `(0,0,0,A,0)` (where `A` is all quantified). The only proof rule which is capable of introducing an all quantified variable in ISABELLE/ISAR is `spec`:

$$\frac{\forall x.P}{P[t/x]}$$

And indeed, by applying `spec` as an introduction rule we may indeed introduce perform a generalisation. For example, assume the following subgoal:

```
1.  "(0,0,0,0,0) ∈ coverstates"
```

Executing `apply(rule spec)` and backtracking (using the proof command `back`) generates as the 30th possibility (out of 38):

```
1.  ∀ x.  (0, 0, 0, x, 0) ∈ coverstates
```

However, we did not succeed yet in implementing a complete proof script using this rule as the search for the appropriate alternative subgoal has to be controlled by the proof script. Within the execution oriented proof style we have focused on ISABELLE/ISAR does not seem to provide enough control without implementing new proof tactics on ISABELLE's ML-implementation level.

40 Conclusion and Further Work

We have shown the similarity between controlling partial deduction and inductive theorem proving. We have formally established a relationship between the program specialiser ECCE and the proof system ISABELLE when applied to verifying infinite state Petri nets. We have shown that verification of ECCE output using the proof system ISABELLE can be achieved for small nets. The execution of the proof script of Section 38 on a Pentium II/400 needed about 90s and the underlying PolyML required 80MB of memory. However, as further experiments with a net containing 14 places and 13 transitions reveiled, more specific proof methods have to be employed as the use of the method `blast` required more than the available 200MB of main memory and therefore had to be canceled. One way of tuning the proof process further is by restricting the number of rules potentially applied by `blast`. However, while rules can easily be removed from and added to the list of simplification rules in ISABELLE/ISAR, a similar simple manipulation of the "`blast` rules" without rewriting underlying ISABELLE proof tactics seems not possible. An indirect way of restricting the search space of `blast` could also be to derive the theory `PN` not from `Main` but from (sets of) more basic theories.

A way of improving the readability of the proof script could be to employ the mathematical proof style instead of the execution oriented style. In the mathematical proof style higher-order pattern matching can be used to control the proof. This may also increase the flexibility of the proof significantly, in particular if the results have to be generalised for other specifications than those of Petri nets.

Finally, for ISABELLE to automatically generate the coverability relation from the specification of the Petri net we believe that it is necessary to implement a new proof rule/proof method at ISABELLE's implementation level which allows to automatically backtrack over potential hypotheses which are more general than the subgoal to be shown. Another option worth exploring might be to attempt to define a proof scheme using the higher-order pattern matching of ISABELLE/ISAR, which performs the abstraction on proof level: E.g., if a state description matches a certain pattern we attempt to prove a lemma concerning a similar pattern where a constant is replaced by some variable.

Part V
Abstract Domains Based on Regular Types

We show how to transform a set of regular type definitions into a finite pre-interpretation for a logic program. The core of the transformation is a determinization procedure for non-deterministic finite tree automata. The derived pre-interpretation forms the basis for an abstract interpretation for logic programs. This approach provides a flexible way of building program-specific analysis domains. For a given set of types, precision is strictly improved compared to regular type analysis and set constraint analysis. The work also shows how various instantiation modes such as *ground*, *variable* and *non-variable* can be expressed as regular types and hence integrated with type analysis. We highlight applications in binding time analysis for offline partial evaluation and infinite-state model checking.

41 Background

Regular types are a familiar way of describing sets of terms. They may be either declared (prescriptive typing) or inferred (descriptive typing). Types are widely used in logic program development and analysis. Usually, we think of types as specifications of data structures such as lists, trees and so on.

There is a well-established connection between regular types and finite tree automata (FTAs). Roughly speaking it may be said that FTAs are specifications of regular types. However, FTAs can define sets that are not usually thought of as types, and type definition notations do not usually exploit the full expressiveness of FTAs. The method described in this paper uses general FTAs, even when the programmer uses restricted types, since the given types are transformed to disjoint types using standard algorithms from FTA theory.

In Section 42, the essential concepts from types and FTAs are introduced. Section 43 contains a review of the approach to logic program analysis based on pre-interpretations. In Section 44 it is shown how to derive a pre-interpretation from a given set of type definitions, and compute a model based on the pre-interpretation. Section 45 contains some examples. Implementation and complexity issues are discussed in Section 46.

42 Preliminaries

Tree automata are "machines" that recognise terms. Let Σ be a set of function symbols. Each function symbol in Σ has a rank (arity) which is a natural number. Whenever we write an

expression such as $f(t_1, \ldots, t_n)$, we assume that $f \in \Sigma$ and has arity n. We write f^n to indicate that function symbol f has arity n. If the arity of f is 0 we often write the term $f()$ as f and call f a *constant*.

The set of *ground terms* (or *trees*) Term_Σ associated with Σ is the least set containing the constants and all terms $f(t_1, \ldots, t_n)$ such that $t_1, \ldots, t_n$ are elements of Term_Σ and $f \in \Sigma$ has arity n.

Finite tree automata provide a means of finitely specifying possibly infinite sets of ground terms, just as finite automata specify sets of strings. A finite tree automaton (FTA) is defined as a quadruple $\langle Q, Q_f, \Sigma, \Delta \rangle$, where Q is a finite set called *states*, $Q_f \subseteq Q$ is called the set of accepting (or final) states, Σ is a set of ranked function symbols and Δ is a set of *transitions*. Each element of Δ is of the form $f(q_1, \ldots, q_n) \to q$, where $f \in \Sigma$ and $q, q_1, \ldots, q_n \in Q$.

FTAs can be "run" on terms in Term_Σ, a successful run of a term and an FTA is one in which the term is *accepted* by the FTA. The details are omitted here, except to say that whenever a term is accepted, it is associated with one of the final states of the FTA. Implicitly, a tree automaton R defines a set of terms, that is, a tree language, denoted $L(R)$, as the set of all terms that it accepts.

FTAs can be extended to allow ϵ-transitions, without altering their expressive power. An ϵ-transition is of the form $q \to q'$ where q and q' are states. Such transitions can be eliminaed from Δ, after adding all transitions $f(q_1, \ldots, q_n) \to q'$ such that there is a transition $f_(q_1, \ldots, q_n) \to q$ in Δ and a chain of ϵ-transitions $q \to \cdots \to q'$.

42.1 Tree Automata and Types

A type is simply regarded as an accepting state of an automaton. Given an automaton $R = \langle Q, Q_f, \Sigma, \Delta \rangle$, and $q \in Q_f$, define the automaton R_q to be $\langle Q, \{q\}, \Sigma, \Delta \rangle$. The language $L(R_q)$ is the set of terms corresponding to type q. We say that a term *is of type* q, written $t : q$, if and only if $q \in L(R_q)$.

Example 42.1 In the following examples, let $\Sigma = \{[]^0, [\_|\_]^2, leaf^1, tree^2, 0^0, s^1\}$, and let $Q = \{list, listnat, nat, zero, one, bintree, any, list0, list1, list2\}$. We define the set Δ_{any} to be the following set of transitions.

$$\{f(\overbrace{any, \ldots, any}^{n \text{ times}}) \to any \mid f^n \in \Sigma\}$$

- $Q_f = \{listnat\}$, $\Delta = \{[] \to listnat, [nat|listnat] \to listnat, 0 \to nat, s(nat) \to nat\}$. The type $listnat$ is the set of lists of natural numbers in successor notation.
- $Q_f = \{list\}$, $\Delta = \Delta_{any} \cup \{[] \to list, [any|list] \to list\}$. The type $list$ is the set of lists of arbitrary terms in Term_Σ.

- $Q_f = \{list2\}$, $\Delta = \{[] \rightarrow list0, [one|list0] \rightarrow list1, [zero|list1] \rightarrow list2, 0 \rightarrow zero, s(zero) \rightarrow one\}$. The type $list2$ is the set consisting of the single term $[0, s(0)]$.
- $Q_f = \{bintree\}$, $\Delta = \Delta_{any} \cup \{leaf(any) \rightarrow bintree, tree(bintree, bintree) \rightarrow bintree\}$. The type $bintree$ is the set of binary trees whose leaves are any terms in Term_Σ.
- $Q_f = \{list1\}$, $\Delta = \{[] \rightarrow list1, [one|list1] \rightarrow list1, [zero|list0] \rightarrow list1, [] \rightarrow list0, [zero|list0] \rightarrow list0, 0 \rightarrow zero, s(zero) \rightarrow one\}$. The type $list1$ is the set of lists consisting of zero or more elements $s(0)$ followed by zero or more elements 0 (such as $[s(0), 0], [s(0), s(0), 0, 0, 0], [0, 0], [s(0)], \ldots$).

42.2 Deterministic and Non-deterministic Tree Automata

There are two notions of non-determinism in tree automata: bottom-up and top-down.

It can be shown that (so far as expressiveness is concerned) we can limit our attention to FTAs in which the set of transitions Δ contains no two transitions with the same left-hand-side. These are called *bottom-up deterministic* finite tree automata. For every FTA R there exists a bottom-up deterministic FTA R' such that $L(R) = L(R')$.

Bottom-up deterministic FTAs define disjoint types, since each term is accepted by at most one accepting state. The transformation to bottom-up deterministic form can introduce an exponential number of new states, in the worst case. However, it is often useful and practical in the context of types. Example 42.2 illustrates the derivation of disjoint types from overlapping types.

An automaton $R = \langle Q, Q_f, \Sigma, \Delta \rangle$ is called *complete* if it contains a transition $f(q_1, \ldots, q_n) \rightarrow q$ for all n-ary functions $f \in \Sigma$ and states $q_1, \ldots, q_n \in Q$. We may always extend an FTA $\langle Q, Q_f, \Sigma, \Delta \rangle$ to make it complete, by adding a new state $q^\flat$ to Q. Then add transitions of the form $f(q_1, \ldots, q_n) \rightarrow q^\flat$ for every combination of f and states $q_1, \ldots, q_n$ (including $q^\flat$) that does not appear in Δ. Note that a complete bottom-up deterministic finite tree automaton in which every state is an accepting state is one which partitions the set of terms into disjoint subsets (types), one for each state. In such an automaton $q^\flat$ can be thought of as the error type, that is, the set of terms not accepted by any other type.

Example 42.2 Let $\Sigma = \{[]^0, [\_|\_]^2, 0^0\}$, and let $Q = \{list, listlist, any\}$. The set Δ_{any} is defined as before. let $Q_f = \{list, listlist\}$, $\Delta = \Delta_{any} \cup \{[] \rightarrow list, [any|list] \rightarrow list, [] \rightarrow listlist, [list|listlist] \rightarrow listlist, [listlist|listlist] \rightarrow listlist\}$. The type $list$ is the set of lists of any terms, while the type $listlist$ is the set of lists whose elements are of type $list$ or $listlist$.

The automaton is not bottom-up deterministic; for example, three transitions have the same left-hand-side, namely, $[] \rightarrow list$, $[] \rightarrow listlist$ and $[] \rightarrow any$. So for example the term $[[0]]$

is accepted by $list, listlist$ and any. A determinization algorithm can be applied, yielding the following. Intuitively, we can think of q_1 as the type $any \cap list \cap listlist$, $q2$ as the type $(list \cap any) - listlist$, and q_3 as $any - (list \cup listlist)$. Thus q_1, q_2 and q_3 are disjoint. The automaton is given by $Q = \{q_1, q_2, q_3\}$, Σ as before, $Q_f = \{q_1, q_2\}$ and $\Delta = \{[] \to q_1, [q_1|q_1] \to q_1, [q_2|q_1] \to q_1, [q_1|q_2] \to q_2, [q_2|q_2] \to q_2, [q_3|q_2] \to q_2, [q_3|q_1] \to q_2, [q_2|q_3] \to q_3, [q_1|q_3] \to q_3, [q_3|q_3] \to q_3, 0 \to q_3\}$. This automaton is also complete.

This determinization algorithm for this example will be discussed in more detail is Section 44.

A more restrictive kind of deterministic automaton can be defined, which is also highly relevant in the context of types. An FTA is *top-down deterministic* if it has no two transitions with both the same right-hand-side and the same function symbol on the left-hand-side (for example $f(q_1, q_2) \to q$ and $f(q_2, q_1) \to q$). When constructing a top-down derivation in a top-down deterministic automaton, there is thus at most one transition that can be used to construct a move for each leaf. Thus checking whether $t \in L(R)$ for such an automaton R can be done in $O(|t|)$ steps.

Top-down determinism introduces a loss in expressiveness. It is *not* the case that for each FTA R there is a top-down deterministic FTA R' such that $L(R) = L(R')$. Note that a top-down deterministic automaton can be transformed to an equivalent bottom-up deterministic automaton, as usual, but the result might not be top-down deterministic.

Example 42.3 Take the final automaton from Example 42.1. This is not top-down deterministic, due to the presence of transitions $[one|list1] \to list1, [zero|list0] \to list1$. No top-down deterministic automaton can be defined that has the same language.

Now consider the automaton with transitions $\Delta_{any} \cup \{[] \to list, [any|list] \to list\}$. This is top-down deterministic, but not bottom-up deterministic (since $[] \to list$ and $[] \to any$ both occur). Determinizing this automaton would result in one that is not top-down deterministic, since we would have disjoint types corresponding to $list$ and $q = any - list$. This would lead to transitions $[q|list] \to list$ and $[list|list] \to list$ which violates top-down non-determinism.

Despite the reduced expressiveness most type systems assume top-down deterministic types [160, 218].

42.3 Operations on Finite Tree Automata

Tree automata have a number of desirable properties and operations. The relevant ones in the present context are summarised below. Let R, R_1, R_2 be FTAs and t a term. Then $t \in L(R)$ is decidable and $L(R) = \emptyset$ is decidable. We can construct the product automaton $R_1 \times R_2$,

where $L(R_1 \times R_2) = L(R_1) \cap L(R_2)$, and the union automaton $R_1 \cup R_2$, where $L(R_1 \cup R_2) = L(R_1) \cup L(R_2)$. The complement automaton of R can also be constructed, which accepts those terms not accepted by R.

Most importantly for our purposes, given an automaton R a bottom-up deterministic automaton R' can be constructed, such that $L(R) = L(R')$. Also, given R a complete automaton R' can be constructed, such that $L(R) = L(R')$. The algorithms for determinization and completion will be examined in more detail in Section 44.

Further details on FTAs and their properties and associated algorithms can be found elsewhere [30].

43 Analysis Based on Pre-Interpretations

We now define the analysis framework for logic programs. Bottom-up declarative semantics captures the set of logical consequences (or a model) of a program. The theoretical basis of this approach to static analysis of definite logic programs was set out in [12], [11] and [58]. We follow standard notation for logic programs [151].

Let P be a definite program and Σ the signature of its underlying language L. A *pre-interpretation* of L consists of

1. a non-empty domain of interpretation D;
2. an assignment of an n-ary function $D^n \rightarrow D$ to each n-ary function symbol in Σ ($n \geq 0$).

A pre-interpretation with a finite domain D over a signature Σ defines a complete bottom-up deterministic FTA over the same signature. The domain D is the set of states of the FTA. Let $\hat{f}$ be the function $D^n \rightarrow D$ assigned to $f \in \Sigma$ by the pre-interpretation. In the corresponding FTA there is a set of transitions $f(d_1, \ldots, d_n) \rightarrow d$, for each $d_1, \ldots, d_n, d$ such that $\hat{f}(d_1, \ldots, d_n) = d$.

Let J be a pre-interpretation of L with domain D. Let V be a mapping assigning each variable in L to an element of D. A *term assignment* $T_J^V(t)$ is defined for each term t as follows:

1. $T_J^V(x) = V(x)$ for each variable x.
2. $T_J^V(f(t_1, \ldots, t_n)) = f'(T_J^V(t_1), \ldots, T_J^V(t_n))$, ($n \geq 0$) for each non-variable term $f(t_1, \ldots, t_n)$, where f' is the function assigned by J to f.

Definition 43.1 Domain atom

Let J be a pre-interpretation of a language L, with domain D, and let p be an n-ary function symbol from L. Then a domain atom for J is any atom $p(d_1, \ldots, d_n)$ where $d_i \in D$, $1 \leq i \leq n$.

Let $p(t_1, \ldots, t_n)$ be an atom. Then a *domain instance* of A with respect to J and V is a domain atom $p(T_J^V(t_1), \ldots, T_J^V(t_n))$. Denote by $[A]_J$ the set of all domain instances of A with respect to J and some V.

The definition of domain instance extends naturally to formulas. In particular, let C be a clause. Denote by $[C]_J$ the set of all domain instances of the clause with respect to J.

The core bottom-up declarative semantics is parameterised by a pre-interpretation of the language of the program.

Definition 43.2 Core bottom-up semantics function T_P^J

Let P be a definite program, and J a pre-interpretation of the language of P, with domain D. Let $Atom_D$ be the set of domain atoms with respect to J.

$$T_P^J : 2^{Atom_D} \rightarrow 2^{Atom_D}$$

$$T_P^J(I) = \left\{ A' \;\middle|\; \begin{array}{l} A \leftarrow B_1, \ldots, B_n \in P \\ A' \leftarrow B'_1, \ldots, B'_n \in [A \leftarrow B_1, \ldots, B_n]_J \\ \{B'_1, \ldots, B'_n\} \subseteq I \end{array} \right\}$$

$$\mathsf{M}^J[\![P]\!] = \mathsf{lfp}(T_P^J)$$

$\mathsf{M}^J[\![P]\!]$ is the minimal model of P with pre-interpretation J.

