Program Specialisation and Abstract Interpretation Reconciled

Michael Leuschel[1]
Department of Computer Science, K.U. Leuven, Belgium
DIKU, University of Copenhagen, Denmark

Abstract

We clarify the relationship between abstract interpretation and program specialisation in the context of logic programming. We present a generic top-down abstract specialisation framework, along with a generic correctness result, into which a lot of the existing specialisation techniques can be cast. The framework also shows how these techniques can be further improved by moving to more refined abstract domains. It, however, also highlights inherent limitations shared by all these approaches. In order to overcome them, and to fully unify program specialisation with abstract interpretation, we also develop a generic combined bottom-up/top-down framework, which allows specialisation *and* analysis outside the reach of existing techniques.

1 Introduction

At first sight *abstract interpretation* (see, e.g., [5, 3]) and *program specialisation* (see, e.g., [7]) might appear to be completely unrelated techniques: abstract interpretation focusses on *correct and precise* analysis, while the main goal of program specialisation is to produce more *efficient residual code* (for a given task at hand). Nonetheless, it is often felt that there is a close relationship between abstract interpretation and program specialisation and, recently, there has been a lot of interest in the integration of these two techniques (see, e.g., [16, 11, 22].

Indeed, for good specialisation to take place, program specialisers have to perform some form of analysis. For instance, the incomplete SLD-trees produced by *partial deduction* [20, 7] can be seen as complete (given the closedness condition of [20]) description of the top-down computation-flow.

In this paper we want to substantiate this intuition and make the link to abstract interpretation fully explicit. We therefore present a generic (augmented) top-down abstract interpretation framework in which most of the specialisation techniques (such as partial deduction [20, 7], ecological partial deduction [18, 13], constrained partial deduction [15], conjunctive partial deduction [17, 10]) can be cast. It also paves the way for more refined and

[1]The work was done while the author was Post-doctoral Fellow of the Belgian Fund for Scientific Research (F.W.O.-Vlaanderen). and visiting DIKU, University of Copenhagen. The author's present address is: Dept. of Electronics and Computer Science, University of Southampton, Southampton SO17 1BJ, UK. E-mail : mal@ecs.soton.ac.uk

powerful specialisation, by allowing more refined abstract domains and more refined "abstract unfolding" rules.

However, we will not stop there. The above formalisation will actually make two (additional) shortcomings of most earlier specialisation techniques apparent (already identified in, e.g., [16, 13]): the lack of *side-ways information passing* and of *inference of global success information*. Recent techniques [16, 23, 22] (as well as some earlier attempts such as [21] and [9, 6]) have tried to overcome these limitations by incorporating *bottom-up* abstract interpretation techniques. However, we feel that a fully satisfactory integration of program specialisation with abstract interpretation has not been achieved yet, and we strive to do so in this paper.

This integration is not solely beneficial for specialisation purposes. Indeed, as shown in [16, 13] (for one particular abstract domain), a full integration of abstract interpretation with program specialisation can yield *analysis* outside the reach of either method alone (and can even be used to perform inductive theorem proving or infinite model checking).

We thus present an augmented, combined top-down/bottom-up abstract specialisation framework in which all these earlier techniques (and more) can be cast, and provide the first "full-blown" integration of abstract interpretation and program specialisation, leading towards powerful specialisation *and* analysis beyond the reach of existing techniques.

2 Top-Down Abstract Partial Deduction

In this paper, we restrict ourselves to definite programs and goals (but possibly with declarative built-in's such as is, $call$, $functor$, arg, $\backslash==$; this allows to express a very large number of interesting, practical programs; one can even implement and use certain higher-order features such as $map/3$).

An *expression* is either a term, an atom or a conjunction. We use $E_1 \preceq E_2$ to denote that the expression E_1 is an instance of the expression E_2. By $mgu(E_1, E_2)$ we denote a most general unifier and by $msg(E_1, E_2)$ a most specific generalisation of E_1 and E_2. Also, as common in partial deduction, the notion of SLD-trees is extended to allow *incomplete* SLD-trees which may contain leaves where no literal has been selected for further derivation.

2.1 The Abstract Domain

We denote by $\mathcal{Q}$ the set of all conjunctions. Our abstract domain $\mathcal{AQ}$ is then a set of *abstract conjunctions* equipped with a *concretisation function* $\gamma : \mathcal{AQ} \mapsto 2^{\mathcal{Q}}$, providing the link between the abstract and the concrete domain. We suppose that $\gamma(\mathbf{A})$ is always *downwards closed* ($Q \in \gamma(\mathbf{A}) \Rightarrow Q\theta \in \gamma(\mathbf{A})$), i.e. we restrict ourselves to "declarative" properties. Also, for reasons that will become clear below, we suppose that all conjunctions in $\gamma(\mathbf{A})$ have the same number of conjuncts and with the same predicates at

the same position. Observe that this still admits the possibility of a bottom element $\bot$ whose concretisation is empty.

We will denote the fact that $\gamma(\mathbf{A_1}) \subseteq \gamma(\mathbf{A_2})$ by $\mathbf{A_1} \sqsubseteq \mathbf{A_2}$. We also sometimes use γ on sets of abstract conjunctions: $\gamma(S) = \{Q \mid Q \in \gamma(\mathbf{A}) \wedge \mathbf{A} \in S\}$. One particular abstract domain, which will often serve for illustration purposes, is the $\mathcal{PD}$-domain where $\mathcal{AQ} = \mathcal{Q}$ (i.e. the abstract conjunctions are the concrete ones) and $\gamma(Q) = \{Q' \mid Q' \preceq Q\}$ (i.e. an abstract conjunction denotes all its instances).

2.2 Abstract Unfolding

Program specialisation can achieve more efficient residual code — amongst others — by pre-computing certain operations at compile time (which then no longer have to be performed at run-time). In other words, one computation step in the residual program may actually represent an entire sequence of computation steps within the original program.

