Abstract Interpretation with Specialized Definitions

Germán Puebla[1], Elvira Albert[2], and Manuel Hermenegildo[1,3]

[1] School of Computer Science, Technical U. of Madrid, {german,herme}@fi.upm.es
[2] School of Computer Science, Complutense U. of Madrid, elvira@sip.ucm.es
[3] Depts. of Comp. Sci. and El. and Comp. Eng., U. of New Mexico, herme@unm.edu

Abstract. The relationship between abstract interpretation and partial deduction has received considerable attention and (partial) integrations have been proposed starting from both the partial deduction and abstract interpretation perspectives. In this work we present what we argue is the first fully described generic algorithm for efficient and precise integration of abstract interpretation and partial deduction. Taking as starting point state-of-the-art algorithms for context-sensitive, polyvariant abstract interpretation and (abstract) partial deduction, we present an algorithm which combines the best of both worlds. Key ingredients include the accurate success propagation inherent to abstract interpretation and the powerful program transformations achievable by partial deduction. In our algorithm, the calls which appear in the analysis graph are not analyzed w.r.t. the original definition of the procedure but w.r.t. *specialized definitions* of these procedures. Such specialized definitions are obtained by applying both unfolding and abstract executability. Our framework is parametric w.r.t. different control strategies and abstract domains. Different combinations of such parameters correspond to existing algorithms for program analysis and specialization. Simultaneously, our approach opens the door to the efficient computation of strictly more precise results than those achievable by each of the individual techniques. The algorithm is now one of the key components of the `CiaoPP` analysis and specialization system.

1 Introduction and Motivation

The relationship between abstract interpretation [3] and partial evaluation [11] has received considerable attention (see for example [5, 7, 2, 17, 10, 12, 21, 24, 6, 15, 4, 23, 13] and their references). In order to motivate and illustrate our proposal for an integration of abstract interpretation and partial evaluation, we use the running example of Fig. 1. It is a simple `Ciao` program which uses Peano's arithmetic.[1] We use the `Ciao` assertion language in order to provide precise descriptions on the initial call patterns. In our case, the `entry` declaration is used to inform that all calls to the only exported predicate (i.e., `main/2`) will always be of the form $\leftarrow$ `main`(`s`(`s`(`s`(`L`))), `R`) with `L` ground and `R` a variable. The predicate `main/2` performs two calls to predicate `formula/2`, which contains mode tests `ground(X)` and `var(W)` on its input arguments. A call `formula(X,W)` returns $W = (X - 2) \times 2$. Predicate `two/1` returns the natural number 2 in Peano's arithmetic. A call `minus(A,B,C)` returns $C = B - A$. However, if the result becomes a negative number, C is left as a free variable. This indicates that the result is not valid. In turn, a call `twice(A,B)` returns $B = A \times 2$. Prior to computing the result, this predicate checks whether A is valid, i.e., not a variable, and simply returns a variable otherwise.

[1] Rules are written with a unique subscript attached to the head atom (the rule number), and a dual subscript (rule number, body position) attached to each body literal. We sometimes use this notation for denoting calls to atoms as well.

```
:- module(_,[main/1],[assertions]).
:- entry main(s(s(s(L))),R) : (ground(L),var(R)).
main1(X,X2):- formula1,1(X,X1), formula1,2(X1,X2).
formula2(X,W):- ground2,1(X),var2,2(W),two2,3(T),minus2,4(T,X,X2),twice2,5(X2,W).
two3(s(s(0))).
minus4(0,X,X).
minus5(s(X),s(Y),R):- minus5,1(X,Y,R).
minus6(s(_X),0,_R).
twice7(X,_Y):- var7,1(X).
twice8(X,Y):- ground8,1(X), tw8,2(X,Y).
tw9(0,0).
tw10(s(X),s(s(NX))):- tw10,1(X,NX).
```