43.1 Interpretations of the Core Semantics

The usual declarative semantics is obtained by taking J to be the Herbrand pre-interpretation, which we call H. $\mathsf{M}^H[\![P]\!]$ is the minimal Herbrand model of P.

In order to capture information about the occurrence of variables, a modified version of H is taken, which will be called the *concrete semantics*. Let L be the language of program P, with signature Σ. We extend Σ with an infinite set of extra constants $\mathcal{V} = \{v_0, v_1, v_2, \ldots\}$. The Herbrand pre-interpretation over the extended language is called HV.

$\mathsf{M}^{HV}[\![P]\!]$, the minimal model with this pre-interpretation, is a set of terms that represents the set of *atomic logical consequences* of P. More precisely, let Ω be some fixed bijective mapping from $\mathcal{V}$ to the variables in L. Let A be an atom; denote by $\Omega(A)$ the result of replacing any constant v_j in A by $\Omega(v_j)$. Then the least model with respect to HV is the set of atomic logic consequence. More precisely, $A \in \mathsf{M}^{HV}[\![P]\!]$ iff $P \models \forall(\Omega(A))$.

The model $\mathsf{M}^{HV}[\![P]\!]$ is also known as the Clark semantics [25]. In practice (and in the rest of this paper) we ignore the Ω function and treat the constants $v_0, v_1, v_2, \ldots$ as variables

when they occur in elements of $\mathsf{M}^{HV}[\![P]\!]$. E.g. an atom $p(v_1, f(a, v_2), v_1)$ is interpreted as $p(x_1, f(a, x_2), x_1)$ where x_1, x_2 are variables. In this case we can simply state that $A \in \mathsf{M}^{HV}[\![P]\!]$ iff $P \models \forall A$.

A slightly simpler but more abstract version of the concrete semantics is defined by adding only a single special constant v rather than an infinite set $v_0, v_1, v_2, \ldots$. This is sufficient for capturing information about occurrence of variables (see Section 45). Let us call this pre-interpretation H_v.

43.2 Abstract Interpretations

Let J be any pre-interpretation and P a program. It can be shown that $\mathsf{M}^J[\![P]\!]$ is a model based on J.

Definition 43.3 [Concretisation of a model]

Let $\mathsf{M}^J[\![P]\!]$ be a model of P based on pre-interpretation J. The concretisation of $\mathsf{M}^J[\![P]\!]$ is a set of atoms defined as follows:

$$\gamma(\mathsf{M}^J[\![P]\!]) = \left\{ A \,\middle|\, [A]_J \subseteq \mathsf{M}^J[\![P]\!] \right\}$$

$\mathsf{M}^J[\![P]\!]$ is an abstraction of the atomic logical consequences of P, in the following sense.

Proposition 43.4 Let J be a pre-interpretation of $\Sigma \cup \mathcal{V}$ and $\mathsf{M}^J[\![P]\!]$ be the least model of P based on J. Then $\mathsf{M}^{HV}[\![P]\!] \subseteq \gamma(\mathsf{M}^J[\![P]\!])$.

A similar proposition holds if we replace $\Sigma \cup \mathcal{V}$ by $\Sigma \cup \{v\}$ and HV by H_v.

43.3 Abstract Compilation of a Pre-Interpretation

The idea of abstract compilation was introduced first by Debray and Warren [45]. Operations on the abstract domain are coded as logic programs and added directly to the target program, which is then executed according to standard concrete semantics. The reason for this technique is to avoid some of the overhead of interpreting the abstract operations.

A pre-interpretation can be defined by two predicates, $domain/1$ and $denotes/2$. They are given suitable definitions as follows.

- $domain(d)$ is true iff d is in the domain of interpretation.
- $denotes(f(d_1, \ldots, d_n), d)$ is true if the pre-interpretation of the n-ary function f maps $\langle d_1, \ldots, d_n \rangle$ to d

These two predicates are incorporated directly in the program to be analysed. Each clause of the program of the form is transformed by

1. repeatedly replacing non-variable subterms $f(r_1, \ldots, r_m)$ in the clause by a fresh variable u and adding the atom $denotes(f(r_1, \ldots, r_m), u)$ to the clause body, until the only non-variables in the clause occur in the first argument of $denotes$;

2. adding $domain(z)$ to the body of each clause in which variable z occurs in the head but not the body.

If P is the original program, the transformed program is called $\bar{P}$.

When specific definitions of $domain/1$ and $denotes/2$ defining a pre-interpretation J are added to $\bar{P}$, the result is a *domain program* for J, called $\bar{P}^J$. Clearly $\bar{P}^J$ has a different language than P, since the definitions of $domain/1$ and $denotes/2$ contain elements of the domain of interpretation. It can easily be shown that the minimal Herbrand model of $\bar{P}^J$ (restricted to the original program predicates) is isomorphic to $\mathsf{M}^J[\![P]\!]$.

43.4 Computation of the Least Domain Model

The least model $\mathsf{M}^J[\![P]\!] = \mathsf{lfp}(T_P^J)$ is obtained by computing $\mathsf{lfp}(T_{\bar{P}^J})$, and then restricting to the predicates in P (that is, omitting the predicates $denotes$ and $domain$ which were introduced in the abstract compilation). Optimised algorithms for computing $\mathsf{lfp}(T_P)$ for an arbitrary program P have been developed (see Section 46).

44 Deriving a Pre-Interpretation from Regular Types

An algorithm for transforming a non-deterministic FTA (NFTA) to a deterministic FTA (DFTA) is presented in [30]. The algorithm is shown here in a modified version that is more suitable for implementation:

input: NFTA $R = \langle Q, Q_f, \Sigma, \Delta \rangle$,
begin
  Set Q_d to $\emptyset$
  Set Δ'_d to $\emptyset$
  repeat
    Set $\Delta_d = \Delta'_d$
    for each $f^n \in \Sigma$
      for each choice $s_1, \ldots, s_n \in Q_d$

for each $q_1, \ldots, q_n \in s_1 \times ... \times s_n$
$s = \{q \in Q | f(q_1, \ldots, q_n) \rightarrow q \in \Delta\}$
if $s \neq \emptyset$ then
Set $\Delta'_d = \Delta'_d \cup \{f(s_1, \ldots, s_n) \rightarrow s\}$
Set Q_d to $Q_d \cup \{s\}$
end if
end for each
end for each
end for each
until $\Delta'_d = \Delta_d$
Set Q_{d_f} to $\{s \in Q_d \mid s \cap Q_{d_f} \neq \emptyset\}$
output: DFTA $R_d = \langle Q_d, Q_{d_f}, \Sigma, \Delta_d \rangle$
end

Description: The algorithm transform the NFTA from one that operates on states, to one that operates on sets containing states from the NFTA. The NFTA allowed multiple occurrences of the same state on the left hand side of a transition. In the DFTA, which is the output of the algorithm, all reachable states in the NFTA are contained in sets that makes up the new states - these are contained in the set Q_d. A state in the NFTA *can* occur in more than state in the DFTA. Potentially every non-empty subset of set of states of the NFTA can be a state of the DFTA.

The sets in Q_d and the new set of transitions, Δ_d, are generated in an iterative process. In an iteration of the process, a function f is chosen from Σ. Then a number of sets, $s_1, \ldots, s_n$ corresponding to the arity of f, is selected from Q_d - the same set can be chosen more than once. The cartesian product is then formed, $(s_1 \times \cdots \times s_n)$, and for each element in the cartesian product, $q_1, \ldots, q_n$, such that a transition $f(q_1, \ldots, q_n) \rightarrow q$ exists, q is added to a set s. When all elements in the cartesian product have been selected, the set s is added to Q_d if s is non-empty and not already in Q_d. A transition $f(s_1, \ldots, s_n) \rightarrow s$ is added to Δ_d if s is non-empty.

The algorithm terminates when Q_d is such that no new transitions are added. Initially Q_d is the empty set, so no set containing a state can be chosen from Q_d and therefore only the constants (0-ary functions) can be selected.

Example 44.1 In example 42.2 a non-deterministic FTA is shown; $\Sigma = \{[]^0, [\_|\_]^2, 0^0\}$, $Q = \{list, listlist, any\}$, $\Delta = \Delta_{any} \cup \{[] \rightarrow list, [any|list] \rightarrow list, [] \rightarrow listlist, [list|listlist] \rightarrow listlist, [listlist|listlist] \rightarrow listlist\}$.

A step by step application of the algorithm follows:

Step 1: $Q_d = \emptyset, \Delta_d = \emptyset$. Choose f as a constant, $f = []$. Now $s = \{q \in Q \mid [] \to q \in \Delta\} = \{any, list, listlist\}$. Add s to Q_d and the transition $[] \to \{any, list, listlist\}$ to Δ_d.

Step 2: Choose $f = 0$. Now $s = \{q \in Q \mid 0 \to q \in \Delta\} = \{any\}$. Add s to Q_d and the transition $0 \to \{any\}$ to Δ_d.

Step 3: Choose $f = [\_ \mid \_]$, $s_1 = s_2 = \{any, list, listlist\}$. Now $s = \{q \in Q \mid \exists q_1 \in s_1, \exists q_2 \in s_2, [q_1 \mid q_2] \to q \in \Delta\} = \{any, list, listlist\}$. Add s to Q_d and the transition $[\{any, list, listlist\} \mid \{any, list, listlist\}] \to \{any, list, listlist\}$ to Δ_d.

Step 4: Choose $f = [\_ \mid \_]$, $s_1 = s_2 = \{any\}$. Now $s = \{q \in Q \mid \exists q_1 \in s_1, \exists q_2 \in s_2, [q_1 \mid q_2] \to q \in \Delta\} = \{any\}$. Add s to Q_d and the transition $[\{any\} \mid \{any\}] \to \{any\}$ to Δ_d.

Step 5: Choose $f = [\_ \mid \_]$, $s_1 = \{any\}, s_2 = \{any, list, listlist\}$. Now $s = \{q \in Q \mid \exists q_1 \in s_1, \exists q_2 \in s_2, [q_1 \mid q_2] \to q \in \Delta\} = \{any, list\}$. Add s to Q_d and the transition $[\{any\} \mid \{any, list, listlist\}] \to \{any, list\}$ to Δ_d.

Step 6: Choose $f = [\_ \mid \_]$, $s_1 = \{any, list, listlist\}, s_2 = \{any\}$. Now $s = \{q \in Q \mid \exists q_1 \in s_1, \exists q_2 \in s_2, [q_1 \mid q_2] \to q \in \Delta\} = \{any\}$. Add s to Q_d and the transition $[\{any, list, listlist\} \mid \{any\}] \to \{any\}$ to Δ_d.

Step 7 to 11: No new sets added to Q_d. New transitions added: $[\{any, list\} \mid \{any, list\}] \to \{any, list\}$, $[\{any, list\} \mid \{any, list, listlist\}] \to \{any, list, listlist\}$, $[\{any, list, listlist\} \mid \{any, list\}] \to \{any, list\}$, $[\{any\} \mid \{any, list\}] \to \{any, list\}$, $[\{any, list\} \mid \{any\}] \to \{any\}$.

Resulting DFTA: $\Sigma = \{[]^0, [\_|\_]^2, 0^0\}$, $Q_d = \{\{any, list, listlist\}, \{any\}, \{any, list\}\}$, $Q_{d_f} = \{\{any, list, listlist\}, \{any, list\}\}$, $\Delta_d = \{[] \to \{any, list, listlist\}, 0 \to \{any\}, [\{any, list, listlist\} \mid \{any, list, listlist\}] \to \{any, list, listlist\}, [\{any\} \mid \{any\}] \to \{any\}, [\{any\} \mid \{any, list, listlist\}] \to \{any, list\}, [\{any, list, listlist\} \mid \{any\}] \to \{any\}, [\{any, list\} \mid \{any, list\}] \to \{any, list\}, [\{any, list\} \mid \{any, list, listlist\}] \to \{any, list, listlist\}, [\{any, list, listlist\} \mid \{any, list\}] \to \{any, list\}, [\{any\} \mid \{any, list\}] \to \{any, list\}, [\{any, list\} \mid \{any\}] \to \{any\}\}$.

The states in Q_d are equivalent to the states q_1, q_2, q_3 in example 42.2. q_1 is equivalent to the type $any \cap list \cap listlist$ represented in Q_d as the set $\{any, list, listlist\}$, q_2 is equivalent to the type $(list \cap any) - listlist$ represented by the set $\{any, list\}$ and finally q_3 is equivalent to the type $any - (list \cup listlist)$ represented by the set $\{any\}$.

In a naive implementation of the algorithm where every combination of arguments to the chosen f would have to be tested in each iteration, the complexity lies in forming and testing each element in the cartesian product, for every combination of states in Q_d. It is possible to estimate of the number of operations required in a single iteration of the process, where an

$$rev([],[]).$$
$$rev([X|Xs],Zs) \leftarrow rev(Xs,Ys), app(Ys,[X],Zs).$$

$$app([],Ys,Ys).$$
$$app([X|Xs],Ys,[X|Zs]) \leftarrow app(Xs,Ys,Zs).$$

Figure 18: Naive Reverse program

operation is the steps necessary to determin whether $f(q_1,\ldots,q_n) \rightarrow q \in \Delta$. Since Δ is static, an operation can be considered to be of constant time. The number of operations can be estimated by the formula $\#op = (s*e)^a$, where s is the number of states in Q_d, e is the number of elements in a single state in Q_d (possibly an estimate) and a is the arity of the chosen f. Every time a state is added to Q_d, an iteration in the algorithm will require additional operations. Worst case is if the algorithm causes an exponential blow-up in the number of states[30].

45 Examples

In this section we look at typical examples involving types and modes. It will be seen how types and modes can be mixed. The usefulness of this approach in a binding time analysis (BTA) for offline partial evaluation will be shown. We also illustrate the applicability of the domains to model-checking and failure detection.

To align with typical type notations, types will be defined using type rules, which are alternative syntax for FTAs. Let Σ be a set of function symbols, each with an arity, and N be a set of types. A type rule is of the form $t \rightarrow R_1 \mid \cdots \mid R_k$, $(k > 0)$, where $t \in N$ is a type and $\{R_1,\ldots,R_k\} \subseteq \mathsf{Term}_{\Sigma \cup N}$. This is sometimes written as k type rules $t \rightarrow R_1, \ldots, t \rightarrow R_k$.

In all the following examples, the type any is defined by the set of all rules of the form $any \rightarrow f(\overbrace{any,\ldots,any}^{n \text{ times}})$ for each n-ary function $f \in \Sigma$. We assume that Σ includes one special constant v (see Section 43). That is, the type rules for any include the rule $any \rightarrow v$.

45.1 Simple Lists

Consider the usual definition of lists, and assume that the signature contains at least one function other than [] and [.|.].

$$list \rightarrow [] \mid [any|list]$$

Together with the rules for any, this is (bottom-up) non-deterministic, since we have both $list \rightarrow []$ and $any \rightarrow []$. The determinization algorithm of Section 44 yields two states $\{any, list\}$ and $\{any\}$, representing the set of lists and the set of non-lists (which is $any - list$). Let us abbreviate these as $list$ and $nonlist$ respectively. Analysis of the naive reverse program in Figure 18 with predicates $rev/2$ and $app/3$ yields the model $\{rev(list, list), app(list, X, X)\}$ (where X is $list$ or $nonlist$), indicating that $rev/2$ succeeds only with lists in both arguments, while $app/3$ succeeds with a list in the first argument, while the second and third arguments are either both lists or both nonlists.

45.2 Simple Groundness

Recall that the concrete semantics is defined over an extended language containing an extra constant v representing variables. Using this information, we can define the set of ground terms as those that do not include any occurrence of the extra constant. The type rules for a type $ground$ are all rules of the form $ground \rightarrow f(\overbrace{ground, \ldots, ground}^{n \text{ times}})$ for every n-ary function f apart from the special constant v. Thus the type $ground$ represents a subset of the type any.

The type $ground$ is already bottom-up deterministic, but not complete, since any terms containing the special constant v are not recognised. Completion adds a new type (called $other$) and type rules defining $other$, including $other \rightarrow v$ and $other \rightarrow f(\ldots, other, \ldots)$ for each function symbol.

Analysis of naive reverse with respect to this pre-interpretation is isomorphic to the POS abstract domain of boolean groundness dependencies. For the naive reverse program the abstract model is

$$\{rev(ground, ground), rev(other, other),$$
$$app(ground, X, X), app(other, other, other), app(other, ground, other)\}.$$

45.3 Simple Lists with Groundness

Consider the set of type rules for $ground$, $list$ and any as given in the previous examples. Note that the types $list$ and $ground$ intersect. After determinization, we obtain four states $\{any, ground, list\}$, $\{any, list\}$, $\{any, ground\}$, $\{any\}$, representing (i) ground lists (gl) (ii) non-ground lists (ngl) (iii) ground non-lists (gnl) and (iv) non-ground non-lists ($other$) respectively. Analysis of the naive reverse program yields the following abstract model:

$$\{rev(gl, gl), rev(ngl, ngl),$$
$$app(gl, X, X), app(ngl, gl, ngl), app(ngl, ngl, gnl),$$
$$app(ngl, gnl, other), app(ngl, other, other)\}.$$

45.4 Static, Dynamic and Non-variable Types for Binding Time Analysis

Binding time analysis (BTA) for offline partial evaluation in Logen [133] distinguishes between various kinds of term instantiations. $Static$ corresponds to $ground$, and $dynamic$ to any. In addition we add a type var with the single type rule $var \rightarrow v$, where v is the special extra constant.