In the context of logic programming, this can be seen as producing a *residual* clause which, when resolved against, has the same effect as a sequence of resolution steps in the original program. Partial deduction, for example, produces these clauses by *unfolding* an atom A, thereby producing an SLD-tree τ for $P \cup \{\leftarrow A\}$. Every non-failing branch of τ is translated into a residual clause by taking the *resultant* of the derivation.[2] These resultants can then be used in a sound manner for any concretisation of A (i.e., any instance of A) in the sense that resolution will lead to computed answers and resolvents which can also be obtained in the original program. But actually the use by partial deduction algorithms of these resultants is not limited to code generation. Take, e.g., the resultant $p(f(X)) \leftarrow q(f(X))$. When resolving a particular runtime call $p(\bar{t})$ with that resultant we will obtain resolvents which are instances of $\leftarrow q(f(X))$. Partial deduction therefore also analyses (and specialises) the atom $q(f(X))$. In other words, the body of the residual clause is used for the flow analysis as a representative of all possible resolvents. This multiple use of the residual clauses relies on using an abstract domain identical to the concrete domain.

In the more general setting we endeavour to develop, these two roles of unfolding will have to be separated out (as the residual program has to be expressed in the concrete domain). In other words, to specialise an abstract conjunction $\mathbf{A}$ we generate:

- resultants $H_i \leftarrow B_i$, totally correct for the calls in $\gamma(\mathbf{A})$ and
- for each resultant $H_i \leftarrow B_i$ an abstract conjunction $\mathbf{A_i}$ approximating all the possible resolvent goals which can occur after resolving an element of $\gamma(\mathbf{A})$ with $H_i \leftarrow B_i$.

This leads to Definition 2.1 of abstract unfolding and resolution below. (Observe that the resultants $H_i \leftarrow B_i$ below are not necessarily Horn clauses. We will discuss the generation of Horn clause programs later in Section 2.4.)

[2] A resultant is a formula $H \leftarrow B$ where H and B are conjunctions of literals.

First, we introduce the following notations. Given an SLD-tree τ for $P \cup \{\leftarrow Q\}$ we denote by $Q \leadsto^{\theta}_{\tau} L$ the fact that a leaf goal L of τ can be reached via c.a.s. θ. Given a resultant $C_i = H_i \leftarrow B_i$ and a conjunction Q we denote by $Q \leadsto^{\theta}_{C_i} L$ the fact that $mgu(Q, H_i) = \theta'$ with $\theta' \mid_{vars(Q)} = \theta$ and $L = B_i\theta'$.[3]

Definition 2.1 An *abstract unfolding* operation $aunf(.)$ 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 an SLD-tree τ for $P \cup \{\leftarrow Q\}$ such that:

$$Q \leadsto^{\theta}_{\tau} L \quad \Leftrightarrow \quad \exists C_i \in aunf(\mathbf{A}) \text{ s.t. } Q \leadsto^{\theta}_{C_i} L \tag{1}$$

An *abstract resolution* operation $ares(.)$ maps an abstract conjunction $\mathbf{A}$ and a concrete resultant C_i to another abstract conjunction such that for all $Q \in \gamma(\mathbf{A})$: $\quad Q \leadsto^{\theta}_{C_i} L \Rightarrow L \in \gamma(ares(\mathbf{A}, C_i))$.

The $\Rightarrow$ part of point 1 requests that the code generated by $aunf(.)$ is *complete* while the $\Leftarrow$ part additionally requests *soundness* (as we want to have residual code which is *totally correct* and not just a safe approximation). We call an abstract unfolding rule *conservative* if the $\Leftarrow$ part of point 1 holds for all Q (and not just for $Q \in \gamma(\mathbf{A})$).

Example 2.2 Let P be the following program:

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

Let $\mathbf{A} = eq([a|T], Z)$ in the $\mathcal{PD}$-domain. $aunf(\mathbf{A}) = \{H \leftarrow B\}$ and $ares(\mathbf{A}, H \leftarrow B) = eq(T, Y)$ where $H = eq(X, [a|Y])$ and $B = eq(X, Y)$ are correct. Also, both remain correct with $H = eq(H, [H|Y])$ but not with $H = eq(X, [b|Y])$. $H = eq([H], [H|Y])$ and $B = eq([], Y)$ is also incorrect.

Observe, that in Definition 2.1 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 optimisations not achievable within existing "classical" specialisation frameworks. For instance, in the example below we are able to completely eliminate a type-like check from the residual program.

Example 2.3 Let P be the program from Example 2.2 and $\mathbf{A}$ represent the set of all calls $eq(L, L)$ where L is a bounded (nil-terminated) list (this can obviously not be represented in the $\mathcal{PD}$-domain). Then $aunf(\mathbf{A}) = C_1 = \{eq(X, Y) \leftarrow\}$ and $ares(\mathbf{A}, C_1) = \Box$ are correct according to the above definition ! One can thus generate the residual code:

$eq(X, Y) \leftarrow$

Observe that this abstract unfolding is, in contrast to Example 2.2, not conservative. In other words the residual code is only sound for concretisations of $\mathbf{A}$ but not, e.g., for the call $eq(a, [])$.

[3] If Q and H_i are atoms this is equivalent to saying that $\leftarrow Q$ resolves with the clause $H_i \leftarrow B_i$ via c.a.s. θ yielding $\leftarrow L$ as resolvent. Also observe that for any Q, C_i and there is at most one choice of θ and L such that $Q \leadsto^{\theta}_{C_i} L$.

Example 2.4 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 $\mathbf{A}$ represent all calls $p(X)$ where X has type $\tau ::= a \mid g(\tau)$. Then $aunf(\mathbf{A}) = \{C_1, C_3\}$, $ares(\mathbf{A}, C_1) = \Box$ and $ares(\mathbf{A}, C_3) = \mathbf{A}$ is correct and by abstract unfolding we are able to safely remove the redundant clause C_2.