Fig. 1. Running Example

By observing the behaviour of the program it can be seen that for initial queries satisfying the `entry` declaration, all calls to the tests $\texttt{ground}_{2,1}\texttt{(X)}$ and $\texttt{var}_{2,2}\texttt{(W)}$ will definitely succeed, even if we do not know the concrete values of variable `L` at compile time. Also, the calls to $\texttt{ground}_{8,1}\texttt{(X)}$ will succeed, while the calls to $\texttt{var}_{7,1}\texttt{(X)}$ will fail. This shows the benefits of (1) *exploiting abstract information in order to abstractly execute certain atoms, which in turn may allow unfolding of other atoms*. However, the use of an abstract domain which captures groundness and freeness information will in general not be sufficient to determine that in the second execution of `formula/2` the tests $\texttt{ground}_{2,1}\texttt{(X)}$ and $\texttt{var}_{2,2}\texttt{(W)}$ will also succeed. The reason is that, on success of $\texttt{minus}_{2,4}\texttt{(T,X,X2)}$, `X2` cannot be guaranteed to be ground since $\texttt{minus}_6\texttt{/3}$ succeeds with a free variable on its third argument position. It can be observed, however, that for all calls to `minus/3` in executions described by the `entry` declaration, such third clause for `minus/3` is useless. It will never contribute to a success of `minus/3` since such predicate is always called with a value greater than zero on its second argument. Unfolding can make this explicit by fully unfolding calls to `minus/3` since they are sufficiently instantiated (and as a result the "dangerous" third clause is disregarded). It allows concluding that in our particular context, all calls to `minus/3` succeed with a ground third argument. This shows the importance of (2) *performing unfolding steps in order to prune away useless branches, which will result in improved success information*. By the time execution reaches $\texttt{twice}_{2,5}\texttt{(X2,W)}$, we hopefully know that `X2` is ground. In order to determine that, upon success of $\texttt{twice}_{2,5}\texttt{(X2,W)}$ (and thus on success of $\texttt{formula}_{1,1}\texttt{(X,W)}$), `W` is ground, we need to perform a fixpoint computation. Since, for example, the success substitution for $\texttt{formula}_{1,1}\texttt{(X,X1)}$ is indeed the call substitution for $\texttt{formula}_{1,2}\texttt{(X1,X2)}$, the success of the second test $\texttt{ground}_{2,1}\texttt{(X)}$ (i.e., the one reachable from $\texttt{formula}_{1,2}\texttt{(X1,X2)}$) cannot be established unless we propagate success substitutions. This illustrates the importance of (3) *propagating (abstract) success information, performing fixpoint computations when needed, which simultaneously will result in an improved unfolding*. Finally, whenever we call `formula(X,W)`, `W` is a variable, a property which cannot be captured if we restrict ourselves to downwards-closed domains. This indicates (4) *the usefulness of having information on non* downwards-closed *properties*.

Throughout the paper we show that the framework we propose is able to eliminate all calls to mode tests `ground/1` and `var/1`, and predicates `two/1` and `minus/3` are both fully unfolded and no longer appear in the residual code. We have used *sharing–freeness* as abstract domain instead of one based on, say regular types, for two reasons.[2] First, to illustrate how non-downwards closed information, including

[2] The values for the rest of parameters are: *AGeneralize* and *AUnfold* rules based on *homeomorphic embedding* [14], and the identity function as *Widen_Call* function.

freeness and definite independence, can be correctly exploited by our algorithm in order to optimize the program, and second, to show how unfolding can be of great use in order to improve the accuracy of analyses apparently unrelated to partial deduction, such as the classical *sharing–freeness*.

Example 1. The results obtained by `CiaoPP`—which implements abstract interpretation with specialized definitions—are both the following specialized code and an accurate analysis for such program (rules are renamed using the prefix `sp`).

```
sp_main1(s(s(s(0))),0).
sp_main2(s(s(s(s(B)))),A) :- sp_tw2,1(B,C), sp_formula2,2(C,A).
sp_tw2(0,0).
sp_tw3(s(A),s(s(B))) :- sp_tw3,1(A,B).
sp_formula4(0,s(s(s(s(0))))).
sp_formula5(s(A),s(s(s(s(s(B)))))) :- sp_tw5,1(A,B).
```

In this case, the success information for `sp_main(X,X2)` guarantees that `X2` is definitely ground on success. Note that this is equivalent to proving $\forall X \geq 3,\ main(X, X2) \rightarrow X2 \geq 0$. Furthermore, our system is able to get to that conclusion even if the `entry` only informs about `X` being any possible ground term and `X2` a free variable.