Determinization of these types yields three states $\{dynamic, static\}$, $\{dynamic, var\}$ and $\{dynamic\}$, representing three disjoint types containing respectively ground terms ($ground$), variables (var) and non-variable non-ground terms ($nvng$). Analysis of naive reverse yields the following model.

$$\{rev(ground, ground), rev(nvng, nvng),$$
$$app(ground, var, nvng), app(ground, var, var), app(ground, ground, ground),$$
$$app(ground, nvng, nvng), app(nvng, X, nvng)\}.$$

The presence of var in an argument indicates possible freeness, or alternatively, the absence of var indicates definite non-freeness. For example, the answers for rev are definitely not free, the first argument of app is not free, and if the second argument of app is not free then neither is the third. Such dependencies allow accurate propagation of binding time information.

45.5 BTA types Combined with Program-specific Types

The types described above can be added to user-defined or automatically inferred types defining data structures. Adding the type $list \rightarrow [] \mid [dynamic|list]$ and determinizing results in the generation of types representing static lists (ground lists) and non-ground lists, in addition to the types of the previous example. More refined types such as lists of lists, lists of integers and so on can be added.

45.6 Detecting Failures

Regular type analysis has been used to detect unsuccessful (failing or looping) computations. Building a type domain as described in this paper allows extra precision to be obtained, from the same set of types.

For example, consider the set of lists, and the set of lists of element a. The respective types are

$$list \rightarrow [] \mid [any|list]$$
$$lista \rightarrow [] \mid [atype|lista]$$
$$atype \rightarrow a$$

Determinization of these types, along with the type any (which includes the rules $any \rightarrow a$ and $any \rightarrow b$) yields four states $\{any, list, lista\}$, $\{any, list\}$, $\{any, typea\}$ and $\{any\}$ namely, lists of a (which we will call $lista$) other lists ($listnona$), the constant a ($typea$) and non-a non-list terms ($other$). Consider the following $memb$ program.

$$memb(X, [X|Y]).$$
$$memb(X, [Z|Y]) \leftarrow memb(X, Y)$$

Analysis of the standard $memb$ program with respect to the determinized types yields the model

$$\{memb(typea, lista), memb(typea, listnona), memb(lista, listnona),$$
$$memb(listnona, listnona), memb(other, listnona), memb(A, other))\}$$

Note that the atom $memb(other, lista)$ is not present. From this, it can be concluded that, for example, the goal $\leftarrow makeLista(X), memb(b, X)$ fails, where $makeLista(X)$ is assumed to succeed only with a list of a. This is because b is of type $other$, and the atom $memb(other, alist)$ is not in the abstract least model of $memb$.

Note that techniques such as set-based analysis and regular type inference could also detect such failures. However a top-down analysis of the goal would be required. In the example above, a bottom-up analysis of the $memb$ predicate is sufficient, independent of the goals for $memb$. This is because the abstract domain is condensing [95] and so the same precision is gained by restricting a bottom-up analysis to a given goal as for performing a goal-dependent analysis.

45.7 Infinite-State Model Checking

The following example is by Charatonik and Podelski [21]; it is a simple model of a token ring transition system. A state of the system is a list of processes indicated by 0 and 1 where a 0 indicates a waiting process and a 1 indicates an active process. The initial state is defined by the predicate gen and the the predicate $reachable$ defines the reachable states with respect to the transition $trans$. The required property is that exactly one process is active in any state. The state space is infinite, since the number of processes (the length of the lists) is unbounded. Hence finite model checking techniques do not suffice. The example was used in [21] to illustrate set constraint techniques for infinite-state model checking.

$$gen([0, 1]).$$
$$gen([0|X]) \leftarrow gen(X).$$

$$trans(X, Y) \leftarrow trans1(X, Y).$$
$$trans([1|X], [0|Y]) \leftarrow trans2(X, Y).$$

$$trans1([0, 1|T], [1, 0|T]).$$
$$trans1([H|T], [H|T1]) \leftarrow trans1(T, T1).$$

$$trans2([0], [1]).$$
$$trans2([H|T], [H|T1]) \leftarrow trans2(T, T1).$$

$$reachable(X) \leftarrow gen(X).$$
$$reachable(X) \leftarrow reachable(Y), trans(Y, X).$$

We define simple regular types defining the states. The set of all lists of ones and zeros is called $list$ and the set of "good" states in which there is exactly one 1 is $goodlist$. The type $zerolist$ is the set of list of zeros.

$$one \rightarrow 1$$
$$zero \rightarrow 0$$
$$list \rightarrow [] \mid [zero|list] \mid [one|list]$$
$$goodlist \rightarrow [zero|goodlist] \mid [one|zerolist]$$
$$zerolist \rightarrow [] \mid [zero|zerolist]$$

Determinization of these types along with any results in six states representing disjoint types: $\{any, one\}$, $\{any, zero\}$, the good lists $\{any, list, goodlist\}$, the zero lists $\{any, list, zerolist\}$, the non-goodlist, non-zerolist lists $\{any, list\}$ and $\{any\}$ for all other terms. We abbreviate these as one, $zero$, $goodlist$, $zerolist$, $badlist$ and $other$ respectively. The least model of the above program over this domain is as follows.

$$\{gen(goodlist),$$
$$trans2(badlist, badlist), trans2(other, other),$$
$$trans2(goodlist, badlist), trans2(goodlist, goodlist)$$
$$trans1(goodlist, goodlist), trans1(badlist, badlist), trans1(other, other),$$
$$trans(goodlist, goodlist), trans(badlist, badlist), trans(other, other),$$
$$reachable(goodlist)\}$$

The key property of the model is the presence of $reachable(goodlist)$ (and the absence of other atoms for $reachable$), indicating that if a state is reachable then it is a $goodlist$. Note that the transitions will handle $badlist$ and $other$ states, but in the context in which they are invoked, only $goodlist$ states are encountered.

46 Implementation and Complexity Issues

The implementation is based on three components; the FTA *determinization* algorithm described in Section 44, the *abstract compilation* transformation and the *fixpoint* algorithm for computing the least model.

We have implemented all three components. The determinization algorithm is a prototype based on a relatively direct implementation of the algorithm as presented in Section 44: it is clearly amenable to major optimization. Nevertheless the scalability of the determinization algorithm in Section 44 is a critical topic for future study and experiment.

Abstract compilation is a simple transformation with no serious complexity or implementation problems.

The computation of the least model is an iterative fixpoint algorithm. Various optimisations have been applied. The *predicate dependency graph* of a program has the predicates of a program as nodes, and there is a directed arc from p to q to iff q appears in the body of a clause in which p is in the head. A directed graph can be split into *strongly connected components* (SCCs). The SCCs are the largest sets of nodes such that there is a path from any element in a set to any other in the same set.

The iterations of the basic fixpoint algorithm, which terminates when a fixed point is found, can be decomposed into a sequence of smaller fixpoint computations. Each subcomputation returns the solution of a group of mutually recursive predicates. Breaking down the computation in this way has several advantages.

- Relatively few clauses are solved on each iteration.
- Not every atom in a clause body needs to be resolved on each iteration.
- The fixed point for non-recursive groups is found in one iteration.

An algorithm, linear in the size of the graph, for finding the SCCs of a directed graph was discovered by Tarjan [207]. Furthermore, the algorithm naturally returns the SCCs in a sequence consistent with the graph. (There is more than one possible sequence). In other words, if there is a path from node x to node y in the graph, then either x and y are in the same component, or x's component precedes y's component.

In addition to the SCC optimisation, our implementation incorporates a variant of the *semi-naive* optimisation [211], which makes use of the information about new results on each iteration. A clause body containing predicates whose models have not changed on some iteration need not be processed on the next iteration.

Our fixpoint implementation has been extensively used on programs with up to 4000 clauses. The key finding is that if the SCCs are relatively small (mutually recursive groups with more than 2 or 3 predicates are rare) then the analysis is roughly linear in the number of SCCs.

47 Related Work and Conclusions

The approach described in this paper provides an integration of regular type abstractions with discrete abstract domains expressed as pre-interpretations. We showed how to transform any given regular type into a pre-interpretation, using standard algorithms on Finite Tree Automata, namely, determinization and completion.

The domain of the pre-interpretation is a set of disjoint types, partitioning the set of terms. The least model under this pre-interpretation provides accurate information about the success types, including type dependencies, with respect to these disjoint types (and hence with respect to the original types from which they were derived).

The analysis domain induced by the pre-interpretation is condensing, which implies that a bottom-up analysis (usually much cheaper and more scalable than goal-dependent analysis) can be used as the basis for goal-directed analysis, with no loss in precision.

Applications in binding time analysis for offline partial evaluation have been investigated, with promising results. As noted in Section 45 various mode analyses can be reproduced with this approach, including the well-known POS analysis.

The potential of this method for model-checking, by detecting unreachable states (represented as predicates which are proved unsolvable) seems to be considerable, since the approach seems both faster and more precise than set constraint analysis, which is already useful [21].

Part VI
Abstract Interpretation with Specialized Definitions

The relationship between abstract interpretation and partial deduction has received considerable attention and (partial) integrations have been proposed from both the partial deduction and abstract intepretation perspectives. In this work we present what we argue is the first *full* integration of abstract interpretation and partial deduction from an abstract interpretation perspective. The proposed framework can be used both for analysis and specialization of logic programs and provides results which are strictly more precise than those achievable by the individual techniques. Interestingly, the central idea in this framework is simple: the abstract interpretation algorithm is modified in such a way that calls in the program are not analyzed w.r.t. the definition of the procedure in the original program but rather w.r.t. a *specialized definition* of the procedure for the given call. The process of obtaining a specialized definition from the original definition corresponds to the transformations performed during on-line program specialization, including unfolding. This apparently simple modification to the analysis algorithm has important consequences. First, the analysis process can be improved both in terms of efficiency and accuracy. Second, the set of specialized definitions computed during analysis provide a powerful partial evaluation of the program. Third, the new features of the framework introduce non-trivial termination and control issues which are studied in the paper. The framework has been implemented in the context of the `CiaoPP`analysis and specialization system. We briefly describe this implementation.

48 Introduction

Abstract interpretation [35] is a well known technique for static analysis of programs. It allows obtaining at compile-time safe approximations about the run-time behavior of the program. The information obtained by means of abstract interpretation has long been used for both program optimization and verification. A typical approach to abstract interpretation-based program optimization is to analyze the program in order to obtain a safe approximation of the states at which the corresponding *program point* can be reached. This is done by annotating the program points of interest with abstract substitutions which are guaranteed to be safe approximations. Then, these annotations are used to optimize the code as much as possible. Such optimizations can be performed both at the source-level and afterward at the compiled-code level. If the abstract interpretation framework is *multivariant* on calls, the same program procedure can be analyzed

for different (abstract) call patterns. This has two effects. On one hand it may allow improving the accuracy of analysis results since different call patterns do not need to be collapsed on a single one. On the other hand, a *multiply specialized* program [189] may be achieved by expanding the program in such a way that a different implemented version is generated by each pair $\langle procedure, call\ pattern \rangle$.

Partial evaluation [99]—often referred to simply as program specialization—optimizes programs by specializing them for (partially) known static data. Essentially, partial evaluators are non-standard interpreters which evaluate the known data while enough information is available and produce residual code otherwise. The partial evaluation of logic programs, also known as *partial deduction* [152, 57], has received considerable attention. The shortcomings of traditional partial deduction when compared to abstract interpretation techniques have been identified early on, and several partial solutions to overcome these limitations have been proposed in the literature. The shortcomings are related to two sources of precision loss during partial deduction. One is related to the lack of information propagation among different concrete call patterns (often refereed to as *atoms*) once they are transferred to the global control. The second one is related to the usage of the most specific generalization operator as a means of generalizing call patterns in order to guarantee that the set of atoms which are specialized remains finite. Existing improved partial deduction systems are available which overcome some of these problems by different means. However, to the best of our knowledge, no system actually has overcome all the previously mentioned shortcomings simultaneously.

More recently, a very general framework called *abstract* partial deduction [124] has been proposed which provides very interesting insights into the way an integrated framework should look like. This formalization departs from traditional partial deduction in several ways. It includes the use of an abstract domain and replaces the classical unfolding operation by two operations, abstract unfolding and abstract resolution. These can be defined in different ways and this work provides the conditions which these operations have to satisfy in order for the whole specialization system to be correct. It is proved that several frameworks, including traditional partial deduction, are instances of this more general one.

There has also been significant progress following the alternative approach of starting from an abstract specialization perspective. In [192] the relationship between partial deduction and abstract interpretation is studied from this point of view. This work identifies the need to allow performing unfolding steps during analysis time and proposes several possibilities for the practical integration of such unfolding. However, it also identifies that the questions of when and how to perform unfolding in the integrated framework are not trivial at all. In fact, that is the central motivation of this work. Summarizing, our answer to the question of "when to perform unfolding" is to do it right before analyzing the particular call. And our answer to the question of

"how to perform unfolding" is to do it by obtaining a specialized definition of a predicate from the original one and then letting the analyzer process the specialized definition. The resulting abstract interpretation-based specialization framework, which we propose herein, requires almost no modification to existing analyzers, other than the addition of a way to obtain the specialized definitions from the original ones. However, the modifications introduced in the new framework have a big impact in terms of efficiency and accuracy thanks to the combination of *execution* and *approximation* during specialization.

48.1 Approximation vs. Execution

Traditional partial deduction is characterized by *executing* (concrete) atoms and goals for producing resultants—i.e., specialized rules—in which the partial computed answer substitution is actually applied to the resultant. On the other hand, abstract interpretation is characterized by *simulating* the execution of the program using abstract substitutions instead of concrete ones. Both approaches have advantages and disadvantages. An important advantage of partial evaluation is that of efficiency: propagating and applying substitutions is straightforward to implement on a (constraint) logic programming system and very efficient to execute. The disadvantage is that no information can be propagated about unbound variables. Another advantage of execution is that it improves accuracy when compared to abstract interpretation under certain circumstances. If concrete information is available, it will in general be more precise than abstract descriptions, In contrast, abstract interpretation allows obtaining safe approximations of the actual values in situations in which concrete execution would not capture any information, either because it corresponds to a variable or because the set of concrete values which would require to be handled would be infinite or too large to be practical.

Given this situation, it seems difficult to design a system which integrates abstract interpretation and partial deduction by using concrete substitution or abstract substitution only. For example, abstract partial deduction introduces abstract substitutions to the partial deduction algorithm, since concrete substitutions are not enough. From the point of view of abstract interpretation it may seem feasible to capture both concrete and abstract values by using a refined abstract domain.

In this work we advocate for a hybrid approach which allows propagating both concrete and abstract substitutions. This has both theoretical and practical advantages: it allows conceptually separating the information generated as a result of specializing definitions, which uses concrete bindings, from that approximated by the analysis algorithm, which remains abstract. The practical advantage is that the analysis algorithm remains basically unmodified and there is no need to implement new abstract domains which capture both concrete and abstract values separately.

On the other hand, the specializer will mostly work with concrete information, though abstract information can also be used in order to remove useless rules from the specialized definition or to perform abstract execution of some of the literals in the specialized definitions.

49 Preliminaries

This section recalls preliminary concepts on logic programming and abstract interpretation [35]. Terms are constructed from variables (e.g., X), functors (e.g., f) and predicates (e.g., p). We denote by $\{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$ the *substitution* σ with $\sigma(X_i) = t_i$ for all $i = 1, \ldots, n$ (with $X_i \neq X_j$ if $i \neq j$) and $\sigma(X) = X$ for any other variable X, where t_i are terms. We denote by $vars(O)$ the set of variables in a syntactic object O.

An *atom* A has the form $p(t_1, ..., t_n)$ where p is a predicate symbol, and the t_i are terms. Function $pred(A)$ returns the predicate symbol p for the atom A. We use *Atoms* to denote the set of atoms. We say that an atom A is more general than another atom A' and we denote it $A' \subseteq A$ if $\exists$ a substitution θ s.t. $A' = A\theta$. A *goal* is a finite sequence of atoms $A_1, \ldots, A_n$. A *rule* is of the form $H \leftarrow B$ where H, the *head*, is an atom and B, the *body*, is a possibly empty finite sequence of literals. A *program*, is a finite set of rules. A *renamed apart* rule for a rule R in a program P is another rule R' such that $vars(R') \cap vars(P) = \emptyset$ and there is a renaming ρ such that $R' = R\rho$. Given an atom A and a program P, we denote by $Def(A, P)$ the set of renamed apart rules for the rules $H \leftarrow B$ in P with $pred(A) = pred(H)$, such that H unifies with A.