To more concisely express the flow analysis, we extend $aunf(.)$ so that it maps sets of abstract conjunctions to sets of abstract conjunctions in the following way: $aunf^*(S) = \{ares(\mathbf{A}, C) \mid C \in aunf(\mathbf{A})\}$.

2.3 Widening by Splitting

The computation flow aspect of program specialisation could now be performed by calculating $U \uparrow^\infty$, where $U(S) = S \cup aunf^*(S)$. However, it is obvious that, for but the simplest abstract domains, this construction will not terminate and that generalisation is required.

As usual in abstract interpretation, one could imagine to represent generalisation by a widening function $\omega : \mathcal{AQ} \mapsto \mathcal{AQ}$ such that $\forall \mathbf{A} \in \mathcal{AQ} : \mathbf{A} \sqsubseteq \omega(\mathbf{A})$. Unfortunately, this is not enough to be able to ensure termination of abstract interpretation in the present setting, because all concretisations of an abstract conjunction must have the same number of conjuncts. In other words, no terminating analysis could be produced for, e.g., a program containing the clause $p \leftarrow p, p$. This is why we need a more refined notion of widening, which involves splitting conjunctions into subconjunctions:

Definition 2.5 A sequence $\langle \mathbf{A_1}, \ldots, \mathbf{A_n} \rangle$ of abstract conjunctions is an *abstraction* of an abstract conjunction $\mathbf{A}$ iff $\gamma(\mathbf{A}) \subseteq \{Q_1 \wedge \ldots \wedge Q_n \mid Q_i \in \gamma(\mathbf{A_i})\}$.

Observe that for $i = 1$ this condition is equivalent to $\mathbf{A} \sqsubseteq \mathbf{A_1}$. Also observe that this splitting operation does not allow re-ordering of conjunctions. It is, however, straightforward to do so. One just has to be careful to use the *same* reordering for *all* concretisations (otherwise it will be impossible to synchronise the code generation with the flow analysis, cf. the next subsection).

We extend the abstraction concept to sets:

Definition 2.6 A set $\mathcal{A}'$ is called an *abstraction* of another set of abstract conjunctions $\mathcal{A}$, denoted by $\mathcal{A}' \sqsupseteq_{split} \mathcal{A}$, iff for all $\mathbf{A} \in \mathcal{A}$ there exists an abstraction $\langle \mathbf{A_1}, \ldots, \mathbf{A_n} \rangle$ of $\mathbf{A}$ such that all $\mathbf{A_i} \in \mathcal{A}'$.

For example, in the $\mathcal{PD}$-domain, $\langle p(X) \wedge q(X), p(b) \rangle$ is an abstraction of $p(b) \wedge q(b) \wedge p(b)$ and we have thus, for example, $\{p(X) \wedge q(X), p(b)\} \sqsupseteq_{split} \{p(b) \wedge q(b) \wedge p(b), p(c) \wedge q(c)\}$.

We can now define a more refined *widening operator* to be a function[4] $\omega : 2^{\mathcal{AQ}} \mapsto 2^{\mathcal{AQ}}$ satisfying that $\omega(\mathcal{A}) \sqsupseteq_{split} \mathcal{A}$ for all $\mathcal{A}$.

By using appropriate widening operators it is now possible to ensure termination for any program (we refer the reader, e.g., to [18, 10, 13] on how to devise ω in the context of partial deduction).

We also say that a set $\mathcal{A}$ of abstract conjunctions is *covered* iff $\mathcal{A} \sqsupseteq_{split} aunf^*(\mathcal{A})$. Intuitively, this means that $\mathcal{A}$ is a complete description of the computation flow (induced by $aunf(.)$) for all concretisations of $\mathcal{A}$.

2.4 Generating Residual Code

Generating residual code from the resultants $H_i \leftarrow B_i$ produced by the abstract unfolding involves transforming them into Horn clauses. This can be achieved by mapping the abstract conjunctions produced by the flow analysis to atoms and then appropriately renaming the heads H_i and the bodies B_i.

We first introduce the following concepts. A *concrete dominator* of an abstract conjunction $\mathbf{A}$ is a concrete conjunction which is more general than all the concretisations of $\mathbf{A}$. A *skeleton* for an abstract conjunction $\mathbf{A}$ is a maximally general concrete dominator of $\mathbf{A}$. By our earlier assumption that all conjunctions in $\gamma(\mathbf{A})$ have the same predicates at the same position we know that a concrete dominator (and thus skeleton) exists for all abstract conjunctions. By $\lceil \mathbf{A} \rceil$ we denote some skeleton for $\mathbf{A}$.

We also require that for all $\mathbf{A} \in \mathcal{AQ}$ and $H_i \leftarrow B_i \in aunf(\mathbf{A})$ we have $H_i \preceq \lceil \mathbf{A} \rceil$. The requirement prevents garbage code (any $H_i \not\preceq \lceil \mathbf{A} \rceil$ can never unify with a concretisation of $\mathbf{A}$) and simplifies the construction below.

Definition 2.7 An *atomic renaming* $\rho_{\mathbf{A}}$ for an abstract conjunction $\mathbf{A}$ is an atom 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 $Q = \lceil \mathbf{A} \rceil \theta$.

In the $\mathcal{PD}$-domain, we might have $\mathbf{A} = p(f(X)) \wedge q(Z)$, $\lceil \mathbf{A} \rceil = p(X) \wedge q(Y)$, $\rho_{\mathbf{A}} = pq(X, Y)$ and $Q = p(f(a)) \wedge q(b)$. In that case $\rho_{\mathbf{A}}(Q) = pq(f(a), b)$.