The above results cannot be achieved unless all four points mentioned before are available in a program analysis/specialization system. For example, if we use traditional partial deduction [19, 8] (PD) with the corresponding *Generalize* and *Unfold* rules followed by abstract interpretation and *abstract specialization* as described in [22, 23] we only obtain a comparable program after four iterations of the: "PD + abstract interpretation + abstract specialization" cycle. If we keep on adding more calls to `formula`, every time more iterations are necessary to obtain results comparable to ours. This shows the importance of achieving an algorithm which is able to *interleave* PD, with abstract interpretation, extended with abstract specialization, in order to communicate the accuracy gains achieved from one to the other as soon as possible. In any case, iterating over "PD + analysis" is not a good idea from the efficiency point of view. Also, sometimes partially evaluating a partially evaluated program can degrade the quality of the residual program.

The remaining of the paper is organized as follows. Sect. 2 recalls some preliminary concepts. In Sect. 3, we present abstract unfolding which already integrates abstract executability. Section 4 introduces our notion of specialized definition and embeds it within an abstract partial deducer. In Sect. 5, we propose our scheme for abstract interpretation with specialized definitions. Section 6 discusses how use interpret the results or our algorithm. Finally, Sect. 7 compares to related work and Sect. 8 concludes.

2 Preliminaries

Very briefly (see for example [18] for details), an *atom* A is a syntactic construction of the form $p(t_1, \ldots, t_n)$, where p/n, with $n \geq 0$, is a predicate symbol and $t_1, \ldots, t_n$ are terms. A *clause* is of the form $H \leftarrow B$ where its head H is an atom and its body B is a conjunction of atoms. A *definite program* is a finite set of clauses. A *goal* (or query) is a conjunction of atoms.

Let G be a goal of the form $\leftarrow A_1, \ldots, A_R, \ldots, A_k$, $k \geq 1$. The concept of *computation rule*, denoted by $\mathcal{R}$, is used to select an atom within a goal for its evaluation. The operational semantics of definite programs is based on derivations [18]. Let $C = H \leftarrow B_1, \ldots, B_m$ be a renamed apart clause in P such that $\exists\theta = mgu(A_R, H)$. Then $\leftarrow \theta(A_1, \ldots, A_{R-1}, B_1, \ldots, B_m, A_{R+1}, \ldots, A_k)$ is *derived* from G and C via $\mathcal{R}$.

As customary, given a program P and a goal G, an *SLD derivation* for $P \cup \{G\}$ consists of a possibly infinite sequence $G = G_0, G_1, G_2, \ldots$ of goals, a sequence $C_1, C_2, \ldots$ of properly renamed apart clauses of P, and a sequence $\theta_1, \theta_2, \ldots$ of mgus such that each G_{i+1} is derived from G_i and C_{i+1} using θ_{i+1}. A derivation step can be nondeterministic when A_R unifies with several clauses in P, giving rise to several possible SLD derivations for a given goal. Such SLD derivations can be organized in *SLD trees*. A finite derivation $G = G_0, G_1, G_2, \ldots, G_n$ is called *successful* if G_n is empty. In that case $\theta = \theta_1\theta_2 \ldots \theta_n$ is called the computed answer for goal G. Such a derivation is called *failed* if it is not possible to perform a derivation step with G_n. Given an atom A, an *unfolding rule* [19, 8] computes a set of finite SLD derivations $D_1, \ldots, D_n$ (i.e., a possibly incomplete SLD tree) of the form $D_i = A, \ldots, G_i$ with computed answer substitution θ_i for $i = 1, \ldots, n$ whose associated *resultants* (or residual rules) are $\theta_i(A) \leftarrow G_i$.

The following standard operations are used in the paper to handle keyed-tables: $\mathsf{Create\_Table}(T)$ initializes a table T. $\mathsf{Insert}(T, Key, Info)$ adds $Info$ associated to Key to T and deletes previous information associated to Key, if any. $\mathsf{Isln}(T, Key)$ returns true iff Key is currently stored in the table. Finally, $\mathsf{Look\_up}(T, Key)$ returns the information associated to Key in T. For simplicity, we sometimes consider tables as sets and we use the notation $(Key \rightsquigarrow Info) \in T$ to denote that there is an entry in the table T with the corresponding Key and associated $Info$.