In this work, we assume a top-down operational semantics for logic programs under the standard (Prolog) left-to-right computation rule, i.e., LD resolution. At each stage, the current goal G can be represented by $\leftarrow \sigma_i(A_1, \ldots, A_n)$ where $\sigma_i = \theta_1 \ldots \theta_n$ is the composition of the substitutions applied so far (the *accumulated* substitution). For the initial goal, we have that $\sigma_0 = id$, the empty substitution. To perform a *SLD derivation step*, the computation rule selects the leftmost subgoal $\sigma_i(A_1)$. The search rule selects a rule $H \leftarrow B_1, \ldots, B_m$ renames it apart and unifies the head H with $\sigma_i(A_i)$. If the unification is successful with $mgu\ \theta_{i+1}$, then the goal statement $\leftarrow \sigma_{i+1}(B_1, \ldots, B_m, A_2, \ldots, A_n)$ is derived with $\sigma_{i+1} = \sigma_i\theta_{i+1}$. An *SLD-derivation* consists of a possibly infinite sequence $G_0 = G, G_1, \ldots$ of goals, a sequence of $C_1, C_2, \ldots$ of properly renamed rules of P, a sequence $\theta_1, \theta_2, \ldots$ of mgus such that each G_{i+1} is derived from G_i and C_{i+1} with $mgu\ \theta_{i+1}$. Given a finite SLD derivation D of $P \cup \{\leftarrow G\}$ ending in $\leftarrow B$ and θ the composition of mgus in the derivation steps, we say that θ restricted to the variables of G is the *computed answer substitution* (*c.a.s.*). A derivation D is a SLD refutation, or *successful* derivation if it ends in the empty goal. In such case, θ restricted to the variables of G is also simply called a *computed answer*. A derivation D is failed if the current goal is nonempty and no derivation step can be performed. The operational semantics of an

atom A is defined in terms of its computed answers, i.e., for a program P, we write $[\![A]\!]_P = \{A\theta | \theta \text{ is a computed answer for } A \text{ in } P\}$ to denote the semantics of the goal for the program.

Static program analysis aims at deriving at compile-time certain properties of the run-time behavior of a program. Abstract interpretation [35] is arguably one of the most successful techniques for static program analysis. In abstract interpretation, the execution of the program is "simulated" on an *abstract domain* (D_α) which is simpler than the actual, *concrete domain* (D). An abstract value is a finite representation of a, possibly infinite, set of actual values in the concrete domain (D). The set of all possible abstract semantic values represents an abstract domain D_α which is usually a complete lattice or cpo which is ascending chain finite.

The abstract domain $\langle D_\alpha, \sqsubseteq \rangle$ and the powerset of the concrete domain $\langle 2^D, \subseteq \rangle$ are related via a pair of monotonic mappings $\langle \alpha, \gamma \rangle$: *abstraction* $\alpha : 2^D \to D_\alpha$, and *concretization* $\gamma : D_\alpha \to 2^D$, such that

$$\forall x \in 2^D : \ \gamma(\alpha(x)) \supseteq x \quad \text{and} \quad \forall y \in D_\alpha : \ \alpha(\gamma(y)) = y.$$

Note that in general $\sqsubseteq$ is induced by $\subseteq$ and α (in such a way that $\forall \lambda, \lambda' \in D_\alpha : \ \lambda \sqsubseteq \lambda' \Leftrightarrow \gamma(\lambda) \subseteq \gamma(\lambda')$). Similarly, the operations of *least upper bound* ($\sqcup$) and *greatest lower bound* ($\sqcap$) mimic those of 2^D in some precise sense.

50 Specialized definitions

A distinguishing feature of our framework is that, as will be presented later, analysis for an atom A and an abstract substitution λ is performed w.r.t. a *specialized definition* of the A rather than the original definitions for it. In order to guarantee the correctness of the analysis framework, we need to demonstrate the semantical *equivalence* between the original and specialized definitions. The aim of Sect. 50.1 is to formalize the particular notion of equivalence necessary for this purpose. The process of creating specialized definitions can be viewed as a *transformation sequence* in which an original definition is transformed into another, equivalent one, by means of a series of transformations. Sect. 50.2 presents a number of semantics-preserving transformations used for the construction of specialized definitions. Finally, Sect. 50.3 introduces the notion of *specialized definition* and states its correctness.

50.1 Equivalence of Definitions

This section presents our notion of *equivalence* between two definitions for the same predicate. By a *definition* we mean a set of rules for the same predicate p such that p is one of the predicates in program P. First, we give a (non-standard) definition of the operational semantics of a set of rules in a program.

Definition 50.1[semantics of a rule] Let P be a program and $H \leftarrow B$ be a rule such that $pred(H)$ is defined in P. We define the semantics of $H \leftarrow B$ in P as follows:

$$[\![H \leftarrow B]\!]_P = \{H\theta \mid \exists \text{ a computed answer } \theta \text{ for } B \text{ in } P\}$$

The semantics of a definition $\{H_1 \leftarrow B_1, \ldots, H_n \leftarrow B_n\}$ with $n \geq 1$ is defined as the union of the semantics of the individual rules.

The notion of specialized definition is the subject of the next subsection. Nevertheless, we anticipate that a specialized definition is generated for a particular context which includes an atom A and an abstract substitution λ for A. This is to say that we specialize the definition of a predicate for its (restricted) use within the context of the pair $\langle A, \lambda \rangle$. Therefore, the specialized definition has to preserve the original semantics only for the context w.r.t. the specialization has been performed. A first step to formalize our notion of equivalence is to restrict the semantics of Def. 50.1 to atoms.

Definition 50.2[semantics of rules restricted to an atom] Let A be an atom such that $pred(A) = p$ and P be a program. Let $H \leftarrow B$ be a rule such that $pred(H) = p$. We define the semantics of $H \leftarrow B$ in P restricted to the atom A as follows:

$$[\![H \leftarrow B]\!]_P^A = \{A' \in [\![H \leftarrow B]\!]_P | \exists \theta \text{ with } A' = A\theta\}$$

The semantics of a definition $\{H_1 \leftarrow B_1, \ldots, H_n \leftarrow B_n\}$ with $n \geq 1$ restricted to an atom A is $\bigcup_{i=1\ldots n} [\![H_i \leftarrow B_i]\!]_P^A$.

Intuitively, the semantics of a set of rules restricted to an atom is formed by only those computed answers for the rules which unify with the atom (while the rest are discarded).

In our framework, the specialization context includes an additional parameter—the *abstract substitution*—which distinguishes our method from traditional specialization techniques. Informally, the abstract substitution allows expressing some properties in the selected abstract domain which can restrict further the specialization context. Thus, we now need to introduce the notion of semantics of rules restricted to the context of an atom *and* an abstract substitution.

Definition 50.3[semantics of rules restricted to an atom and abstract substitution] Let A be an atom and P be a program. Let λ be an abstract substitution for A. Let $H \leftarrow B$ be a rule such that $pred(H) = p$.We define the semantics of $H \leftarrow B$ in P restricted to A and λ as follows:

$$[\![H \leftarrow B]\!]_P^{\langle A,\lambda \rangle} = \{A' \in [\![H \leftarrow B]\!]_P^A | \exists \theta \in \gamma(\lambda) \text{ with } A' = A\theta\}$$

The semantics of a definition $\{H_1 \leftarrow B_1, \ldots, H_n \leftarrow B_n\}$ with $n \geq 1$ restricted to an atom A and abstract substitution λ is $\bigcup_{i=1\ldots n} [\![H_i \leftarrow B_i]\!]_P^{\langle A,\lambda \rangle}$.

Roughly speaking, we now select from the computed answers for the rules and atom (as stated in Def. 50.2) those which are compatible with the abstract substitution, i.e., the computed answers which satisfy the properties in λ.

Finally, the next definition provides a notion of *equivalence* between definitions. This equivalence can be restricted to an atom or to both an atom and an abstract substitution.

Definition 50.4[equivalent definitions] Let P be a program, A an atom and λ an abstract substitution. Let D and D' be two definitions for the same predicate $p = pred(A)$. We have the following notions of *equivalence* between D and D':

1. $D \equiv_P D'$ iff $[\![D]\!]_P = [\![D']\!]_P$.

2. $D \equiv_P^A D'$ iff $[\![D]\!]_P^A = [\![D']\!]_P^A$.

3. $D \equiv_P^{\langle A,\lambda\rangle} D'$ iff $[\![D]\!]_P^{\langle A,\lambda\rangle} = [\![D']\!]_P^{\langle A,\lambda\rangle}$.

50.2 Transformation Rules

As already mentioned, our goal is, given a definition D, to obtain another definition D' which is a specialized w.r.t. an atom A and an abstract description λ such that $D \equiv_P^{\langle A,\lambda\rangle} D'$ (and D' is in some sense preferred to D). In our setting, the initial definition, D_0, will be formed by the set of rules $Def(A, P)$. Then, we perform a series of transformation steps and construct the next definitions $\langle D_0, D_1, \ldots, D_k\rangle$ in the sequence. Each definition D_i is obtained from D_{i-1} by applying one of the rules listed below.

Definition 50.5[instantiation] Let D_k be a definition and let A be an atom. The result of applying *instantiation* to D_k w.r.t. A is

$$D_{k+1} = instantiation(D_k, A) = \{(H \leftarrow B)\theta \mid (H \leftarrow B) \in D_k \text{ and } \exists\theta = mgu(A, H)\}$$

Informally, instantiation returns the subset of rules in D_k whose head unifies with A and applies the mgu to them. Thus, instantiation filters out those rules which are not directly applicable for the given atom and further instantiates the remaining rules.

We now present the well-know unfolding transformation. As usual, by unfolding a rule w.r.t. an atom, we replace the rule by a set of new (unfolded) rules. In particular, as many new rules as clauses in P are applicable for the atom which is unfolded.

Definition 50.6[unfolding] Let D_k be a definition and let $R : (H \leftarrow A, G)$ be a rule in D_k where A is an atom and G a (possibly empty) conjunction of atoms. Let C be a variant of a rule in P such that $vars(R) \cap vars(C) = \{\}$ and the atoms $hd(C)$ and A are unifiable with mgu θ. The unfolding of R using C is the rule $(H \leftarrow bd(C), G)\theta$. Let $C_1, \ldots, C_n$ with $n \geq 1$ be the renamed apart variants of rules in P whose heads $hd(C_i)$ are unifiable with A.

The result of applying unfolding to D_k w.r.t rule $R \in D_k$ is

$$D_{k+1} = unfold(D_k, R) = D_k - \{R\} \cup \{U_1, \ldots, U_n\}.$$

where each U_i is the rule resulting from unfolding R using C_i.

Now we proceed to introduce a transformation which is able to exploit the information available in an abstract description. For this purpose, we apply the *partial concretization* of the abstract substitution to the definition. Given an abstract domain D_α, we say that a function $part\_conc : D_\alpha \rightarrow D$ is a *partial concretization* [192] iff $\forall \lambda \in D_\alpha \; \forall \theta' \in \gamma(\lambda) \; \exists \theta''$ s.t. $\theta' = part\_conc(\lambda)\theta''$.

Definition 50.7[instantiation with abstract information] Let D_k be a definition, let A be an atom, and let λ be an abstract substitution for A in D_α. Let $\theta = part\_conc(\lambda)$. The result of applying *instantiation_with_abs_info* to D_k w.r.t. A and λ is

$$D_{k+1} = instantiation\_with\_abs\_info(D_k, A, \lambda) = instantiation(D_k, A\theta)$$

All the transformations proposed are semantic preserving. Some of them preserve the semantics in general, and others are correct for the particular context described by the pair $\langle A, \lambda \rangle$. The next theorem states the correctness of the above transformations.

Theorem 50.8[correctness] Let A be an atom and P be a program. Let λ be an abstract substitution for A. Let $\langle D_0, \ldots, D_n \rangle$ be a transformation sequence obtained by applying the above transformations with D_0 the set of rules in $Def(G, P)$. Then, $D_0 \equiv_P^{\langle A, \lambda \rangle} D_n$.

Proof [sketch] All the transformation proposed are semantic preserving. In particular, unfolding preserves the semantics of definitions in general, whereas instantiation (resp. instantiation with abstract information) are correct for the particular context described by the atom A (resp. the pair $\langle A, \lambda \rangle$). In any case, after any number of transformations, the semantics is always preserved. □

50.3 The Specialization Strategy

Though all the transformation presented above preserve the semantics of definitions and generate definitions which are more specialized, we now propose a given order, or a *strategy*, in which such transformations should be performed in order to obtain the best possible specialization results.

Definition 50.9[specialization strategy] Let P be a program and $\leftarrow A$ be the initial atomic query. Let λ be an abstract description for A in a given domain D_α. Our *specialization strategy* computes a transformation sequence $\langle D_0, \ldots, D_n \rangle$ obtained as follows:

1. D_0 = $Def(A, P)$
2. D_1 = $instantiation(D_0, A)$
3. D_2 = instantiation_with_abs_info(D_1, A, λ)
4. the remaining $D_3, \ldots, D_n$ are generated by $n - 2$ *unfolding* transformations.

We use D = specialized_definition(A, λ, P) to denote that D is the result of specialization using the strategy presented above.

An interesting point to note is that neither the original program P nor the atom A are modified through the specialization process. The specialization can be seen as the generation of a new, additional, definition for $pred(A)$. Since the rules for the definition being specialized are kept separate from the rules in the original program, there is no problem in having additional rules for an already existing predicate. However, if we would like to have a program which contains both the original and specialized definitions for A together, the new definition can be safely added to the original program P by *renaming* both the head of the new clauses and the initial query A with a fresh predicate name. This guarantees that the semantics is preserved, since the new definition will only be used to resolve the initial query. In the following, we assume that function ren performs this renaming.

Theorem 50.10[correctness] Let A be an atom and P be a program. Let λ be an abstract substitution for A. Let D = specialized_definition(A, λ, P). Let $D' = ren(D)$. Then, $[\![A]\!]_P \equiv [\![ren(A)]\!]_{P \cup D'}$.

Proof [sketch] This result is a particular instance of Theorem 50.8. □

51 Abstract Interpretation with Specialized Definitions

In this section we present a generic analysis algorithm which is a modified version of that in [89]. In essence this analyzer produces a *program analysis graph* which can be viewed as a finite representation of the (possibly infinite) set of (possibly infinite) AND-OR trees explored by the concrete execution [13]. The graph has two sorts of nodes: those belonging to rules (also called "AND-nodes") and those belonging to atoms (also called "OR-nodes"). The rules are annotated by descriptions at each program point when the rule is executed from the calling pattern of the node connected to the rules. The program points are at the entry to the rule, the point between each two literals, and at the return from the call. Atoms in the rule body have arcs to OR-nodes with the corresponding calling pattern. If such a node is already in the tree it becomes a recursive call.

How this program analysis graph is constructed is detailed in Figure 19. This algorithm differs from the original algorithm mainly in that it analyzes *specialized definitions* rather than the original ones.

Intuitively, the analysis algorithm is just a graph traversal algorithm which places entries in an *answer table* and *dependency arc table* as new nodes and arcs in the program analysis graph are encountered.

- *Answer table:* The answer table contains entries of the form A : *CP* $\mapsto$ *AP*. A is an atom, *CP* is the calling pattern and AP is the answer pattern. Each entry in the answer table corresponds to an OR-node in the analysis graph of the form $\langle A$: *CP* $\mapsto$ *AP*$\rangle$. It is interpreted as the answer pattern for calls of the form *CP* to A is *AP*.

- *Dependency arc table:* A dependency arc is of the form H_k : $CP_0 \Rightarrow [CP_1]\ B_{k,i}$: CP_2. This is interpreted as follows: if the rule with H_k as head is called with description CP_0 then this causes literal $B_{k,i}$ to be called with description CP_2. The remaining part CP_1 is the program annotation just before $B_{k,i}$ is reached and contains information about all variables in rule k. CP_1 is not really necessary, but is included for efficiency. Dependency arcs represent the arcs in the program analysis graph from atoms in a rule body to an atom node.

The program analysis graph is implicitly represented in the algorithm simply by means of the answer table and the dependency arc table. In the sense that, given the information in these, it is straightforward to construct the graph and the associated program point annotations.

To capture the different graph traversal strategies used in different fixed-point algorithms, we use a *priority queue* which is the final structure used in our algorithm. It handles events of three forms:

```
analyze_sp_defs(S,P)
  foreach A : CP ∈ S
    add_event(newcall(A : CP))
  main_loop(P)

main_loop(P)
  while E := next_event()
    if (E = newcall(A : CP))
      new_calling_pattern(A : CP,P)
    elseif (E = updated(A : CP))
      add_dependent_rules(A : CP)
    elseif (E = arc(R))
      process_arc(R)
  endwhile
  remove_useless_calls(S)

new_calling_pattern(A : CP,P)
  P' := specialized_definition(A,CP,P)
  foreach rule A_k :- B_{k,1}, ..., B_{k,n_k} in P'
    CP' := Acalltoentry(A, CP, A_k)
    CP_0 :=
      Aextend(CP', vars(B_{k,1}, ..., B_{k,n_k}))
    CP_1 := Arestrict(CP_0, vars(B_{k,1}))
    add_event(arc(
      A_k : CP ⇒ [CP_0] B_{k,1} : CP_1))
  AP := initial_guess(A : CP)
  if (AP ≠ ⊥)
    add_event(updated(A : CP))
  add A : CP ↦ AP to answer table

add_dependent_rules(A : CP)
  foreach arc of the form
      H_k : CP_0 ⇒ [CP_1] B_{k,i} : CP_2
      in graph
    where there exists renaming σ
      s.t. A : CP = (B_{k,i} : CP_2)σ
    add_event(arc(
      H_k : CP_0 ⇒ [CP_1] B_{k,i} : CP_2))

process_arc(H_k : CP_0 ⇒ [CP_1] B_{k,i} : CP_2)
  if (B_{k,i} is not a unification)
    add H_k : CP_0 ⇒ [CP_1] B_{k,i} : CP_2
    to dependency arc table
  W := vars(A_k :- B_{k,1}, ..., B_{k,n_k})
  CP_3 := get_answer(B_{k,i} : CP_2, CP_1, W)
  if (CP_3 ≠ ⊥ and i ≠ n_k)
    CP_4 := Arestrict(CP_3, vars(B_{k,i+1}))
    add_event( arc(
      H_k : CP_0 ⇒ [CP_3] B_{k,i+1} : CP_4))
  elseif (CP_3 ≠ ⊥ and i = n_k)
    AP_1 := Arestrict(CP_3, vars(H_k))
    insert_answer_info(H : CP_0 ↦ AP_1)

get_answer(L : CP_2, CP_1, W)
  if (L is a unification t_1 = t_2)
    return Aunif(t_1, t_2, CP_1)
  else
    AP_0 := lookup_answer(L : CP_2)
    AP_1 := Aextend(AP_0, W)
    return Aconj(CP_1, AP_1)

lookup_answer(A : CP)
  if (there exists a renaming σ s.t.
    σ(A : CP) ↦ AP in answer table)
    return σ^{-1}(AP)
  else
    add_event(newcall(σ(A : CP)))
    where σ is a renaming s.t.
    σ(A) is in base form
    return ⊥

insert_answer_info(H : CP ↦ AP)
  AP_0 := lookup_answer(H : CP)
  AP_1 := Alub(AP, AP_0)
  if (AP_0 ≠ AP_1)
    add (H : CP ↦ AP_1) to answer table
    add_event(updated(H : CP))
```

Figure 19: Abstract Interpretation with Specialized Definitions.