Observe that for all $Q \preceq \lceil \mathbf{A} \rceil$ we have $\rho_{\mathbf{A}}(Q\theta) = \rho_{\mathbf{A}}(Q)\theta$. 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 $H_i \leftarrow B_i \in aunf(\mathbf{A})$ 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). If our flow analysis also contains $\mathbf{A_i} = ares(\mathbf{A}, H_i \leftarrow B_i)$ (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 applied a widening step and that we actually did not analyse $\mathbf{A_i}$ but an abstraction $\langle \mathbf{G_1}, \ldots, \mathbf{G_n} \rangle$ of it. In that case B_i has to be chopped up

[4] It is of course possible to give extra parameters to ω, e.g., to take the specialisation history into account.

and then renamed using the renaming functions of the abstraction. We thus define $\rho_{\mathcal{A},\mathbf{A}}(B) = \rho_{\mathbf{G_1}}(B_1) \wedge \ldots \wedge \rho_{\mathbf{G_1}}(B_n)$ where $B = B_1 \wedge \ldots \wedge B_n$ and $\langle \mathbf{G_1}, \ldots, \mathbf{G_n} \rangle$ is an abstraction of $\mathbf{A}$ 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.

Definition 2.8 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:
$\{\rho_{\mathbf{A}}(H) \leftarrow \rho_{\mathcal{A},\mathbf{A}'}(B) \mid H \leftarrow B \in \mathit{aunf}(\mathbf{A}) \wedge\ \mathbf{A}' = \mathit{ares}(\mathbf{A}, H \leftarrow B) \wedge \mathbf{A} \in \mathcal{A}\}$.

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

Observe that, a skeleton always has distinct variables as its only terms. In other words, we perform no structure filtering (i.e. $p(f(a))$ might get renamed into $p'(f(a))$ but never into $p'(a)$ or p'). Filtering could be achieved by using a concrete dominator, ideally $\mathit{msg}(\gamma(\mathbf{A}))$, instead of the skeleton $\lceil \mathbf{A} \rceil$ for the definition of $\rho_{\mathbf{A}}$. This, however, makes the exposition more tricky and would detract from the main points of the paper. Anyway, one can always apply [8] (as well as RAF, see, e.g., [13]) as a post-processing.

2.5 A Generic Correctness Result and Algorithm

We can now present a very general correctness result.

Theorem 2.9 Let P' be an abstract partial deduction of P wrt a covered set of abstract conjunctions $\mathcal{A}$ and let $Q \in \gamma(\mathbf{A})$ with $\mathbf{A} \in \mathcal{A}$. Then $P \cup \{\leftarrow Q\}$ has an SLD-refutation with c.a.s. θ iff $P' \cup \{\leftarrow \rho_{\mathbf{A}}(Q)\}$ has an SLD-refutation with c.a.s. θ.

Proofs can be found in the technical report [14]. In order to derive results about the preservation of finite failure we have to impose that the unfolding operation $\mathit{aunf}(.)$ is *fair*[5], i.e. when computing $\mathit{aunf}(\mathbf{A})$ it eventually selects every conjunct of $Q \in \gamma(\mathbf{A})$ in every non-failing branch. One can then prove (by reusing results from [17, 13]) that:

$P \cup \{\leftarrow Q\}$ has a finitely failed SLD-tree iff $P' \cup \{\leftarrow \rho_{\mathbf{A}}(Q)\}$ has.

Based upon the notions introduced above, we can now present a generic algorithm for top-down program specialisation in a very concise manner:

Algorithm 2.10 (Top-Down Abstract Partial Deduction)
Input: A program P and an abstract conjunction $\mathbf{A}$
Output: A specialised program P'
Initialise: $i = 0$, $\mathcal{A}_0 = \{\mathbf{A}\}$
repeat
  let $\mathcal{A}_{i+1} := \omega(\mathcal{A}_i \cup \mathit{aunf}^*(\mathcal{A}_i))$; **let** $i := i + 1$;
until $\mathcal{A}_{i-1} = \mathcal{A}_i$
Let P' be an abstract partial deduction wrt $\mathcal{A}_i$

[5] Or even better weakly fair, see [17, 13].

The differences over "traditional" top-down abstract interpretation methods for logic programs (like, e.g., the top-down component of [3]) are:
1) abstract *conjunctions* instead of abstract atoms are used, 2) widening can generalise by "going up the lattice" *and* by splitting, 3) a *full abstract unfolding* (which can do, e.g., deforestation) is used instead of just a single abstract resolution step, and 4) there is no sideways information passing between abstract conjunctions (but perfect [16] sideways propagation within each abstract conjunction).

2.6 Expressing Existing Partial Deduction Techniques

Classical partial deduction [20, 7] can be seen as an instance of the above generic framework by taking

- the $\mathcal{PD}$-domain (i.e. the concrete domain is the abstract domain and an abstract element represents all its instances),
- abstract unfolding performs concrete resolution steps,
- ω will only produce sets of atoms and the initial abstract conjunction $\mathbf{A}$ is an atom.

To represent *conjunctive* partial deduction [17, 10, 13] we just have to drop the last requirement. *Ecological* partial deduction [18, 13] can be seen as an instance of the above generic framework by taking

- $\mathcal{AQ} = (\mathcal{A}, T)$, where $\mathcal{A}$ is the set of atoms and T is the set of characteristic trees.
- $\gamma((A,\tau)) = \{A'' \mid A'' \preceq A' \preceq A \wedge\ A' \text{ has characteristic tree } \tau\ \}$,
- $aunf((A,\tau))$ is based on using the SLD-tree τ (see [18, 13]).

Similarly, *constrained* partial deduction [15] can be cast into the present framework , and its correctness results are a special case of the ones above.

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 specialisation method.