2.1 Abstract Interpretation

Abstract interpretation [3] provides a general formal framework for computing safe approximations of programs behaviour. Programs are interpreted using values in an *abstract domain* (D_α) instead of the *concrete domain* (D). The set of all possible abstract values which represents D_α is usually a complete lattice or cpo which is ascending chain finite. Values in the abstract domain $\langle D_\alpha, \sqsubseteq \rangle$ and sets of values in the concrete domain $\langle 2^D, \subseteq \rangle$ are related via a pair of monotonic mappings $\langle \alpha, \gamma \rangle$: the *abstraction* function $\alpha : 2^D \rightarrow D_\alpha$ which assigns to each (possibly infinite) set of concrete values an abstract value, and the *concretization* function $\gamma : D_\alpha \rightarrow 2^D$ which assigns to each abstract value the (possibly infinite) set of concrete values it represents. The operations on the abstract domain D_α that we will use in our algorithms are:

- $\mathsf{Arestrict}(\lambda, E)$ performs the abstract restriction (or projection) of a substitution λ to the set of variables in the expression E, denoted $vars(E)$;
- $\mathsf{Aextend}(\lambda, E)$ extends the substitution λ to the variables in the set $vars(E)$;
- $\mathsf{Aunif}(t_1, t_2, \lambda)$ obtains the description which results from adding the abstraction of the unification $t_1 = t_2$ to the substitution λ;
- $\mathsf{Aconj}(\lambda_1, \lambda_2)$ performs the abstract conjunction ($\sqcap$) of two substitutions;
- $\mathsf{Alub}(\lambda_1, \lambda_2)$ performs the abstract disjunction ($\sqcup$) of two substitutions.

In our algorithms we also use $\mathsf{Atranslate}(A : CP, H \leftarrow B)$ which adapts and projects the information in an abstract atom $A : CP$ to the variables in the clause $C = H \leftarrow B$. This operation can be defined in terms of the operations above as: $\mathsf{Atranslate}(A : CP, H \leftarrow B) = \mathsf{Arestrict}(\mathsf{Aunif}(A, H, \mathsf{Aextend}(CP, C)), C)$. Finally, the most general substitution is represented as $\top$, and the least general (empty) substitution as $\bot$.

3 Unfolding with Abstract Substitutions

We now present an extension of SLD semantics which exploits abstract information. This will provide the means to overcome difficulties (1) and (2) introduced in Section 1. The extended semantics handles *abstract goals* of the form $G : CP$, i.e., a concrete goal G comes equipped with an *abstract substitution* CP which is defined over $vars(G)$ and provides additional information on the context in which the goal will be executed at run-time. The first rule corresponds to a derivation step.

Definition 1 (derivation step). *Let $G : CP$ be an abstract goal where $G = \leftarrow A_1, \ldots, A_R, \ldots, A_k$. Let $\mathcal{R}$ be a computation rule and let $\mathcal{R}(G) = A_R$. Let $C = H \leftarrow B_1, \ldots, B_m$ be a renamed apart clause in P. Then the abstract goal $G' : CP'$ is* derived *from $G : CP$ and C via $\mathcal{R}$ if $\exists\theta = mgu(A_R, H) \wedge CP_u \neq \bot$, where:*

$$CP_u = \mathsf{Aunif}(A_R, H\theta, \mathsf{Aextend}(CP, C\theta))$$
$$G' = \theta(A_1, \ldots, A_{R-1}, B_1, \ldots, B_m, A_{R+1}, \ldots, A_k)$$
$$CP' = \mathsf{Arestrict}(CP_u, vars(G'))$$

An important difference between the above definition and the standard derivation step is that the use of abstract (call) substitutions allows imposing further conditions for performing derivation steps, in particular, CP_u cannot be $\bot$. This is because if $CP \neq \bot$ and $CP_u = \bot$ then the head of the clause C is incompatible with CP and the unification $A_R = H$ will definitely fail at run-time. Thus, abstract information allows us to remove useless clauses from the residual program. This produces more efficient resultants and increases the accuracy of analysis for the residual code.