- $newcall(A : CP)$ which indicates that a new calling pattern for atom A with description *CP* has been encountered.

- $arc(R)$ which indicates that the rule referred to in R needs to be (re)computed from the

position indicated.

- $updated(A : CP)$ which indicates that the answer description to calling pattern A with description *CP* has been changed.

The main procedure of the algorithm is `analyze_sp_defs`, which is defined in terms of six abstract operations on the description domain D_α of interest:

- $\mathsf{Acalltoentry}(\mathsf{A}_1, CP, \mathsf{A}_2)$ performs the abstract unification of A_1 and A_2 and returns the abstract description *CP* in terms of A_2;
- $\mathsf{Arestrict}(CP, \mathsf{V})$ performs the abstract restriction of a description *CP* to the set of variables in the set V, denoted $vars(V)$;
- $\mathsf{Aextend}(CP, \mathsf{V})$ extends the description *CP* to the variables in the set V;
- $\mathsf{Aunif}(\mathsf{t}_1, \mathsf{t}_2, CP)$ performs the abstract unification of terms t_1 and t_2 in the context of description *CP*;
- $\mathsf{Aconj}(CP_1, CP_2)$ performs the abstract conjunction of two descriptions;
- $\mathsf{Alub}(CP_1, CP_2)$ performs the abstract disjunction of two descriptions.

Apart from the parametric description domain-dependent functions, the algorithm has several other undefined functions. The functions add_event and next_event respectively add an event to the priority queue and return (and delete) the event of highest priority. When an event being added to the priority queue is already in the priority queue, a single event with the maximum of the priorities is kept in the queue. When an arc $H_k : CP \Rightarrow [CP'']B_{k,i} : CP'$ is added to the dependency arc table, it replaces any other arc of the form $H_k : CP \Rightarrow [\_]B_{k,i} : \_$ in the table and the priority queue. Similarly when an entry $H_k : CP \mapsto AP$ is added to the answer table, it replaces any entry of the form $H_k : CP \mapsto \_$. Note that the underscore (_) matches any description, and that there is at most one matching entry in the dependency arc table or answer table at any time.

The function initial_guess returns an initial guess for the answer to a new calling pattern. The default value is $\bot$ but if the calling pattern is more general than an already computed call then its current value may be returned.

The algorithm centers around the processing of events on the priority queue in main_loop, which repeatedly removes the highest priority event and calls the appropriate event-handling function. When all events are processed it calls remove_useless_calls. This procedure traverses the dependency graph given by the dependency arcs from the initial calling patterns S and marks

those entries in the dependency arc and answer table which are reachable. The remainder are removed. More details on the algorithm (without specialized definitions) can be found in [89].

As already mentioned, the main difference of our analysis algorithm of Fig. 19 w.r.t. the original algorithm of [89] is that we analyze the program by using the specialized rules computed by function specialized_definition of Def. 50.9 rather than the original rules.

51.1 Correctness

In this section we discuss whether the analysis results obtained by the proposed analysis framework are correct.

Theorem 51.1[correctness] Let P be a program and let $A : CP$ be an initial call pattern in S. Let $A : CP \mapsto AP$ be the answer pattern computed by the analysis algorithm in Fig. 19. Then, AP is a correct answer pattern for $A : CP$, i.e., $\gamma(A, AP) \supseteq [\![A]\!]_P^{(A,CP)}$.

Proof By Theorem 50.8, we know that specialized definitions always preserve the concrete semantics of the original one. As a result, since the analysis algorithm is guaranteed to obtain a safe approximation of the success set, the analysis obtained by analyzing a call $A : CP$ w.r.t. the specialized definition is guaranteed to be also a safe approximation of $[\![A]\!]_P^{(A,CP)}$.

□

In addition to obtaining results which are correct, we conjecture that the proposed framework allows obtaining more precise results than those achieved by traditional abstract interpretation algorithms. Several examples already show a gain in accuracy although the formal proof is a subject of further research.

52 Termination

In this section we recall the termination problems which appear in both abstract interpretation and partial deduction and then relate these problems in the context of the integrated framework we propose.

52.1 Termination in Abstract Interpretation

As it is well known, termination of traditional algorithms for abstract interpretation of logic programs [13, 89] is achieved by using abstract domains with certain characteristics and possibly the use of *widening* operators. More precisely, two termination problems can be considered:

A.1 the success computation problem: when computing an answer pattern for a call pattern $A : CP$, different tentative answer patterns $AP_0, AP_1, \ldots, AP_\omega$, with $AP_0 \sqsubseteq AP_1 \sqsubseteq \ldots \sqsubseteq AP_\omega$ can be computed until a fixpoint is reached.

A.2 the call computation problem: if analysis is context-sensitive and multivariant, several call patterns $\{A_1 : CP_1, \ldots, A_\omega : CP_\omega\}$ with $pred(A_1) = \ldots = pred(A_\omega) = p$ can be generated during analysis for the same predicate p.

Intuitively, A.1 is related to the complexity of computing the final answer pattern AP_ω for a given call pattern $A : CP$. Problem A.2 is related to keeping finite the number of call patterns which are analyzed, i.e., the answer table must be finite. Termination w.r.t. A.1 is guaranteed by using abstract domains without infinite ascending chains or by the use of widening operators.

Definition 52.1[widening] We say that an operator ∇ is a widening iff for any increasing chain $a_0 \subset a_1 \subset a_2 \subset ...$ then chain $b_0 = a_0 \nabla a_1, ..., b_{i+1} = b_i \nabla a_{i+1}, ...$ is not strictly increasing for $\subseteq$, that is, it should be a stationary sequence.

In addition to being used for ensuring termination, widening operators can be used in abstract domains with finite ascending chains to accelerate convergence.

Termination w.r.t. A.2 does not represent a problem if the analysis algorithm is context-insensitive or monovariant. However, if the domain is infinite, the ascending chain finite condition is not enough for guaranteeing termination w.r.t A.2. In this situation, some way to limit multivariance is needed in order to guarantee termination.

Let us discuss this problem in more detail. Given a predicate p, in order to guarantee termination w.r.t. A.2 it is required that the number of call patterns of the form $A : CP$ such that *pred(A)=p* which are handled by analysis, i.e., for which an entry in the answer table is computed must be finite. It is important to observe that, each atom A in a program encodes a concrete substitution θ defined as *θ=Inst(A)=mgu(A,base_form(A))*. Thus, each call pattern $A : CP$ can be seen as a triple $\langle base\_form(A), Inst(A), CP \rangle$. As a result, the number of combinations of concrete and abstract substitutions for which we would like to analyze a program procedure must be finite. In traditional abstract interpretation this is obtained by first fixing the maximum number of concrete substitutions for which a procedure can be analyzed and then by using some multivariance control strategy which guarantees that the number of abstract substitutions for which a concrete atom can be analyzed always remains finite.

Many frameworks for abstract interpretation of logic programs *normalize* programs prior to analysis. A program is normalized if all atoms only contain distinct variables. This is in general not restrictive since all logic programs can be normalized. A normalized representation allows

simplifying both the formalization and the implementation of the algorithm. In particular it limits the number of concrete substitutions for which a procedure can be analyzed to just one per procedure.

Other analysis frameworks, such as the one we propose[32], do not require programs to be normalized. For this, an additional abstract operation, *Acalltoentry* in our case, has to be added to the algorithm. This design decision has several consequences: it augments the multi-variance level of the analysis since calls which correspond to different concrete atoms will be analyzed separately. This will have an impact both on the accuracy and the efficiency of the analysis. On the accuracy side, more accurate results will be obtained since it allows eliminating from $Def(A, P)$ those rules whose head unifies with *base_form(A)* but does not unify with the atom A to be analyzed. On the efficiency side, more call patterns will have to be analyzed, which means that more analysis time will be required. Note, however, that even in this scenario, the set of concrete substitutions for which a procedure can be analyzed is also fixed (and finite) since it is limited to the atoms which explicitly appear in the program to be analyzed. Thus, in both cases (normalized and unnormalized programs) termination w.r.t. A.2 is guaranteed by the use of a multi-variance control strategy which guarantees that the set of abstract substitutions $\{CP_1, \ldots, CP_\omega\}$ for a given concrete atom A is finite.

52.2 Termination in Program Specialization

Termination of program specialization is often split in two levels:

S.1 the so-called *local termination:* this is the problem of ensuring that a finite number of unfolding steps are performed for a given initial call pattern $A : CP$.

S.2 and the *global termination:* in this case, we have to ensure that the number of atoms $A : CP$ for which a specialized definition is to be computed remains finite.

The topic of local and global termination has received considerable attention in the partial deduction community. It is not the goal of this work to present a thorough study of local and global control strategies. We present however in Section 54 the different control strategies currently available in our implementation of the framework.

Definition 52.2[generalize] A function *generalize*:$Atoms \times 2^{Atoms} \rightarrow Atoms$ is any function such that for any atom A and set of atoms $\mathcal{A}$ *generalize(A,*$\mathcal{A}$*)* $\supseteq A$.

[32]One important difference between the algorithm herein presented and that in [89] is that the current algorithm allows analyzing programs which are not *normalized*

In other words, the function *generalize* returns a generalization of atom A. The more information *generalize* loses, the faster global termination will be achieved. However, the more information is lost the less productive specialization will in principle be. On the other extreme, the identity function is trivially a correct *generalize* function which loses no information. However, termination is not guaranteed and more conservative functions will be used in practice.

52.3 Termination in the Integrated Framework

Since our framework performs both specialization and abstract interpretation, it is natural that the four termination problems mentioned above appear in this context.

Problem S.1 appears because the algorithm now contains an additional phase which is that of specializing definitions and it corresponds to guaranteeing that the execution of *specialized_definition* terminates. Note that it may often be the case that an infinite number of unfolding steps may be performed.

Intuitively, problem S.2 appears because the program to be analyzed during abstract interpretation with specialized definitions is not fixed, but rather is dynamically generated during analysis. Since the process of specializing definitions may introduce new concrete bindings, the assumption that the number of concrete atoms per predicate is fixed no longer holds.

Note that clearly, problem (S.2) is indeed very related to the problem (A.2). S.2 is solved in program specializers by keeping the number of concrete substitutions per predicate finite, by means of the application of a *generalize* operation during global control. A.2 is solved in abstract interpretation by first fixing a set of atoms and then keeping the number of abstract substitutions per atom each atom finite by means of multi-variance control. As a result both a terminating global control strategy and a terminating multi-variance control strategy are required in order to guarantee termination of our integrated approach.

Even if local termination is guaranteed, for example by applying zero unfolding transformations, global termination (S.2) is threatened as soon as we apply of *instantiation*, *instantiation_with_abs_subs*, and *unfolding*.

Note that termination of abstract interpretation is guaranteed w.r.t. both S.1 and S.2 since no unfolding steps are performed and instantiation nor partial concretization are applied, i.e., the atom A is always analyzed w.r.t. $D_0 = Def(A, P)$.

In order to guarantee termination in `analyze_sp_defs` while still allowing performing any of the transformations presented in Section 50.2, we need to introduce the possibility of using a global control strategy which will abstract away part of the concrete information in an atom before applying *instantiation*.

For this we have to augment the algorithm `analyze_sp_defs` in several ways. We will

```
spec_def_glob_control(A,CP,P)
   (A' : CP') := generalize(A,CP)
   add (A : CP) ⇝ (A' : CP') to generalization table
   if (there exists a renaming σ s.t.
       σ(A : CP) ↦ SD in specialization table)
       return SD
   else
       SD := specialized_definition(A',CP',P)
       add (A' : CP') ↦ SD to specialization table
       return SD
```

Figure 20: Adding global control

add two more global data structures:

- *Specialization table:* it contains entries of the form $A : CP \mapsto D$, where $A : CP$ is a call pattern and D is a definition for $pred(A)$. It should be interpreted as D is the specialized definition which has been obtained by specialization w.r.t. $A : CP$.

- *Generalization table:* it contains entries of the form $A : CP \leadsto A' : CP'$. It should be interpreted as: the call pattern $A : CP$ is analyzed w.r.t. a definition of $pred(A)$ which has been specialized w.r.t. $A' : CP'$. Correctness of analysis requires that $(A : CP) \sqsubseteq (A' : CP')$.

The specialization table is useful in two ways. The most obvious one is to record the set of call patterns for which a specialized definition has already been computed. This is exactly the role usually played by the set of atoms during global control of partial deduction. The second one is more a practical reason: since several call patterns may share the same specialized definition, it can be a good idea to store the result of specialization. Also, in contrast to partial deduction, abstract interpretation often has to iterate and process body clauses several times until a fixpoint is reached, thus storing the specialized clauses is often a good idea.

The generalization table actually stores the results of generalization obtained up to the present moment. In contrast to the specialization table, it is not actually required for termination of the algorithm. However, this table together with the dependency arc table allow implementing an efficient and accurate code generation scheme which is strictly more accurate than that used in partial deduction.

Figure 20 shows the definition of the function `spec_def_glob_control` which specializes definitions while performing global control. Note that this function should be called instead of `specialized_definitions` in algorithm `analyze_sp_defs`. Both the specialization

table and the generalization table are global arguments. The specialization table is implicitly used by the `generalize` function.

53 The Framework as a Specializer

As already mentioned, the integrated framework we propose has applications both in program analysis and specialization. In fact, it is natural to consider the possibility of using the specialized definitions which were generated during the execution of `analyze_sp_defs`(S, P) rather than the original program P.

Since when abstract interpretation terminates the set of call patterns analyzed is guaranteed to be covered, it is possible to use the rules which correspond to specialized definitions and throw away the original program altogether.

As usually done in partial deduction and also in abstract specialization, we will use different names for each different specialized version of each predicate. This will make it possible to have a multiply specialized program without introducing run-time tests to select among the different implementations for the predicate. This will also guarantee the independence condition among atoms usually required in partial deduction.

Given that we will rename program rules, the main difficulty now is to also rename calls in body atoms so that the corresponding version is used. In partial deduction, deciding which is the correct version to use is often based on the abstraction operator used during the specialization phase. In our case, the dependency arc table together with the generalization table can directly be used in order to determine precisely which is the version to be used in each literal of each rule of the specialized program.

There are other interesting question to take into account. One is that the results of analysis, i.e., the answer table may contain entries which correspond to *spurious* call pattern. These corresponds to tentative call patterns which are not really used in the final analysis graph. In our algorithm, the spurious call patterns are removed right after reaching a fixpoint by the `remove_useless_calls` operation. Another important thing to mention is that the use of a generalization operation allows using the same specialized definition for different call patterns. This will help to reduce the size of the final program.

In spite of this, more powerful techniques for minimizing the size of the final program could be used. We are in fact investigating the possibility of extending the minimization algorithm already used in abstract specialization [189] for its appliaction in this context.

54 System Description

CiaoPP[88] is the abstract interpretation-based preprocessor of the Ciaomulti-paradigm constraint logic programming system. It uses modular, incremental abstract interpretation as a fundamental tool to obtain information about the program. A version of the analysis engine existing in CiaoPPhas been extended in order to cope with specialized definitions, as explained throughout the paper. The new framework is fully integrated into the latest distribution of the CiaoPPsystem. This section shows the different options of a typical session with the new extension of the analyzer. First, CiaoPPis started by loading the library ciaopp in a Ciaoshell[33].

```
use_module(library(ciaopp)).
```

There are several fixpoint algorithms coexisting in the CiaoPPsystem. Our contribution has been integrated with the so-called "di" fixpoint, which corresponds to the *depth independent* fixpoint algorithm described in [187]. It is selected by setting up the flag fixpoint to the value di as follows.

```
set_pp_flag(fixpoint,di).
```

Then, we have a good number of options for controlling the local and global levels in the specialization process. We also have the possibility of applying the partial concretization of the abstract properties to the concrete atoms (see Sect. 50). Finally, we show the instructions to generate the code of the specialized program. The next sections discuss these options in more detail.