2.7 Future Prospects

Improved Generalisation A lot of existing specialisation techniques (see, e.g., [18, 10]) ensure termination by using refined methods, such as homeomorphic embedding $\trianglelefteq$, to detect "dangerous" growth of structure. However, once such a growth has been detected these techniques still have to rely on rather crude generalisation operators, such as more specific generalisation (msg), because the resulting generalisation (or part of it, as in [18]) has to be expressed in the $\mathcal{PD}$-domain. For instance, when a specialiser goes from $A_1 = p(a)$ to $A_2 = p(f(a))$ then the homeomorphic embedding $\trianglelefteq$ will signal danger ($A_1 \trianglelefteq A_2$) and will even pinpoint the extra $f(.)$ in A_2 as the potential source of non-termination. But the msg of A_1 and A_2 is just $p(X)$ and no use of the information provided by $\trianglelefteq$ was made (nor is it possible to do so in

the $\mathcal{PD}$-domain). In the enriched context of abstract partial deduction, however, we can now derive, e.g., a (regular) *type* describing the growth detected by $\trianglelefteq$ and arrive at much more intelligent generalisation and much improved specialisation. For instance A_1 and A_2 can be abstracted by something like $p(X{:}\,\tau)$ where the type τ is defined as $\tau ::= a \mid f(\tau)$. Also, atoms such as $p([])$ and $p([H|T])$ can be abstracted by $p(X{:}\,\mathbf{list})$. The example worked out in [14] makes use of that possibility.

Improved Unfolding and Code Generation As already hinted at in Section 2.2, our enriched abstract unfolding operation allows us to generate much more efficient code. Given the simple program

$p([]) \leftarrow$
$p([H|T]) \leftarrow p(T)$

one can, e.g., use the information that a particular variable X is a list to abstractly unfold $p(X)$ into $p(Z) \leftarrow$, i.e. generate a single residual fact instead of the "usual" recursive definition. This is something that *no* other specialisation framework (we are aware of) can currently achieve. In languages like Mercury or Gödel such type information will even be explicitly given and does not have to be inferred. We believe that our framework(s) will be especially useful for these languages.

Improved Handling of Built-in's If we know that a given variable X represents an integer we can, e.g., specialise both $atomic(X)$ or $number(X)$ into $true$. One can imagine various other optimisations not possible in conventional techniques based upon the $\mathcal{PD}$-domain, like specialising arg or $functor$ calls based upon type information of the arguments. A similar idea has been used in [23] and [22] to remove groundness tests (controlling parallel execution) from the residual program.

3 A Bottom-Up Analysis

Although using refined abstract domains within Algorithm 2.10 can lead to major improvements over existing specialisation techniques, it is still not possible to achieve side-ways (between different abstract conjunctions) or bottom-up success information propagation. A (seemingly) simple way to add bottom-up success information propagation to our abstract partial deduction framework is to request point 1 only for $Q \in \gamma(\mathbf{A}) \cap SS_P$ (instead of $Q \in \gamma(\mathbf{A})$) in Definition 2.1 of $aunf(.)$, where SS_P is the success set of P. In practice this means that the operation $aunf(.)$ can make use of a *subsidiary* (bottom-up) abstract interpretation phase to approximate SS_P. In order to achieve some interaction between the top-down and bottom-up components, one could imagine that the abstract interpretation takes the unfoldings into account (this is an approach proposed in [22]). This, however, means that one has to re-analyse whenever a new unfolding has been performed and to

re-specialise whenever a tighter success set has been derived. The precise details of this "co-routining" are non-trivial and one can hardly call the above an algorithm. Furthermore, there are a considerable number of tasks (see [16]) that such an approach simply cannot handle, because the specialisation and analysis components basically still work in isolation. In this paper, we will therefore first present a pure bottom-up analysis algorithm, but which we then *fully* integrate with Algorithm 2.10 in Section 4.

In a bottom-up setting we need, instead of an abstraction of the unfolding operation, an abstraction of the bottom-up T_P operator [1, 19], or better its non-ground version (to capture the C-semantics and thus the computed answers). The (non-ground) T_P operator maps interpretations to interpretations, where an interpretation is usually represented by a set of atoms. Each interpretation in turn can be seen as representing (an approximation to) the success set. One could thus define an abstract version of T_P which maps a set of atomic abstract conjunctions to a set of atomic abstract conjunctions such that $\gamma(AT_P(\mathcal{A})) \supseteq \{H\theta_1 \ldots \theta_n \mid H \leftarrow B_1, \ldots, B_n \in P \wedge\ A_i \in \gamma(\mathcal{A}) \wedge\ \theta_i = mgu(B_i\theta_1, \ldots, \theta_{i-1}, A_i)\}$. However, in light of a full integration with the framework of Section 2 (and in order to be able to capture all the techniques of [21]), we will describe a more refined abstraction of T_P based on *conjunctions* (instead of just atoms) and *resultants* derived by $aunf(.)$ (instead of simply the clauses of the original program P).

Abstract T_P for Abstract Conjunctions

So, instead of just representing the success set for each predicate in general, we want to represent success sets for a given choice of abstract conjunctions $\mathcal{A} = \{\mathbf{A_1}, \ldots, \mathbf{A_n}\}$. This is accomplished by the following definition.

Definition 3.1 An *abstracted interpretation* is a set $\{(\mathbf{A_1}, \mathbf{I_1}), \ldots, (\mathbf{A_n}, \mathbf{I_n})\}$ of couples $(\mathcal{AQ}, \mathcal{AQ})$ such that $\mathbf{I_i} \sqsubseteq \mathbf{A_i}$ and $i \neq j \Rightarrow \mathbf{A_i} \neq \mathbf{A_j}$. We also define a projection for abstracted interpretations $\pi_1(\mathcal{I}) = \{\mathbf{A} \mid (\mathbf{A}, \mathbf{I}) \in \mathcal{I}\}$.

An abstracted interpretation $\mathcal{I}$ represents for each $\mathbf{A_i}$ a set of possible computed instances in the form of the abstract conjunction $\mathbf{I_i}$. Let us now formulate how the knowledge contained in $\mathcal{I}$ can be used to refine some abstract conjunction, say $\mathbf{A}$ (not necessarily identical to some $\mathbf{A_i}$), into another abstract conjunction $\mathbf{A'} \sqsubseteq \mathbf{A}$ approximating the success set of $\mathbf{A}$.