Example 2. Consider the abstract atom $\texttt{formula}(\texttt{s}^4(\texttt{X}), \texttt{X2}) : \{\texttt{X/G}, \texttt{X2/V}\}$, which appears in the analysis of our running example (c.f. Fig. 2). We abbreviate as $\texttt{s}^\texttt{n}(\texttt{X})$ the successive application of `n` functors `s` to variable `X`. The notation `X/G` (resp. `X/V`) indicates that variable `X` is ground (resp. a free variable). After applying a derivation step, we obtain the derived abstract goal:

$$\texttt{ground}(\texttt{s}^4(\texttt{X})), \texttt{var}(\texttt{X2}), \texttt{two}(\texttt{T}), \texttt{minus}(\texttt{T}, \texttt{s}^4(\texttt{X}), \texttt{X2}'), \texttt{twice}(\texttt{X2}', \texttt{X2}) : \{\texttt{X/G}, \texttt{X2/V}, \texttt{T/V}, \texttt{X2}'\texttt{/V}\}$$

where the abstract description has been extended with updated information about the freeness of the newly introduced variables. In particular, both `T` and `X2'` are `V`.

The second rule we present makes use of the availability of abstract substitutions to perform *abstract executability* [22] during resolution. This allows replacing some atoms with simpler ones, and, in particular, with the predefined atoms *true* and *false*, provided certain conditions hold. We assume the existence of a predefined *abstract executability table* which contains entries of the form $T : CP \rightsquigarrow T'$ which specify the behaviour of external procedures: builtins, libraries, and other user modules. For instance, for predicate `ground` contains the information $\texttt{ground(X)} : \{\texttt{X/G}\} \rightsquigarrow \texttt{true}$. For `var`, it contains $\texttt{var(X)} : \{\texttt{X/V}\} \rightsquigarrow \texttt{true}$.[3]

Definition 2 (abstract execution). *Let $G : CP$ be an abstract goal where $G = \leftarrow A_1, \ldots, A_R, \ldots, A_k$. Let $\mathcal{R}$ be a computation rule and let $\mathcal{R}(G) = A_R$. Let $(T : CP_T \rightsquigarrow T')$ be a renamed apart entry in the abstract executability table. Then, the goal $G' : CP'$ is* abstractly executed *from $G : CP$ and $(T : CP_T \rightsquigarrow T')$ via $\mathcal{R}$ if $A_R = \theta(T)$ and $CP_A \sqsubseteq CP_T$, where*

$$G' = A_1, \ldots, A_{R-1}, \theta(T'), A_{R+1}, \ldots, A_k$$
$$CP' = \mathsf{Arestrict}(CP, G')$$
$$CP_A = \mathsf{Atranslate}(A_R : CP, T \leftarrow true)$$

Example 3. From the derived goal in Ex. 2, we can apply twice the above rule to abstractly execute the calls to `ground` and `var` and obtain:

$$\texttt{two}(\texttt{T}), \texttt{minus}(\texttt{T}, \texttt{s}^4(\texttt{X}), \texttt{X2}'), \texttt{twice}(\texttt{X2}', \texttt{X2}) : \{\texttt{X/G}, \texttt{X2/V}, \texttt{T/V}, \texttt{X2}'\texttt{/V}\}$$

since both calls succeed by using the abstract executability table described above and the information in the abstract substitution.

Definition 3 (*AUnfold*). *Let $A : CP$ be an abstract atom and P a program. We define $AUnfold(P, A : CP)$ as the set of* resultants *associated to a finite (possibly incomplete) SLD tree computed by applying the rules of Definitions 1 and 2 to $A : CP$.*

[3] In `CiaoPP` we use assertions to express such information in a domain-independent manner.

The so-called *local control* of PD ensures the termination of the above process. For this purpose, the unfolding rule must incorporate some mechanism to stop the construction of SLD derivations (we refer to [14] for details).

Example 4. Consider an unfolding rule *AUnfold* based on homeomorphic embedding [14] to ensure termination and the initial goal in Ex. 2. The derivation continuing from Ex. 3 performs several additional derivation steps and abstract executions and branches (we do not include them due to space limitations and also because it is well understood). The following resultants are obtained from the resulting tree:

```
formula(s(s(s(s(0),s(s(s(s(0))))).
formula(s(s(s(s(s(A))))),s(s(s(s(s(s(B))))))) :- tw(A,B)
```

which will later be filtered and renamed resulting in rules 4 and 5 of Ex. 1.

It is important to note that SLD resolution with abstract substitutions is not restricted to the left-to-right computation rule. However, it is well-known that non-leftmost derivation steps can produce incorrect results if the goal contains *impure* atoms to the left of A_R. More details can be found, e.g., in [16]. Also, abstract execution of non-leftmost atoms can be incorrect if the abstract domain used captures properties which are not downwards closed. A simple solution is to only allow leftmost abstract execution for non-downwards closed domains (and non-leftmost for derivation steps).