54.1 Local Control

In order to ensure the local termination of the algorithm, we must incorporate some mechanism to stop the construction of the unfolding process. For this purpose, there exist several well-known techniques in the literature, e.g., depth-bounds, loop-checks [9], well-founded orderings [15], well-quasi orderings [202], etc. We have incorporated an unfolding rule in CiaoPPwhich can be controlled by three different strategies:

- off: corresponds to not computing specialized definitions.
- inst: instantiation is performed, but no unfolding steps take place.
- det: this strategy allows the expansion of unfolding while derivations are deterministic and stops them when a non-deterministic branch is required;

[33] More detailed information on CiaoPPcan be found in [88].

- `emb`: the non-embedding unfolding rule, which uses the homeomorphic embedding ordering to stop the unfolding process;

The desired strategy can be selected by setting the `local_control` flag:

```
set_pp_flag(local_control,strategy).
```

where `strategy` corresponds to `emb`, `det`, `inst`, or `off` as explained above. Notice that the strategy `det` is non-terminating but it is included for efficiency purposes.

The selection of local control is necessary in order to perform analysis with specialized definitions. If local control is turned off, we just have the standard analysis regardless of the options selected for global control.

54.2 Global Control

As a result of the specialization performed at the local control level, new patterns are produced and subject to be analysed. The global control is constrained to continue iteratively with the analysis of those patterns which are not *covered* yet by the previously analyzed patterns. Since this process can be infinite, we include some strategies to improve (and in some cases ensure) the termination behavior of analysis:

- `off`: this is equivalent to not using a *generalize* function, i.e., we specialize all call patters which we receive.
- `id`: this strategy allows specializing a new pattern provided it is not equal (modulo renaming) to a formerly analyzed one.
- `inst`: this strategy allows us to specialize a new pattern if it is not an instance of a previous pattern.
- `hom_emb`: the nonembedding abstraction operator uses the homeomorphic embedding ordering to detect when a pattern is covered and, thus, stop the iterative process;

The desired strategy can be selected by setting the corresponding flag:

```
set_pp_flag(global_control,strategy).
```

where `strategy` can be one of the above options. Out of the four strategies, only `hom_emb` always ensures the termination of the process. Although the `id` and `inst` strategies are more efficient and terminating in many practical cases.

54.3 Instantiation w.r.t. Abstract Information

We claim that more specialization can be achieved in some cases by applying *instantiation_with_abs_info* instead of *instantiation* in order to specialize a definition. In order to be able to select between those two possibilities, we can use the part_conc flag as follows.

```
set_pp_flag(part_conc,on).
```

The flag can be deactivated with the value `off` at any time.

54.4 Code Generation

Once the different settings have been selected, we can load the program subject to be specialized (the initial data is written by means of an `entry` declaration in the same file where the program resides):

```
module(app).
```

Then, the analysis is started with the desired domain (e.g., the `eterms` domain):

```
analyze(eterms).
```

In order to generate a specialized program from the analysis results, we have to perform the so-called `codegen` transformation:

```
transform(codegen).
```

The specialized program can be written in a file (e.g., the `output_file`) by calling the predicate `output` as follows.

```
output(output_file).
```

55 A Running Example

Example 55.1

Let us consider the program in Fig. 55.1 which generates lists of "valid" numbers, i.e., lists formed by a certain combination of `1`'s and `2`'s. Predicate `q` generates a non-empty list made up of `2`'s and ending with `1`. The intermediate predicate `p` is used to obtain a value indicating whether this list is empty (returns `0`) or contains at least one number (returns `1`). Then, a call to `app` concatenates the constant list `[1,2,1]` with the one generated by `q`. Finally, the test `validlist` checks whether the resulting list is of type "`validlist`" which represents a

```
myapp(Res):- L1=[L2],q(L),p(L,L2),app([1,2|L1],L,Res),validlist(Res).

q([1]).
q([2|Xs]):- q(Xs).

p(X,Y):- r(X,Y).

r([],0).
r([_|_],1).

app([],Y,Y):- validlist(Y).
app([X|Xs],Y,[X|Zs]):-app(Xs,Y,Zs).

:- regtype validlist/1.

validlist([]).
validlist([X|Xs]):- valid(X),
validlist(Xs).

:- regtype valid/1.

valid(1).
valid(2).
```

Figure 21: Running Example

`valid` list of `2`'s and `1`'s. The definitions of these two regular types are declared by means of a `regtype` declaration, following `CiaoPP`syntax.

Now, we proceed to specialize the above program without any input data (neither concrete nor abstract) in `CiaoPP`by using the `eterms` domain. During this process, the specialized definitions are generated by using the deterministic local control and applying partial concretization (see Sect. 54). Let us see as example the generation of the specialized rule for the call `p(L,L2)` in the definition of predicate `myapp`. After analyzing the first two atoms, we obtain the abstract description `rt0(L1), rt6(L)` where the new regular types are defined like[34]

```
:- regtype rt6/1.
rt6([A|B]) :- valid(A),validlist(B).
```

[34]We refer to [214] for a detailed description of analysis using `eterms`. It is outside the scope of this work.

```
:- regtype rt0/1.
rt0([A]) :- term(A).
```

If we proceed to generate a specialized definition for the next atom `p(L,L2)`, we first apply the partial concretization transformation of the previous description w.r.t. the atom, which gives `p([A],L2)`. Now, we instantiate the rule defining `p` with this atom and obtain `p([A],Y):- r([A],Y)`. It is unfolded by using the second rule defining `r` and we get the specialized rule `p([A],1)` for the atom.

Similarly, the specialization of the different atoms has been performed. By analyzing their specialized definitions, the resulting program is:

```
myapp(A) :-q_1(B),B=[_3|_4],A=[1,2,1,_3|_4].


q_1([1]).
q_1([2|A]) :-q_1(A).
```

Note that in order to obtain this specialized program it is required: (1) to use an abstract domain which captures regular types, (2) to compute approximations of success substitutions, (3) to perform aggressive unfolding, (4) to be able to eliminate rules which are incompatible with a given abstract call pattern, and (5) to be able to abstractly execute calls to predicates which are regular types.

Traditional partial deduction would not be able to optimize this program very much. It can perform (3), but none of the four other requirements. Abstract interpretation without specialized definitions would not be able to infer accurate enough information so as to detect that the `validlist` tests are redundant. Abstract specialization cannot perform (3) and thus would not be able to fully optimize the program.

Note that having all this features simultaneously in the framework further improves the results obtained by the individual techniques. Having (1) and (2) allow performing instantiation with abstract information which improves the results of unfolding or `p`. This in turn may generate new bindings which may improve the analysis results, and so on. Thus, the fine-grained integration we propose allows improving simultaneously the benefits of program analysis and specialization.

Another example of the power of the combination is the optimization of the call to `app`. The system infers that the first list contains three constant elements and an unknown one. After four unfoldings its definition disappears. Also, the test `validlist` is abstractly executed by our system the list resulting from the concatenation since it is inferred from `q_1` that `B` is a valid list. Clearly, the first three elements also satisfy the test.

56 Conclusions

Several works have witnessed the need of unifying the techniques of abstract interpretation and partial evaluation in a single framework able to obtain highly specialized programs. There has been a parallel development of frameworks which integrate notions of abstract interpretation in a partial evaluation algorithm [124, 70] and others which incorporate a code generation phase within an abstract interpreter [191, 192]. However, we are not aware of any practical algorithm able to combine and improve the power of the individual techniques.

The first approach establishes the conditions that the operations of abstract unfolding and abstract resolution must fulfill in order to have a correct framework, but does not give an actual algorithm. In the latter approach, it is specified how to generate code from the program and the analysis results but not specify how to perform unfolding. Indeed, [192] points out the need of integrating an unfolding rule in the abstract interpreter for the purpose of specialization and mentions several possibilities for doing it.

Our work studies an efficient and practical way of interleaving unfolding and abstract interpretation so that we unify the advantages of the individual techniques. In particular, we present an on-line, specialization algorithm for logic programs, whose behavior is parametric w.r.t. the local control strategy ρ, the generalization operator `generalize` and the abstract domain D_α (together with a widening operator when needed). In particular, the analysis algorithm can be formulated as:

$$\texttt{analyze}(\texttt{D}_\alpha, \nabla_\alpha, \texttt{local\_control}(\rho), \texttt{generalize})$$

Useful instances of this generic algorithm can be easily defined by instantiating the above four parameters. In this work, we have considered the case of partial evaluation which usually depends only on the unfolding rule and the generalization operator. Then, it happens that:

$$\texttt{analyze}(\texttt{D}'_\alpha, \emptyset, \texttt{local\_control}(\texttt{inst} + \texttt{unfold}), \texttt{generalize})$$

is a PE procedure if the domain D'_α assigns $\top$ to all terms and no widening operator is used. Thus, by using more refined abstract domains, it has been shown that our algorithm is a reasonable improvement over pure PE. Another instance of the algorithm is obtained by considering the `off` local control rule. In such case, there is no need for a global control rule. We can use the `Id` value, for example. Thus, the algorithm $\texttt{analyze}(\texttt{D}_\alpha, \nabla_\alpha, \texttt{local\_control}(\texttt{off}), \texttt{Id})$ corresponds to the abstract interpreter of [89]. Some examples demonstrate that by considering advanced unfolding rules and abstraction operators we can increase the accuracy of [89]. Consequently, we think that our method can give support and induce new research in hybrid approaches to specialization.

Acknowledgements

The authors would like to thank Maurice Bruynooghe for the valuable feedback he provided on the ACM TOPLAS paper, as well as the anonymous referees of such a paper, whose detailed comments and constructive criticisms have substantially improved the article.

The authors also greatly benefited from discussions with Danny De Schreye, Stefan Gruner, Neil Jones, Jesper Jørgensen, Helko Lehmann, Bern Martens, Torben Mogensen, Jens-Peter Secher, Morten Heine Sørensen, and comments of anonymous referees of JICSLP'98.

Thanks to Francisco Bueno and Pedro López for their help in the implementation of the tools herein presented, to M. García de la Banda, P J. Stuckey, and K. Marriott who have participated in the development of some of the applications presented, and to Claudio Vaucheret for his implementation of the *eterms* domain.

References

[1] P. A. Abdulla, K. Čerāns, B. Jonsson, and Y.-K. Tsay. General Decidability Theorems for Infinite-state Systems. In *11th IEEE Symposium on Logic in Computer Science*, pages 313–321, 1996.

[2] E. Albert, M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Improving control in functional logic program specialization. In G. Levi, editor, Static Analysis. *Proceedings of SAS'98*, LNCS 1503, pages 262–277, Pisa, Italy, September 1998. Springer-Verlag.

[3] M. Alpuente, M. Falaschi, and G. Vidal. Partial Evaluation of Functional Logic Programs. *ACM Transactions on Programming Languages and Systems*, 20(4):768–844, 1998.

[4] K. R. Apt. Introduction to logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 10, pages 495–574. North-Holland Amsterdam, 1990.

[5] K. R. Apt and E.-R. Olderog. *Verification of Sequencial and Concurrent Programs*. Springer-Verlag, 1991.

[6] K. Benkerimi and P. M. Hill. Supporting transformations for the partial evaluation of logic programs. *Journal of Logic and Computation*, 3(5):469–486, October 1993.

[7] F. Benoy and A. King. Inferring argument size relations in CLP($\mathcal{R}$). In *Proceedings of the 6th International Workshop on Logic Program Synthesis and Transformation*, pages 204–223, Sweden, 1996. Springer-Verlag, LNCS 1207.

[8] R. Bol. Loop checking in partial deduction. *The Journal of Logic Programming*, 16(1&2):25–46, 1993.

[9] R. Bol. Loop Checking in Partial Deduction. *Journal of Logic Programming*, 16(1&2):25–46, 1993.

[10] D. Boulanger and M. Bruynooghe. Deriving fold/unfold transformations of logic programs using extended OLDT-based abstract interpretation. *Journal of Symbolic Computation*, 15(5&6):495–521, 1993.

[11] D. Boulanger and M. Bruynooghe. A systematic construction of abstract domains. In B. Le Charlier, editor, *Proc. First International Static Analysis Symposium, SAS'94*, volume 864 of *Springer-Verlag Lecture Notes in Computer Science*, pages 61–77, 1994.

[12] D. Boulanger, M. Bruynooghe, and M. Denecker. Abstracting s-semantics using a model-theoretic approach. In M. Hermenegildo and J. Penjam, editors, *Proc. 6$^{\text{th}}$ International Symposium on Programming Language Implementation and Logic Programming, PLILP'94*, volume 844 of *Springer-Verlag Lecture Notes in Computer Science*, pages 432–446, 1994.

[13] M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10:91–124, 1991.

[14] M. Bruynooghe. A practical framework for the abstract interpretation of logic programs. *The Journal of Logic Programming*, 10:91–124, 1991.

[15] M. Bruynooghe, D. De Schreye, and B. Martens. A General Criterion for Avoiding Infinite Unfolding. *New Generation Computing*, 11(1):47–79, 1992.

[16] M. Bruynooghe, D. De Schreye, and B. Martens. A general criterion for avoiding infinite unfolding during partial deduction. *New Generation Computing*, 11(1):47–79, 1992.

[17] F. Bueno, D. Cabeza, M. Hermenegildo, and G. Puebla. Global Analysis of Standard Prolog Programs. In *European Symposium on Programming*, number 1058 in LNCS, pages 108–124, Sweden, April 1996. Springer-Verlag.

[18] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: a heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253, 1993.

[19] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.

[20] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.

[21] W. Charatonik and A. Podelski. Directional type inference for logic programs. In G. Levi, editor, *Proceedings of the International Symposium on Static Analysis (SAS'98), Pisa, September 14 - 16, 1998*, volume 1503 of *Springer LNCS*, pages 278–294. Springer-Verlag, 1998.

[22] W. Charatonik, A. Podelski, and J.-M. Talbot. Paths vs. Tress in Set-based Program Analysis. In *Principles of Programming Languages*, pages 330–338. ACM Press, January 2000.

[23] B. L. Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. *ACM Transactions on Programming Languages and Systems*, 16(1):35–101, 1994.

[24] W.-N. Chin and S.-C. Khoo. Tupling functions with multiple recursion parameters. In *Proceedings of the Third International Workshop on Static Analysis*, number 724 in LNCS 724, pages 124–140, Padova, Italy, Sept. 1993. Springer-Verlag.

[25] K. Clark. Predicate logic as a computational formalism. Technical Report DOC 79/59, Imperial College, London, Department of Computing, 1979.

[26] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[27] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 2000.

[28] E. M. Clarke and J. M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, Dec. 1996.

[29] M. Comini and M. C. Meo. Compositionality properties of sld-derivations. *Theoretical Computer Science*, 211(1 & 2):275–309, Jan. 1999.

[30] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications*. http://www.grappa.univ-lille3.fr/tata, 1999.

[31] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *Proceedings of ACM Symposium on Principles of Programming Languages (POPL'93)*, pages 493–501, Charleston, South Carolina, January 1993. ACM Press.

[32] C. Consel and O. Danvy. Tutorial Notes on Partial Evaluation. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages POPL'93*, pages 493–501, Charleston, South Carolina, 1993. ACM.

[33] C. Consel and S. C. Khoo. Parameterized partial evaluation. *ACM Transactions on Programming Languages and Systems*, 15(3):463–493, 1993.

[34] C. Consel and S. Koo. Parameterized partial deduction. *ACM Transactions on Programming Languages and Systems*, 15(3):463–493, July 1993.

[35] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[36] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *The Journal of Logic Programming*, 13(2 & 3):103–179, 1992.

[37] P. Cousot and R. Cousot. Systematic Design of Program Transformation Frameworks by Abstract Interpretation. In *POPL'02: 29ST ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 178–190, Portland, Oregon, January 2002. ACM.

[38] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the Conference Record of the 5th ACM Symposium on Principles of Programming Languages*, pages 84–97, Albuquerque, New Mexico, 1978.

[39] P. Dart and J. Zobel. A Regular Type Language for Logic Programs. In *Types in Logic Programming*, pages 157–187. MIT Press, 1992.

[40] M. G. de la Banda, K. Marriott, and P. Stuckey. Efficient Analysis of Constraint Logic Programs with Dynamic Scheduling. In *1995 International Logic Programming Symposium*, pages 417–431, Portland, Oregon, December 1995. MIT Press, Cambridge, MA.

[41] D. De Schreye and S. Decorte. Termination of logic programs: The never ending story. *The Journal of Logic Programming*, 19 & 20:199–260, May 1994.

[42] D. De Schreye, R. Glück, J. Jørgensen, M. Leuschel, B. Martens, and M. H. Sørensen. Conjunctive partial deduction: Foundations, control, algorithms and experiments. *Journal of Logic Programming*, 41(2 & 3):231–277, November 1999.

[43] D. A. de Waal and J. Gallagher. Specialisation of a unification algorithm. In T. Clement and K.-K. Lau, editors, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'91*, pages 205–220, Manchester, UK, 1991.

[44] D. A. de Waal and J. Gallagher. The applicability of logic program analysis and transformation to theorem proving. In A. Bundy, editor, *Automated Deduction—CADE-12*, pages 207–221. Springer-Verlag, 1994.

[45] S. Debray and D. Warren. Automatic mode inference for logic programs. *Journal of Logic Programming*, 5(3):207–229, 1988.