Definition 3.2 Let $\mathcal{I} = \{(\mathbf{A_1}, \mathbf{I_1}), \ldots, (\mathbf{A_n}, \mathbf{I_n})\}$ be an abstracted interpretation. Let $\mathbf{A}$ be an abstract conjunction such that $\langle \mathbf{G_1}, \ldots \mathbf{G_j}, \ldots \mathbf{G_m} \rangle$ is an abstraction of $\mathbf{A}$. Also let $\mathbf{G_j} \sqsubseteq \mathbf{A_i}$ for some i. Then any abstract conjunction $\mathbf{A'} \sqsubseteq \mathbf{A}$ such that $\langle \mathbf{G_1}, \ldots \mathbf{G_{j-1}}, \mathbf{I_i}, \mathbf{G_{j+1}}, \ldots, \mathbf{G_m} \rangle$ is an abstraction of it, is called a *refinement of* $\mathbf{A}$ *under* $\mathcal{I}$. $\mathbf{A}$ itself, as well as any refinement of $\mathbf{A'}$, is also called a refinement of $\mathbf{A}$ under $\mathcal{I}$ (i.e. we take the transitive and reflexive closure). By $ref_{\mathcal{I}}(\mathbf{A})$ we denote some refinement of $\mathbf{A}$ under $\mathcal{I}$.

Example 3.3 In the $\mathcal{PD}$-domain let $\mathbf{A} = p(X) \wedge q(X)$ as well as $\mathcal{I} = \{(p(X), p(f(Y))),\ (q(f(Z)), q(f(a)))\}$. Then $\mathbf{A'} = p(f(V)) \wedge q(f(V))$ is a

refinement of $\mathbf{A}$ under $\mathcal{I}$. Now $\mathbf{A''} = p(f(a)) \wedge q(f(a))$ is in turn a refinement of $\mathbf{A'}$ (and thus also of $\mathbf{A}$) under $\mathcal{I}$. The previously inapplicable couple $(q(f(Z)), q(f(a)))$ became applicable for $\mathbf{A'}$ and allowed us to achieve further refinement.

We can now formulate an abstract bottom-up operator:

Definition 3.4 An *abstract bottom-up operator* $AT_P(.)$ is a function $\mathcal{AQ} \times 2^{\mathcal{AQ}\times\mathcal{AQ}} \mapsto \mathcal{AQ}$ such that, for every abstracted interpretation $\mathcal{I} = \{(\mathbf{A_1}, \mathbf{I_1}), \ldots, (\mathbf{A_n}, \mathbf{I_n})\}$, we have that if $H \leftarrow B \in \mathit{aunf}(\mathbf{A_i})$ and $\mathbf{B} = \mathit{ares}(\mathbf{A_i}, H \leftarrow B)$ then $B\theta \in \gamma(\mathit{ref}_I(\mathbf{B})) \Rightarrow H\theta \in \gamma(AT_P(\mathbf{A_i}, \mathcal{I}))$.

Intuitively, the above states that if a runtime resolvent ($\leftarrow B\theta$) of $\mathbf{A_i}$ may succeed given the abstracted interpretation $\mathcal{I}$ (i.e., $B\theta \in \gamma(\mathit{ref}_{\mathcal{I}}(\mathbf{B}))$) then the corresponding head $H\theta$ should potentially succeed in $AT_P(\mathbf{A_i}, \mathcal{I})$. In other words, $AT_P(\mathbf{A_i}, \mathcal{I})$ is a safe approximation of one concrete non-ground[6] bottom-up propagation step performed on the resultants of $\mathbf{A_i}$. We also define $AT_P(.)$ to work on abstracted interpretations: $AT_P(\mathcal{I}) = \{(\mathbf{A_1}, AT_P(\mathbf{A_1}, \mathcal{I})), \ldots, (\mathbf{A_n}, AT_P(\mathbf{A_n}, \mathcal{I}))\}$.

For an abstract domain with no infinite ascending chains, we can now formulate a terminating bottom-up analysis algorithm basically as calculating $AT_P \uparrow^{\infty} (\mathcal{I}_0)$ where $\mathcal{I}_0 = \{(\mathbf{A_1}, \bot), \ldots, (\mathbf{A_n}, \bot)\}$.

Exploiting Success Information for the Code Generation

The following shows how the information derivation by such an analysis can be exploited to derive a specialised program.

Definition 3.5 Let $\mathcal{I}$ be an abstracted interpretation. We then define an *abstract partial deduction of P wrt $\mathcal{I}$* to be the set of clauses:

$$\{\rho_{\mathbf{A}}(H\theta) \leftarrow \rho_{\mathcal{A},\mathbf{A''}}(B\theta) \mid H \leftarrow B \in \mathit{aunf}(\mathbf{A}) \wedge\ \mathbf{A} \in \pi_1(\mathcal{I}) \wedge\ \mathbf{A''} = \mathit{ref}_{\mathcal{I}}(\mathit{ares}(\mathbf{A}, H \leftarrow B)) \wedge\ B\theta \text{ is a concrete dominator}^7 \text{ of } \mathbf{A''}\}$$

If for all $\mathbf{A''}$ we have that $\{\mathbf{A''}\} \sqsubseteq_{\mathit{split}} \pi_1(\mathcal{I})$ then we call $\mathcal{I}$ *covered* (and all renamings $\rho_{\mathcal{A},\mathbf{A''}}(B\theta)$ above are defined).

The big difference over Definition 2.8 is that the resultants get instantiated using the success information contained in $\mathcal{I}$ and that the notion of coveredness also takes the success information into account. Indeed, $\mathcal{I}$ might be covered even though $\pi_1(\mathcal{I})$ is not (we might have $\{\mathbf{A'}\} \not\sqsubseteq_{\mathit{split}} \pi_1(\mathcal{I})$).