4 Specialized Definitions

We now define an Abstract Partial Deduction (APD) algorithm whose execution can later be *interleaved* in a seamless way with a state-of-the-art abstract interpreter. For this it is essential that the APD process can generate residual code *online*. Thus, we need to produce a residual, specialized definition for a call pattern as soon as we finish processing it. This will make it possible for the analysis algorithm to have access to the improved definition. This may increase the accuracy of the analyzer and addresses the difficulty (2) described in Sect. 1.

Typically, PD is presented as an iterative process in which partial evaluations are computed for the new generated atoms until they *cover* all calls which can appear in the execution of the residual program. This is formally known as the *closedness* condition of PD [19]. In order to ensure termination of this global process, the so-called *global* control defines a *AGeneralize* operator (see [14]) which guarantees that the number of SLD trees computed is kept finite, i.e., it ensures the finiteness of the set of atoms for which partial evaluation is produced. However, the residual program is not generated until such iterative process terminates.

Algorithm 1 presents an APD algorithm. The main difference with standard algorithms is that the resultants computed by *AUnfold* (L26) are added to the program during execution of the algorithm (L30) rather than in a later code generation phase. In order to avoid conflicts among the new clauses and the original ones, clauses for specialized definitions are renamed with a fresh predicate name (L29) prior to adding them to the program (L30). The algorithm uses two global data structures. The *specialization table* contains entries of the form $A : CP \rightsquigarrow A'$. The atom A' provides the link with the clauses of the specialized definition for $A : CP$. The *generalization table* stores the results of the *AGeneralize* function and contains entries $A : CP \rightsquigarrow A' : CP$ where $A' : CP'$ is a generalization of $A : CP$.

Computation is initiated by procedure PARTIAL_EVALUATION_WITH_SPECS_DEFS (L1-4) which initializes the tables and calls PROCESS_CALL_PATTERN for each abstract atom $A_i : CP_i$ in the initial set to be partially evaluated. The task of PROCESS_CALL_PATTERN is, if the atom has not been processed yet (L6), to compute a

Algorithm 1 Abstract Partial Deduction with Specialized Definitions

```
1: procedure PARTIAL_EVALUATION_WITH_SPEC_DEFS(P, {A_1 : CP_1, ..., A_n : CP_n})
2:     Create_Table(GT); Create_Table(ST)
3:     for j = 1..n do
4:         PROCESS_CALL_PATTERN(A_j : CP_j)
5: procedure PROCESS_CALL_PATTERN(A : CP)
6:     if not IsIn(GT, A : CP) then
7:         (A_1, A'_1) ← SPECIALIZED_DEFINITION(P, A : CP)
8:         A_1 : CP_1 ← Look_up(GT, A : CP)
9:         for all ren. apart clause C_k = H_k ← B_k ∈ P s.t. H_k unifies with A'_1 do
10:            CP_k ← Atranslate(A'_1 : CP_1, C_k)
11:            PROCESS_CLAUSE(CP_k, B_k)
12: procedure PROCESS_CLAUSE(CP, B)
13:    if B = (L, R) then
14:        CP_L ← Arestrict(CP, L)
15:        PROCESS_CALL_PATTERN(L : CP_L)
16:        PROCESS_CLAUSE(CP, R)
17:    else
18:        CP_B ← Arestrict(CP, B)
19:        PROCESS_CALL_PATTERN(CP_B, B)
20: function SPECIALIZED_DEFINITION(P, A : CP)
21:    A' : CP' ← AGeneralize(ST, A : CP)
22:    Insert(GT, A : CP, A' : CP')
23:    if IsIn(ST, A' : CP') then
24:        A'' ←Look_up(ST, A' : CP')
25:    else
26:        Def ← AUnfold(P, A' : CP')
27:        A'' ← new_filter(A')
28:        Insert(ST, A' : CP', A'')
29:        Def' ← {(H' ← B) | (H ← B) ∈ Def ∧ H' = ren(H, {A'/A''})}
30:        P ← P ⋃ Def'
31:    return (A', A'')
```