[46] G. Delzanno and A. Podelski. Model Checking in CLP. In W. R. Cleaveland, editor, *5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 223–239. Springer-Verlag, LNCS 1579, 1999.

[47] K. Doets. Levationis laus. *Journal of Logic and Computation*, 3(5):487–516, 1993.

[48] A. Finkel. The minimal coverability graph for Petri nets. *Lecture Notes in Computer Science*, 674:210–243, 1993.

[49] A. Finkel and P. Schnoebelen. Fundamental Structures in Well-structured Infinite Transition Systems. In *Proceedings of LATIN'98*, volume 1380 of *LNCS*, pages 102–118. Springer-Verlag, 1998.

[50] A. Finkel and P. Schnoebelen. Well-structured Transition Systems everywhere! *Theoretical Computer Science*, 1999. To appear.

[51] F. Fioravanti, A. Pettorossi, and M. Proietti. Automated strategies for specialising constraint logic programs. In K.-K. Lau, editor, *10th International Workshop on Logic-based Program Synthesis and Transformation*, pages 125–146. Springer-Verlag, LNCS 2042, 2000.

[52] F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying CTL properties of infinite state systems by specializing constraint logic programs. Technical Report DSSE-TR-2001-3, Department of Electronics and Computer Science, University of Southampton, 2001. Proceedings of the Second International Workshop on Verification and Computational Logic (VCL'01).

[53] T. Früwirth, E. Shapiro, M. Vardi, and E. Yardeni. Logic programs as types for logic programs. In *Proc. LICS'91*, pages 300–309, 1991.

[54] Y. Futamura, K. Nogi, and A. Takano. Essence of generalized partial computation. *Theoretical Computer Science*, 90(1):61–79, 1991.

[55] J. Gallagher. A system for specialising logic programs. Technical Report TR-91-32, University of Bristol, November 1991.

[56] J. Gallagher. Static Analysis for Logic Program Specialization. In *Workshop on Static Analysis WSA'92*, pages 285–294, 1992.

[57] J. Gallagher. Tutorial on specialisation of logic programs. In *Proceedings of PEPM'93, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–98. ACM Press, 1993.

[58] J. Gallagher, D. Boulanger, and H. Sağlam. Practical model-based static analysis for definite logic programs. In J. W. Lloyd, editor, *Proc. of International Logic Programming Symposium*, pages 351–365, 1995.

[59] J. Gallagher and M. Bruynooghe. Some low-level transformations for logic programs. In M. Bruynooghe, editor, *Proceedings of Meta90 Workshop on Meta Programming in Logic*, pages 229–244, Leuven, Belgium, 1990.

[60] J. Gallagher and M. Bruynooghe. The derivation of an algorithm for program specialisation. *New Generation Computing*, 9(3 & 4):305–333, 1991.

[61] J. Gallagher and M. Bruynooghe. The derivation of an algorithm for program specialisation. *New Generation Computing*, 9(1991):305–333, 1991.

[62] J. Gallagher, M. Codish, and E. Shapiro. Specialisation of Prolog and FCP Programs Using Abstract Interpretation. *New Generation Computing*, 6(2–3):159–186, 1988.

[63] J. Gallagher, M. Codish, and E. Shapiro. Specialisation of Prolog and FCP programs using abstract interpretation. *New Generation Computing*, 6:159–186, 1988.

[64] J. Gallagher and D. de Waal. Deletion of redundant unary type predicates from logic programs. In K. Lau and T. Clement, editors, *Logic Program Synthesis and Transformation*, Workshops in Computing, pages 151–167. Springer-Verlag, 1993.

[65] J. Gallagher and D. de Waal. Fast and precise regular approximations of logic programs. In P. Van Hentenryck, editor, *Proc. of the 11th International Conference on Logic Programming*, pages 599–613. MIT Press, 1994.

[66] J. Gallagher and D. A. de Waal. Deletion of redundant unary type predicates from logic programs. In K.-K. Lau and T. Clement, editors, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'92*, pages 151–167, Manchester, UK, 1992.

[67] J. Gallagher and D. A. de Waal. Fast and precise regular approximations of logic programs. In P. Van Hentenryck, editor, *Proceedings of the Eleventh International Conference on Logic Programming*, pages 599–613. The MIT Press, 1994.

[68] J. Gallagher and L. Lafave. Regular approximation of computation paths in logic and functional languages. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation*, volume 1110, pages 115 – 136. Springer Verlag Lecture Notes in Computer Science, 1996.

[69] J. Gallagher and L. Lafave. Regular approximations of computation paths in logic and functional languages. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, International Seminar*, LNCS 1110, pages 115–136, Schloß Dagstuhl, 1996. Springer-Verlag.

[70] J. Gallagher and J. Peralta. Using regular approximations for generalisation during partial evaluation. In *Proc. of the SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, pages 44–51. ACM Press, 2000.

[71] J. Gallagher and G. Puebla. Abstract Interpretation over Non-Deterministic Finite Tree Automata for Set-Based Analysis of Logic Programs. In *Fourth International Symposium on Practical Aspects of Declarative Languages*, number 2257 in LNCS, pages 243–261. Springer-Verlag, January 2002.

[72] J. P. Gallagher and L. Lafave. Regular approximation of computation paths in logic and functional languages. In O. Danvy, R. Glück, and P. Thiemman, editors, *Partial Evaluation*, pages 115–136. Springer-Verlag, LNCS 1110, 1996.

[73] J. P. Gallagher and J. C. Peralta. Using regular approximations for generalisation during partial evaluation. In J. Lawall, editor, *Proceedings of PEPM'00*, pages 44–51. ACM Press, 2000.

[74] J. P. Gallagher and J. C. Peralta. Regular tree languages as an abstract domain in program specialisation. *Higher Order and Symbolic Computation*, 14(2–3):143–172, November 2001.

[75] J. P. Gallagher and J. C. Peralta. Regular tree languages as an abstract domain in program specialisation. *Higher-Order and Symbolic Computation*, 14(2-3):143–172, 2001.

[76] J. P. Gallagher and G. Puebla. Abstract interpretation over non-deterministic finite tree automata for set-based analysis of logic programs. In S. Krishnamurthi and C. R. Ramakrishnan, editors, *Practical Aspects of Declarative Languages, 4th International Symposium, PADL 2002, Portland, OR, USA, LNCS Vol. 2257*, pages 243–261. Springer Lecture Notes in Computer Science, January 2002.

[77] F. Giannotti and M. Hermenegildo. A Technique for Recursive Invariance Detection and Selective Program Specialization. In *Proc. 3rd. Int'l Symposium on Programming Language Implementation and Logic Programming*, number 528 in LNCS, pages 323–335. Springer-Verlag, August 1991.

[78] R. Glück and J. Jørgensen. Generating transformers for deforestation and supercompilation. In B. Le Charlier, editor, *Proceedings of SAS'94*, LNCS 864, pages 432–448, Namur, Belgium, September 1994. Springer-Verlag.

[79] R. Glück, J. Jørgensen, B. Martens, and M. H. Sørensen. Controlling conjunctive partial deduction of definite logic programs. In H. Kuchen and S. Swierstra, editors, *Proceedings of PLILP'96*, LNCS 1140, pages 152–166, Aachen, Germany, September 1996. Springer-Verlag.

[80] R. Glück and M. Leuschel. Abstraction-based partial deduction for solving inverse problems – a transformational approach to software verification. In *Proceedings of the Third International Ershov Conference on Perspectives of System Informatics*, volume 1755 of *LNCS*, pages 93–100, Novosibirsk, Russia, 1999. Springer-Verlag.

[81] R. Glück and M. H. Sørensen. Partial deduction and driving are equivalent. In M. Hermenegildo and J. Penjam, editors, *Programming Language Implementation and Logic Programming. Proceedings, Proceedings of PLILP'94*, LNCS 844, pages 165–181, Madrid, Spain, 1994. Springer-Verlag.

[82] G. Gupta, E. Pontelli, K. Ali, M. Carlsson, and M. Hermenegildo. Parallel Execution of Prolog Programs: a Survey. *ACM Transactions on Programming Languages and Systems*, 23(4):472–602, July 2001.

[83] P. V. Hentenryck, A. Cortesi, and B. L. Charlier. Type analysis of prolog using type graphs. *Journal of Logic Programming*, 22(3):179 – 210, 1994.

[84] M. Hermenegildo, F. Bueno, D. Cabeza, M. Carro, M. García de la Banda, P. López-García, and G. Puebla. The Ciao Logic Programming Environment. In *International Conference on Computational Logic, CL2000*, July 2000.

[85] M. Hermenegildo, F. Bueno, G. Puebla, and P. López-García. Program Analysis, Debugging and Optimization Using the Ciao System Preprocessor. In *1999 International Conference on Logic Programming*, pages 52–66, Cambridge, MA, November 1999. MIT Press.

[86] M. Hermenegildo and K. Greene. The &-Prolog System: Exploiting Independent And-Parallelism. *New Generation Computing*, 9(3,4):233–257, 1991.

[87] M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Program Development Using Abstract Interpretation (and The Ciao System Preprocessor). In *10th International*

Static Analysis Symposium (SAS'03), number 2694 in LNCS, pages 127–152. Springer-Verlag, June 2003.

[88] M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Program Development Using Abstract Interpretation (and The Ciao System Preprocessor). In *Proc. of SAS'03*, pages 127–152. Springer LNCS 2694, 2003.

[89] M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems*, 22(2):187–223, March 2000.

[90] M. Hermenegildo and F. Rossi. Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. *Journal of Logic Programming*, 22(1):1–45, 1995.

[91] M. Hermenegildo, R. Warren, and S. K. Debray. Global flow analysis as a practical compilation tool. *The Journal of Logic Programming*, 13(4):349–366, 1992.

[92] J. Howe and A. King. Specialising finite domain programs using polyhedra. In A. Bossi, editor, *Logic-Based Program Synthesis and Transformation (LOPSTR'99)*, volume 1817 of *Springer-Verlag Lecture Notes in Computer Science*, pages 118–135, April 2000.

[93] G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, 1980.

[94] J. Hughes. A type specialisation tutorial. In J. Hatcliff, T. Æ. Mogensen, and P. Thiemann, editors, *Partial Evaluation: Practice and Theory*, LNCS 1706, pages 293–325, Copenhagen, Denmark, 1999. Springer-Verlag.

[95] D. Jacobs and A. Langen. Static analysis of logic programs for independent and-parallelism. *Journal of of Logic Programming*, 13(2&3):291–314, 1992.

[96] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *The Journal of Logic Programming*, 19(20):503–581, 1994.

[97] G. Janssens and M. Bruynooghe. Deriving descriptions of possible values of program variables by means of abstract interpretation. *The Journal of Logic Programming*, 13(2 & 3):205–258, 1992.

[98] G. Janssens and M. Bruynooghe. Deriving Descriptions of Possible Values of Program Variables by means of Abstract Interpretation. *Journal of Logic Programming*, 13(2 and 3):205–258, July 1992.

[99] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prenctice Hall, New York, 1993.

[100] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Software Generation*. Prentice Hall, 1993.

[101] N. D. Jones. An introduction to partial evaluation. *ACM Computing Surveys*, 28(3):480–503, September 1996.

[102] N. D. Jones. Combining abstract interpretation and partial evaluation. In P. Van Hentenryck, editor, *Static Analysis, Proceedings of SAS'97*, LNCS 1302, pages 396–405, Paris, 1997. Springer-Verlag.

[103] N. D. Jones. Combining Abstract Interpretation and Partial Evaluation. In *Static Analysis Symposium*, number 1140 in LNCS, pages 396–405. Springer-Verlag, 1997.

[104] N. D. Jones. Combining abstract interpretation and partial evaluation. In P. Van Hentenryck, editor, *Symposium on Static Analysis (SAS'97)*, volume 1302 of *Springer-Verlag Lecture Notes in Computer Science*, pages 396–405, 1997.

[105] N. D. Jones. The essence of program transformation by partial evaluation and driving. In *Proceedings of the Third International Ershov Conference on Perspectives of System Informatics*, LNCS 1755, pages 62–79, Novosibirsk, Russia, 1999. Springer-Verlag.

[106] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[107] N. D. Jones and H. Søndergaard. A semantics-based framework for the abstract interpretation of Prolog. In S. Abramski and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 6, pages 124–142. Ellis-Horwood, 1987.

[108] R. M. Karp and R. E. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, 3:147–195, 1969.

[109] H.-P. Ko and M. E. Nadel. Substitution and refutation revisited. In K. Furukawa, editor, *Logic Programming: Proceedings of the Eighth International Conference*, pages 679–692. MIT Press, 1991.

[110] J. Komorovski. An Introduction to Partial Deduction. In A. Pettorossi, editor, *Meta Programming in Logic, Proceedings of META'92*, volume 649 of *LNCS*, pages 49–69. Springer-Verlag, 1992.

[111] J. Komorowski. Partial evaluation as a means for inferencing data structures in an applicative language: a theory and implementation in the case of Prolog. In *Ninth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. Albuquerque, New Mexico*, pages 255–267, 1982.

[112] J. Komorowski. An introduction to partial deduction. In A. Pettorossi, editor, *Proceedings Meta'92*, LNCS 649, pages 49–69. Springer-Verlag, 1992.

[113] L. Lafave and J. Gallagher. Constraint-based partial evaluation of rewriting-based functional logic programs. In N. Fuchs, editor, *Logic Program Synthesis and Transformation. Proceedings of LOPSTR'97*, LNCS 1463, pages 168–188, Leuven, Belgium, July 1997.

[114] J.-L. Lassez, M. Maher, and K. Marriott. Unification revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan-Kaufmann, 1988.

[115] B. Le Charlier and P. Van Hentenryck. Experimental evaluation of a generic abstract interpretation algorithm for Prolog. *ACM Transaction on Programming Langauges and Systems*, 16(1):35–101, 1994.

[116] H. Lehmann and M. Leuschel. Generating inductive verification proofs for Isabelle using the partial evaluator Ecce. Technical Report DSSE-TR-2002-02, Department of Electronics and Computer Science, University of Southampton, UK, September 2002.

[117] M. Leuschel. Ecological partial deduction: Preserving characteristic trees without constraints. In Logic Program Synthesis and Transformation. *Pre-Proceedings of LOPSTR'95*, Utrecht, Netherlands, September 1995. Extended version as Technical Report CW 216, K.U. Leuven.

[118] M. Leuschel. Ecological partial deduction: Preserving characteristic trees without constraints. In M. Proietti, editor, *Proceedings of the 5th International Workshop on Logic Program Synthesis and Transformation*. Springer-Verlag, 1995.

[119] M. Leuschel. The ECCE partial deduction system and the DPPD library of benchmarks. Obtainable via `http://www.ecs.soton.ac.uk/~mal`, 1996-2002.

[120] M. Leuschel. *Advanced Techniques for Logic Program Specialisation.* PhD thesis, K.U. Leuven, May 1997.

[121] M. Leuschel. The ECCE partial deduction system. In G. Puebla, editor, *Proceedings of the ILPS'97 Workshop on Tools and Environments for (Constraint) Logic Programming*, Universidad Politécnica de Madrid, Tech. Rep. CLIP7/97.1, Port Jefferson, USA, October 1997.

[122] M. Leuschel. On the power of homeomorphic embedding for online termination. In G. Levi, editor, Static Analysis. *Proceedings of SAS'98*, LNCS 1503, pages 230–245, Pisa, Italy, September 1998. Springer-Verlag.

[123] M. Leuschel. Program specialisation and abstract interpretation reconciled. In *Proc. of JICSLP'98*, pages 220–234. MIT Press, June 1998.

[124] M. Leuschel. Program Specialisation and Abstract Interpretation Reconciled. In *Joint International Conference and Symposium on Logic Programming*, June 1998.

[125] M. Leuschel. Logic program specialisation. In J. Hatcliff, T. Æ. Mogensen, and P. Thiemann, editors, *Partial Evaluation: Practice and Theory*, LNCS 1706, pages 155–188 and 271–292, Copenhagen, Denmark, 1999. Springer-Verlag.

[126] M. Leuschel. Logic Program Specialisation. In J. Hatcliff, T. Æ. Mogensen, and P. Thiemann, editors, *Partial Evaluation: Practice and Theory*, volume 1706 of *LNCS*, pages 155–188. Springer-Verlag, Denmark, 1999.

[127] M. Leuschel and M. Bruynooghe. Logic program specialisation through partial deduction: Control issues. *Theory and Practice of Logic Programming, Special issue on program development*, 2(4 & 5), July 2002. To appear.

[128] M. Leuschel and D. De Schreye. Constrained partial deduction and the preservation of characteristic trees. Technical Report CW 250, Departement Computerwetenschappen, K.U. Leuven, Belgium, June 1997. Accepted for Publication in New Generation Computing.

[129] M. Leuschel, D. De Schreye, and A. de Waal. A conceptual embedding of folding into partial deduction: Towards a maximal integration. In M. Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming JICSLP'96*, pages 319–332, Bonn, Germany, September 1996. MIT Press.

[130] M. Leuschel and S. Gruner. Abstract conjunctive partial deduction using regular types and its application to model checking. In *Logic Program Synthesis and Transformation (LOPSTR)*, number 2372 in LNCS. Springer, 2001.