Example 3.6 Let P be the program from Example 2.2 and $\mathcal{I} = \{(eq(X,Y), eq(X,X))\}$ in the $\mathcal{PD}$-domain. Also, let $\mathit{aunf}(eq(X,Y)) = \{C_1, C_2\}$ and $\mathit{ares}(eq(X,Y), C_2) = eq(X,Y)$ with $C_1 = eq([],[]) \leftarrow$ as well as $C_2 = eq([H|S],[H|T]) \leftarrow eq(S,T)$. Then, obviously, $eq(X,X)$ is a refinement of $eq(X,Y)$ and the following is an abstract partial deduction of P wrt $\mathcal{I}$ (using $\rho_{eq(X,Y)} = eq_1(X,Y)$):

[6] $H\theta$ and $B\theta$ are not necessarily ground.

[7] One could also allow a set of instantiations $\theta_1, \ldots, \theta_n$ such that all concretisations of $\mathbf{A''}$ are an instance of at least one atom in $\{B\theta_1, \ldots, B\theta_n\}$. This can lead to more instantiated resultants but might also lead to code duplication and considerable slow-downs.

$eq_1([],[]) \leftarrow$
$eq_1([H|T],[H|T]) \leftarrow eq_1(T,T)$

The main technique of [21] can be seen an instance of this analysis by: taking the $\mathcal{PD}$-domain and using a composition of predicate-wise *msg* with the non-ground T_P operator on conjunctions. However, only a simple one-step unfolding is performed in [21] and it is not allowed to further refine refinements (which can be crucial, see [16]).

Theorem 3.7 Let P' be an abstract partial deduction of P wrt $AT_P \uparrow^\infty (\mathcal{I})$. Let $Q \in \gamma(\mathbf{A})$, $\mathbf{A} \in \pi_1(\mathcal{I})$, $AT_P \uparrow^\infty (\mathcal{I})$ be covered and *aunf*(.) be conservative.[8] Then $P \cup \{\leftarrow Q\}$ has an SLD-refutation with c.a.s. θ iff $P' \cup \{\leftarrow \rho_{\mathbf{A}}(Q)\}$ has.

For finite failure we can also derive that, if *aunf*(.) is fair then if $P \cup \{\leftarrow Q\}$ has a finitely failed SLD-refutation then so does $P' \cup \{\leftarrow \rho_{\mathbf{A}}(Q)\}$ (but not necessarily the other way around).

One major problem is now of course how to find interesting sets of abstract conjunctions $\mathcal{A} = \{\mathbf{A_1}, \ldots, \mathbf{A_n}\}$ (this was left open in [21]) as well as how to to ensure that $AT_P \uparrow^\infty (\{(\mathbf{A_1}, \bot), \ldots, (\mathbf{A_n}, \bot)\})$ is covered. Here the top-down framework can help, which in turn can benefit from the information provided by the bottom-up phase. This full integration is developed in the next section.

4 A Combined Top-Down/Bottom-Up Framework

The idea of the following algorithm is to combine the top-down with the bottom-up approach so that the mutually benefit from each other:

- the top-down component can, in addition to propagating goal dependent information downwards, provide interesting sets of abstract conjunctions for the bottom-up phase and ensure coveredness.
- the bottom-up phase can give the top-down component information about the global success-patterns, allowing a more focussed unfolding, producing more instantiated resultants as well as achieving side-ways information passing.

As shown in [16], for a particular abstract domain, such an integration can achieve optimisation and analysis which cannot be derived by either approach alone, nor by combining them in a naive manner (i.e. running them successively in isolation, as, e.g., discussed at the beginning of Section 3).

To formalise the flow analysis component of our integrated algorithm we define a refined abstract unfolding and resolution operator, which takes the current success information into account: $aiunf^*(\mathcal{I}) = \{(\mathbf{L}', \bot) \mid (\mathbf{A}, \mathbf{I}) \in \mathcal{I} \wedge$

[8] The difference with Theorem 2.9 is that calls to predicates are no longer guaranteed to be concretisations of the abstract conjunctions from which their definition has been derived; only their success patterns are ! Therefore the code also has to be sound (but not complete) for calls which are not concretisations. An alternative is to allow non-conservative rules but use the refinements only in, e.g., a left-to-right fashion.

$H \leftarrow B \in aunf(\mathbf{A}) \wedge \mathbf{L}' = ref_{\mathcal{I}}(ares(\mathbf{A}, H \leftarrow B))\}$. We also extend ω to abstracted interpretations and request that $\pi_1(\omega(\mathcal{I})) \sqsupseteq_{split} \pi_1(\mathcal{I})$.

Algorithm 4.1 (Refined Abstract Partial Deduction)
Input: A program P and an abstract conjunction $\mathbf{A}$
Output: A specialised program P'
Initialise: $i = 0$, $\mathcal{I}_0 = \{(\mathbf{A}, \bot)\}$
repeat
  let $j := i$; $\mathcal{I}_{i+1} := AT_P(\mathcal{I}_i)$; **let** $i := i + 1$; /* one BUP step */
  repeat
    let $\mathcal{I}_{i+1} := \omega(\mathcal{I}_i \cup aiunf^*(\mathcal{I}_i))$; **let** $i := i + 1$;
  until $\mathcal{I}_{i-1} = \mathcal{I}_i$
until $\mathcal{I}_j = \mathcal{I}_i$
Let P' be an abstract partial deduction wrt P and $\mathcal{I}_i$

One can easily see that, once the algorithm has terminated, $\mathcal{I}_i$ is covered. In fact, the inner repeat-loop — performing top-down abstract partial deduction — ensures (refined) coveredness. Also, abstract unfolding is applied after every single bottom-up step, i.e. *before* the fixpoint of $AT_P(.)$ is reached. This is the important aspect which makes this algorithm more powerful than running the top-down and bottom-up components in isolation (see [13] for a fully worked out example in the $\mathcal{PD}$-domain).