specialized definition for it (L7) and then process all clauses in its specialized definition by means of calls to PROCESS_CLAUSE (L9-11). Procedure PROCESS_CLAUSE traverses clause bodies, processing their corresponding atoms by means of calls to PROCESS_CALL_PATTERN, in a depth-first, left-to-right fashion. The order in which pending call patterns (atoms) are handled by the algorithm is usually not fixed in PD algorithms. They are often all put together in a set. The reason for this presentation is to be as close as possible to our analysis algorithm which enforces a depth-first, left-to-right traversal of program clauses. In this regard, the relevant point to note is that this algorithm does not perform success propagation yet (difficulty 3). In L16, it becomes apparent that the atom(s) in R will be analyzed with the same call pattern CP than L, which is to their left in the clause. This, on one hand, may clearly lead to substantial precision loss. For instance, the abstract pattern $\mathtt{f(C,A)} : \{\mathtt{C/G}, \mathtt{C/V}\}$ which is necessary to obtain the last two resultants of Ex. 1 cannot be obtained with this algorithm. In particular, we cannot infer the groundness of `C` which, in turn, prevents us from abstractly executing the next call to `ground` and, thus, from obtaining this optimal specialization. On the other hand, this lack of success propagation makes it difficult or even impossible to work with non downwards closed domains, since CP may contain information which holds before execution of the leftmost atom L but which can be uncertain or even false after that. In fact, in our example CP contains the info `C/V`, which becomes false after execution of `tw(B,C)`, since now `C` is ground.

This problem is solved in the algorithm we present in the next section, where analysis information flows from left to right, adding more precise information and eliminating information which is no longer safe or even definitely wrong.

For the integration we propose, the most relevant part of the algorithm comprises L20-31, as it is the code fragment which is *directly* executed from our abstract interpreter. The remaining procedures (L1-L19) will be overridden by more accurate ones later. The procedure of interest is SPECIALIZED_DEFINITION. As it is customary, it performs (L21) a generalization of the call $A : CP$ using the abstract counterpart of the *Generalize* operator, denoted by *AGeneralize*, and which is in charge of ensuring termination at the global level. The result of the generalization, $A' : CP'$, is inserted in the generalization table $\mathcal{GT}$ (L22). Correctness of the algorithm requires that $A : CP \sqsubseteq A' : CP'$. If $A' : CP'$ has been previously treated (L23), then its specialized definition A'' is looked up in $\mathcal{ST}$ (L24) and returned. Otherwise, a specialized definition *Def* is computed for it by using the *AUnfold* operator of Def. 3 (L26). As already mentioned, the specialized definition *Def* for the abstract atom $A : CP$ is used to extend the original program P. First, the atom A' is renamed by using new_filter which returns an atom with a fresh predicate name, A'', and optionally filters constants out (L27). Then, function ren is applied to rename the clause heads using atom A' (L29). The function $\mathsf{ren}(A, \{B/B'\})$ returns $\theta(B')$ where $\theta = mgu(A, B)$. Finally, the program P is extended with the new, *renamed* specialized definition, *Def*$'$.

Example 5. Three calls to SPECIALIZED_DEFINITION appear (within an oval box) during the analysis of our running example in Fig. 2 from the following abstract atoms, first $\mathtt{main}(\mathtt{s}^3(\mathtt{X}), \mathtt{X2}) : \{\mathtt{X/G}, \mathtt{X2/V}\}$, then $\mathtt{tw}(\mathtt{B}, \mathtt{C}) : \{\mathtt{B/G}, \mathtt{C/V}\}$ and finally $\mathtt{f}(\mathtt{C}, \mathtt{A}) : \{\mathtt{C/G}, \mathtt{C/V}\}$. The output of such executions is used later (with the proper renaming) to produce the resultants in Ex. 1. For instance, the second clause obtained from the first call to SPECIALIZED_DEFINITION is

$\mathtt{sp\_main}_2\mathtt{(s(s(s(s(B)))),A)}$:- $\mathtt{tw}_{2,1}\mathtt{(B,C)}$,$\mathtt{formula}_{2,2}\mathtt{(C,A)}$.

where only the head is renamed. The renaming of the body literals is done in a later code generation phase (see Section 6.1). As already mentioned, Alg. 1 is not able to obtain such abstract atoms due to the absence of success propagation.