[131] M. Leuschel and S. Gruner. Abstract partial deduction using regular types and its application to model checking. In A. Pettorossi, editor, *Proc. of 11th Int'l Workshop on Logic-based Program Synthesis and Transformation, LOPSTR'2001*, LNCS 2372, pages 91–110, Paphos, Cyprus, 2001. Springer-Verlag.

[132] M. Leuschel and S. Gruner. Abstract partial deduction using regular types and its application to model checking. In A. Pettorossi, editor, *(Pre)Proceedings of LOPSTR-2001 11th International Workshop on Logic-based Program Synthesis and Transformation (LOPSTR-2001), Paphos, Cyprus*, December 2001.

[133] M. Leuschel and J. Jørgensen. Efficient specialisation in Prolog using a hand-written compiler generator. Technical Report DSSE-TR-99-6, Department of Electronics and Computer Science, University of Southampton, Sept. 1999.

[134] M. Leuschel, J. Jørgensen, W. Vanhoof, and M. Bruynooghe. Offline specialisation in Prolog using a hand-written compiler generator. *Theory and Practice of Logic Programming*, 4(1):139–191, 2004.

[135] M. Leuschel and H. Lehmann. Coverability of reset Petri nets and other well-structured transition systems by partial deduction. In J. Lloyd, editor, *Proceedings of the International Conference on Computational Logic (CL'2000)*, LNAI 1861, pages 101–115, London, UK, 2000. Springer-Verlag.

[136] M. Leuschel and H. Lehmann. Coverability of Reset Petri Nets and other Well-Structured Transition Systems by Partial Deduction. In J. Lloyd, editor, *Proceedings of the International Conference on Computational Logic (CL'2000)*, volume 1861 of *LNCS*, London, UK, 2000. Springer-Verlag.

[137] M. Leuschel and H. Lehmann. Solving Coverability Problems of Petri Nets by Partial Deduction. In M. Gabbrielli and F. Pfenning, editors, *Proceedings of PPDP'2000*, pages 268–279, Montreal, Canada, 2000. ACM Press.

[138] M. Leuschel and B. Martens. Partial deduction of the ground representation and its application to integrity checking. In J. Lloyd, editor, *Proceedings of ILPS'95, the International Logic Programming Symposium*, Portland, USA, December 1995. MIT Press. To appear. Extended version as Technical Report CW 210, K.U. Leuven.

[139] M. Leuschel and B. Martens. Global control for partial deduction through characteristic atoms and global trees. In O. Danvy, R. Glück, and P. Thiemann, editors, *Proceedings of the 1996 Dagstuhl Seminar on Partial Evaluation*, LNCS 1110, pages 263–283, Schloß Dagstuhl, 1996.

[140] M. Leuschel, B. Martens, and D. De Schreye. Controlling generalisation and polyvariance in partial deduction of normal logic programs. *ACM Transactions on Programming Languages and Systems*, 20(1):208–258, January 1998.

[141] M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialisation. In A. Bossi, editor, *Logic-Based Program Synthesis and Transformation*, pages 62–81. Springer Verlag, LNCS 1817, 1999.

[142] M. Leuschel and T. Massart. Infinite State Model Checking by Abstract Interpretation and Program Specialisation. In A. Bossi, editor, *Proceedings of LOPSTR'99*, volume 1817 of *LNCS*, pages 63–82, Venice, Italy, Sept. 1999.

[143] M. Leuschel and D. Schreye. Logic program specialisation: How to be more specific. In H. Kuchen and S. Swierstra, editors, *Proceedings of the International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'96)*, LNCS 1140, pages 137–151, Aachen, Germany, September 1996.

[144] M. Leuschel, D. D. Schreye, and D. A. de Waal. A conceptual embedding of folding into partial deduction: towards a maximal integration. In M. Maher, editor, *Proceedings of the Joint Int,. Conf. and Symp. on Logic Programming (JICSLP'96)*. MIT Press, 1996.

[145] M. Leuschel and M. H. Sørensen. Redundant argument filtering of logic programs. In J. Gallagher, editor, *Logic Program Synthesis and Transformation. Proceedings of LOPSTR'96*, LNCS 1207, pages 83–103, Stockholm, Sweden, August 1996. Springer-Verlag.

[146] T. Lindgren and P. Mildner. The impact of structure analysis on prolog compilation. Technical Report 140, Computing Science Departament, Uppsala University, April 1997.

[147] J. Lloyd. *Foundations of Logic Programming*. Springer, second, extended edition, 1987.

[148] J. Lloyd and J. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11(3–4):217–242, 1991.

[149] J. Lloyd and J. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11(3 & 4):217–242, 1991.

[150] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.

[151] J. W. Lloyd. *Logic Programming*. Springer-Verlag, 1987.

[152] J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11:217–242, 1991.

[153] K. Marriot and P. Stuckey. The 3 R's of optimizing constraint logic programs: Refinement, Removal and Reordering. In *Proceedings of the Twentieth Symposium on Principles of Programming Languages*, pages 334–344, Charleston, South Carolina, 1993. ACM Press.

[154] K. Marriott, L. Naish, and J.-L. Lassez. Most specific logic programs. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 909–923, Seattle, 1988. IEEE, MIT Press.

[155] K. Marriott, L. Naish, and J.-L. Lassez. Most specific logic programs. *Annals of Mathematics and Artificial Intelligence*, 1:303–338, 1990.

[156] B. Martens and D. De Schreye. Automatic finite unfolding using well-founded measures. *The Journal of Logic Programming*, 28(2):89–146, August 1996.

[157] B. Martens, D. De Schreye, and T. Horváth. Sound and complete partial deduction with unfolding based on well-founded measures. *Theoretical Computer Science*, 122(1–2):97–117, 1994.

[158] B. Martens and J. Gallagher. Ensuring global termination of partial deduction while allowing flexible polyvariance. In L. Sterling, editor, *Proceedings ICLP'95*, pages 597–611, Shonan Village Center, Japan, June 1995. MIT Press.

[159] P. Mildner. *Type Domains for Abstract Interpretation: A Critical Study*. PhD thesis, Computing Science Department - Uppsala University, Uppsala, 1999.

[160] P. Mishra. Towards a theory of types in prolog. In *Proceedings of the IEEE International Symposium on Logic Programming*, 1984.

[161] T. Mogensen and P. Sestoft. Partial evaluation. In A. Kent and J. G. Williams, editors, *Encyclopedia of Computer Science and Technology*, pages 247–279. Marcel Decker, 270 Madison Avenue, New York, New YOrk 10016, 1997.

[162] T. Æ.. Mogensen. Partially static structures in a self-applicable partial evaluator. In D. Bjørner, A. Ershov, and N. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 325–347. North-Holland, 1988.

[163] S. S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers, Inc., San Francisco, California, 1997.

[164] K. Muthukumar, F. Bueno, M. G. de la Banda, and M. Hermenegildo. Automatic Compile-time Parallelization of Logic Programs for Restricted, Goal-level, Independent And-parallelism. *Journal of Logic Programming*, 38(2):165–218, February 1999.

[165] K. Muthukumar and M. Hermenegildo. Complete and Efficient Methods for Supporting Side Effects in Independent/Restricted And-parallelism. In *1989 International Conference on Logic Programming*, pages 80–101. MIT Press, June 1989.

[166] K. Muthukumar and M. Hermenegildo. Combined determination of sharing and freeness of program variables through abstract interpretation. In K. Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pages 49–63, Paris, 1991. MIT Press, Cambridge.

[167] K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *The Journal of Logic Programming*, 13(2&3):315–347, July 1992.

[168] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for HIgher-Order Logic*. LNCS 2283. Springer-Verlag, 2002.

[169] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer-Verlag, 1994.

[170] J. C. Peralta and J. P. Gallagher. Imperative program specialisation: An approach using CLP. In A. Bossi, editor, *Logic-Based Program Synthesis and Transformation*, pages 102–117. Springer Verlag, LNCS 1817, 1999.

[171] A. Pettorossi and M. Proietti. Program derivation via list introduction. In R. Bird and L. Meertens, editors, *Proceedings of the IFIP TC2 Working Conference on Algorithmic Languages and Calculi*, Le Bischenberg, France, February 1996. Chapman & Hall.

[172] A. Pettorossi and M. Proietti. A theory of logic program specialization and generalization for dealing with input data properties. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, International Seminar*, LNCS 1110, pages 386–408, Schloß Dagstuhl, 1996. Springer-Verlag.

[173] A. Pettorossi and M. Proietti. A theory of logic program specialization and generalization for dealing with input data properties. In Springer-Verlag, editor, *Dagstuhl Seminar on Partial Evaluation*, number 1110 in LNCS, 1996.

[174] A. Pettorossi and M. Proietti. Synthesis and transformation of logic programs using unfold/fold proofs. *The Journal of Logic Programming*, 41(2&3):197–230, Nov. 1999.

[175] A. Pettorossi, M. Proietti, and S. Renault. Reducing nondeterminism while specializing logic programs. In N. D. Jones, editor, *Proceedings of ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 414–427, Paris, France, January 1997.

[176] M. Proietti and A. Pettorossi. Unfolding-definition-folding, in this order, for avoiding unnecessary variables in logic programs. In J. Małuszyński and M. Wirsing, editors, *Proceedings of PLILP'91*, LNCS 528, pages 347–358. Springer-Verlag, 1991.

[177] M. Proietti and A. Pettorossi. The loop absorption and the generalization strategies for the development of logic programs and partial deduction. *The Journal of Logic Programming*, 16(1 & 2):123–162, May 1993.

[178] G. Puebla, F. Bueno, and M. Hermenegildo. A Generic Preprocessor for Program Validation and Debugging. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 63–107. Springer-Verlag, September 2000.

[179] G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Constraint Logic Programs. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 23–61. Springer-Verlag, September 2000.

[180] G. Puebla, F. Bueno, and M. Hermenegildo. Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs. In *Logic-based Program Synthesis and Transformation (LOPSTR'99)*, number 1817 in LNCS, pages 273–292. Springer-Verlag, 2000.

[181] G. Puebla, M. G. de la Banda, K. Marriott, and P. Stuckey. Optimization of Logic Programs with Dynamic Scheduling. In *1997 International Conference on Logic Programming*, pages 93–107, Cambridge, MA, June 1997. MIT Press.

[182] G. Puebla, J. Gallagher, and M. Hermenegildo. Towards integrating partial evaluation in a specialization framework based on generic abstract interpretation. In M. Leuschel,

editor, *Proceedings of the ILPS'97 Workshop on Specialisation of Declarative Programs and its Application*, K.U. Leuven, Tech. Rep. CW 255, pages 29–38, Port Jefferson, USA, October 1997.

[183] G. Puebla, J. Gallagher, and M. Hermenegildo. Towards Integrating Partial Evaluation in a Specialization Framework based on Generic Abstract Interpretation. In M. Leuschel, editor, *Proceedings of the ILPS'97 Workshop on Specialization of Declarative Programs*, October 1997. Post ILPS'97 Workshop.

[184] G. Puebla and M. Hermenegildo. Implementation of multiple specialization in logic programs. In *Proceedings of PEPM'95, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 77–87, La Jolla, California, June 1995. ACM Press.

[185] G. Puebla and M. Hermenegildo. Implementation of Multiple Specialization in Logic Programs. In *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation*, pages 77–87. ACM Press, June 1995.

[186] G. Puebla and M. Hermenegildo. Abstract specialization and its application to program parallelization. In J. Gallagher, editor, *Logic Program Synthesis and Transformation. Proceedings of LOPSTR'96*, LNCS 1207, pages 169–186, Stockholm, Sweden, August 1996.

[187] G. Puebla and M. Hermenegildo. Optimized Algorithms for the Incremental Analysis of Logic Programs. In *International Static Analysis Symposium*, number 1145 in LNCS, pages 270–284. Springer-Verlag, September 1996.

[188] G. Puebla and M. Hermenegildo. Abstract Multiple Specialization and its Application to Program Parallelization. *J. of Logic Programming. Special Issue on Synthesis, Transformation and Analysis of Logic Programs*, 41(2&3):279–316, November 1999.

[189] G. Puebla and M. Hermenegildo. Abstract Multiple Specialization and its Application to Program Parallelization. *J. of Logic Programming. Special Issue on Synthesis, Transformation and Analysis of Logic Programs*, 41(2&3):279–316, November 1999.

[190] G. Puebla and M. Hermenegildo. Some Issues in Analysis and Specialization of Modular Ciao-Prolog Programs. In *Special Issue on Optimization and Implementation of Declarative Programming Languages*, volume 30 of *Electronic Notes in Theoretical Computer Science*. Elsevier - North Holland, March 2000.

[191] G. Puebla and M. Hermenegildo. Abstract Specialization and its Applications. In *ACM Partial Evaluation and Semantics based Program Manipulation (PEPM'03)*, pages 29–43. ACM Press, June 2003. Invited talk.

[192] G. Puebla, M. Hermenegildo, and J. Gallagher. An Integration of Partial Evaluation in a Generic Abstract Interpretation Framework. In O. Danvy, editor, *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99)*, number NS-99-1 in BRISC Series, pages 75–85. University of Aarhus, Denmark, January 1999.

[193] G. Puebla, M. Hermenegildo, and J. P. Gallagher. An integration of partial evaluation in a generic abstract interpretation framework. In O. Danvy, editor, *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99)*, Technical report BRICS-NS-99-1, University of Aarhus, pages 75–84, San Antonio, Texas, Jan. 1999.

[194] J. C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. *Machine Intelligence*, pages 135–151, 1970.

[195] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1999.

[196] H. Saglam and J. Gallagher. Approximating logic programs using types and regular descriptions. Technical Report CSTR-94-19, Department of Computer Science, University of Bristol, Bristol BS8 1TR, 1994.

[197] H. Sağlam and J. P. Gallagher. Constrained regular approximations of logic programs. In N. Fuchs, editor, *LOPSTR'97*, pages 282–299. Springer-Verlag, LNCS 1463, 1997.

[198] D. Sahlin. Mixtus: An automatic partial evaluator for full Prolog. *New Generation Computing*, 12(1):7–51, 1993.

[199] D. Sands. Total correctness by local improvement in the transformation of functional programs. *ACM Transactions on Programming Languages and Systems*, 18(2):175–234, Mar. 1996.

[200] U. Shankar. An Introduction to Assertional Reasoning for Concurrent systems. *ACM Computing Surveys*, 43(3):225–262, 1993.

[201] D. D. Shreye, R. Glück, J. Jørgensen, M. Leuschel, B. Martens, and M. H. Sørensen. Conjunctive partial deduction: Foundations, control, algorithms, and experiments. *The*

Journal of Logic Programming, 41(2-3):231–277, 1999. Erratum appeared in JLP 43(3): 265(2000).

[202] M. Sørensen and R. Glück. An Algorithm of Generalization in Positive Supercompilation. In *Proc. of ILPS'95*, pages 465–479. The MIT Press, 1995.

[203] M. H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In J. W. Lloyd, editor, *Proceedings of ILPS'95, the International Logic Programming Symposium*, pages 465–479, Portland, USA, December 1995. MIT Press.

[204] M. H. Sørensen, R. Glück, and N. D. Jones. Towards unifying partial evaluation, deforestation, supercompilation, and GPC. In D. Sannella, editor, *Programming Languages and Systems — ESOP '94. Proceedings*, LNCS 788, pages 485–500, Edinburgh, Scotland, 1994. Springer-Verlag.

[205] M. H. Sørensen, R. Glück, and N. D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.

[206] H. Tamaki and M. Sato. Unfold/Fold Transformations of Logic Programs. In *Second International Conference on Logic Programming*, pages 127–138, Uppsala, Sweden, 1984.

[207] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.

[208] V. Turchin. The algorithm of generalization in the supercompiler. In D. B. rner, A. Ershov, and N. Jones, editors, *Proc. of the IFIP TC2 Workshop on Partial Evaluation and Mixed Computation*, pages 531–549. North-Holland, 1988.

[209] V. F. Turchin. Program transformation with metasystem transitions. *Journal of Functional Programming*, 3(3):283–313, 1993.

[210] V. F. Turchin. Metacomputation: Metasystem transitions plus supercompilation. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, International Seminar*, LNCS 1110, pages 482–509, Schloß Dagstuhl, 1996. Springer-Verlag.

[211] J. Ullman. Implementation of Logical Query Languages for Databases. *ACM Transactions on Database Systems*, 10(3), 1985.

[212] P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Type analysis of prolog using type graphs. *Journal of Logic Programming*, 22(3):179–209, 1995.

[213] C. Vaucheret and F. Bueno. More precise yet efficient type inference for logic programs. In *International Static Analysis Symposium*, number 2477 in LNCS, pages 102–116. Springer-Verlag, September 2002.

[214] C. Vaucheret and F. Bueno. More precise yet efficient type inference for logic programs. In *Proc. of SAS'02*, pages 102–116. Springer LNCS 2477, 2002.

[215] P. Wadler. Deforestation: Transforming programs to eliminate intermediate trees. *Theoretical Computer Science*, 73:231–248, 1990. Preliminary version in ESOP'88, LNCS 300.

[216] W. Winsborough. Multiple Specialization using Minimal-Function Graph Semantics. *Journal of Logic Programming*, 13(2 and 3):259–290, July 1992.

[217] E. Yardeni and E. Shapiro. A type system for logic programs. *The Journal of Logic Programming*, 10(2):125–154, 1990.

[218] E. Yardeni and E. Shapiro. A type system for logic programs. *Journal of Logic Programming*, 10(2):125–154, 1990.