The algorithm in [16, 13] is an instance of the above algorithm using the $\mathcal{PD}$-domain and where $AT_P(.)$ is the predicate-wise *msg* composed with the non-ground T_P operator. Algorithm 4.1 is also strictly more powerful than [9, 6] (which uses the analysis information just to remove redundant clauses, and not, e.g., to instantiate them) or [22] (which cannot perform deforestation or tupling as it is restricted to specialising atoms individually). One can actually also express techniques based upon tabling (OLDT [12] or even EOLDT [2]) in a slight extension of our framework. In other words, the reconciliation of bottom-up and top-down evaluation [4] is then just a special case of our reconciliation of specialisation and analysis.

In [14] the reader can find how the *reverse-last* open problem from [16, 13] can be solved in our framework in a rather straightforward manner. Being able to solve this example is of prime relevance, e.g., whenever a statically known value is stored in a dynamic accumulator (e.g., an environment of an interpreter or a substitution in an explicit unification algorithm). No existing analysis, specialisation or transformation technique we are aware of, was able to solve this problem.

It is also possible to use the same approach to prove inductive theorems in a much less ad-hoc (and more generally reusable manner) than, e.g., [24]. We also believe that automation of this approach is feasible and endeavour to incorporate these possibilities into the ECCE partial deduction system in the not too distant future. We believe that the improved specialisation capabilities conferred by our new framework will further extend the practical applicability of program specialisation.

5 Conclusion

In this paper we have presented a generic framework and algorithm for top-down program specialisation, which supersedes earlier top-down approaches in generality and power. We have established a *generic correctness result* and have shown how the additional power can be exploited in practice, for improved generalisation, unfolding and code-generation. We have also clarified the relationship of top-down partial deduction with abstract interpretation, establishing a *common basis* and terminology. This clarification allowed us to precisely pinpoint shortcomings both of existing top-down specialisation methods and of existing abstract interpretation techniques. We then proceeded to remedy these shortcomings by incorporating bottom-up success information propagation, thereby fully *reconciling program specialisation with abstract interpretation* and providing a *unifying framework* into which almost all existing specialisation techniques can be cast. This new integrated framework with its generic algorithm provides the foundation for new, powerful specialisation *and* analysis outside the scope of existing techniques.

Acknowledgements

Maurice Bruynooghe provided very valuable feedback this paper. The author also greatly benefited from discussions with Danny De Schreye, Neil Jones, Jesper Jørgensen, Bern Martens, Torben Mogensen, Morten Heine Sørensen, and comments of anonymous referees.

References

[1] 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.

[2] 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.

[3] M. Bruynooghe. A practical framework for the abstract interpretation of logic programs. *The Journal of Logic Programming*, 10:91–124, 1991.

[4] F. Bry. Query evaluation in recursive databases: bottom-up and top-down reconciled. *Data and Knowledge Engineering*, 5:289–312, 1990.

[5] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *The Journal of Logic Programming*, 13(2 & 3):103–179, 1992.

[6] 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.

[7] J. Gallagher. Tutorial on specialisation of logic programs. In *Proceedings of PEPM'93*, pages 88–98. ACM Press, 1993.

[8] J. Gallagher and M. Bruynooghe. Some low-level transformations for logic programs. In M. Bruynooghe, editor, *Proceedings of Meta90*, pages 229–244, Leuven, Belgium, 1990.

[9] J. Gallagher and D. A. de Waal. Deletion of redundant unary type predicates from logic programs. In K.-K. Lau and T. Clement, editors, *Proceedings of LOPSTR'92*, pages 151–167, 1992.

[10] 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, September 1996. Springer-Verlag.

[11] N. D. Jones. Combining abstract interpretation and partial evaluation. In P. Van Hentenryck, editor, *Proceedings of SAS'97*, LNCS 1302, pages 396–405, 1997. Springer-Verlag.

[12] T. Kanamori and T. Kawamura. OLDT-based abstract interpretation. *The Journal of Logic Programming*, 15:1–30, 1993.

[13] M. Leuschel. *Advanced Techniques for Logic Program Specialisation.* PhD thesis, K.U. Leuven, May 1997.

[14] M. Leuschel. Program Specialisation and Abstract Interpretation Reconciled. Technical Report CW 259, K.U. Leuven, 1998.
Available at `http://www.cs.kuleuven.ac.be/~dtai`

[15] M. Leuschel and D. De Schreye. Constrained partial deduction and the preservation of characteristic trees. *New Generation Computing.* To Appear.

[16] M. Leuschel and D. De Schreye. Logic program specialisation: How to be more specific. In H. Kuchen and S. Swierstra, editors, *Proceedings of PLILP'96*, LNCS 1140, pages 137–151, September 1996. Springer-Verlag.

[17] 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 JICSLP'96*, pages 319–332, September 1996. MIT Press.

[18] 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.

[19] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.

[20] J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11(3& 4):217–242, 1991.

[21] K. Marriott, L. Naish, and J.-L. Lassez. Most specific logic programs. *Annals of Mathematics and Artificial Intelligence*, 1:303–338, 1990. Preliminary version in *Proceedings of JICSLP'88*, pages 909–923, 1988. MIT Press.

[22] 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, October 1997.

[23] G. Puebla and M. Hermenegildo. Abstract specialization and its application to program parallelization. In J. Gallagher, editor, *Proceedings of LOPSTR'96*, LNCS 1207, pages 169–186, August 1996.

[24] V. F. Turchin. Metacomputation: Metasystem transitions plus supercompilation. In O. Danvy, R. Glück, and P. Thiemann, editors, *Proceedings of the 1996 Dagstuhl Seminar on Partial Evaluation*, LNCS 1110, pages 482–509, 1996. Springer-Verlag.