5 Abstract Interpretation with Specialized Definitions

We now present our final algorithm for abstract interpretation with specialized definitions. This algorithm extends both the APD Algorithm 1 and the abstract interpretation algorithms in [20, 9]. W.r.t. Algorithm 1, the main improvement is the addition of success propagation. Unfortunately, this requires computing a global fixpoint. It is an important objective for us to be able to compute an accurate fixpoint in an efficient way. W.r.t the algorithms in [20, 9], the main improvements are the following. (1) It deals directly with non-normalized programs. This point, which does not seem very relevant in a pure analysis system, becomes crucial when combined with a specialization system in order to profit from constants propagated by unfolding. (2) It incorporates a hardwired efficient graph traversal strategy which eliminates the need for maintaining priority queues explicitly [9]. (3) The algorithm includes a widening operation for calls, *Widen_Call*, which limits the amount of multivariance in order to keep finite the number of call patterns analyzed. This is required in order to be able to use abstract domains which are infinite, such as regular types. (4) It also includes a number of simplifications to facilitate understanding, such as the use of the keyed-table ADT, which we assume encapsulates proper renaming apart of variables and the application of renaming transformations when needed.

In order to compute and propagate success substitutions, Algorithm 2 computes a *program analysis graph* in a similar fashion as state of the art analyzers such as

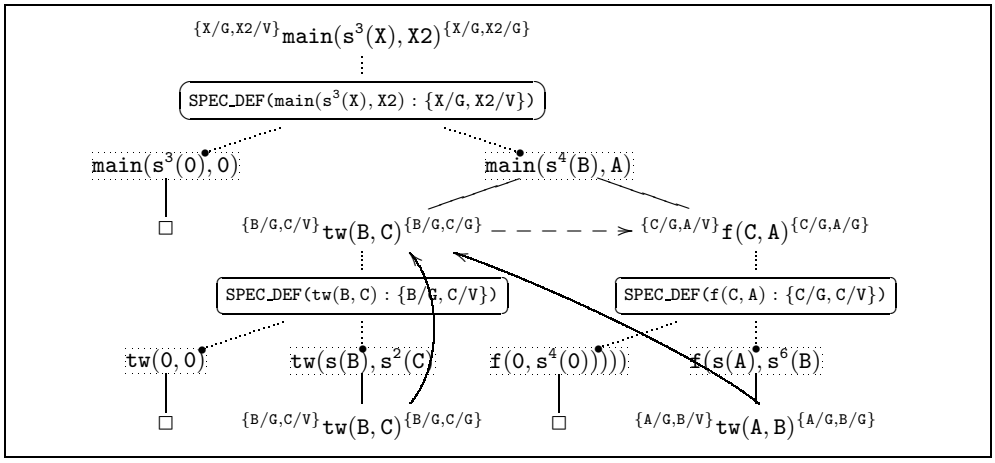

Fig. 2. Analysis Graph computed by ABS_INT_WITH_SPEC_DEF

the CiaoPP analyzer [20, 9]. For instance, the analysis graph computed by Algorithm 2 for our running example is depicted in Fig. 2. The graph has two sorts of nodes. Those which correspond to atoms are called "OR-nodes". For instance, the node ${}^{\{\mathtt{X/G,X2/V}\}}\mathtt{main}(\mathtt{s}^3(\mathtt{X}),\mathtt{X2})^{\{\mathtt{X/G,X2/G}\}}$ indicates that when the atom $\mathtt{main}(\mathtt{s}^3(\mathtt{X}),\mathtt{X2})$ is called with description $\{\mathtt{X/G},\mathtt{X2/V}\}$ the answer (or success) substitution computed is $\{\mathtt{X/G},\mathtt{X2/G}\}$. Those nodes which correspond to rules are called "AND-nodes". In Fig. 2, they appear within a dashed box and contain the head of the corresponding clause. Each AND-node has as children as many OR-nodes as literals there are in the body. If a child OR-node is already in the tree, it is no further expanded and the currently available answer is used. For instance, the analysis graph in Figure 2 contains three occurrences of the abstract atom $\mathtt{tw}(\mathtt{B},\mathtt{C}) : \{\mathtt{B/G},\mathtt{C/V}\}$ (modulo renaming), but only one of them has been expanded. This is depicted by arrows from the two non-expanded occurrences of $\mathtt{tw}(\mathtt{B},\mathtt{C}) : \{\mathtt{B/G},\mathtt{C/V}\}$ to the expanded one. More information on the efficient construction of the analysis graph can be found in [20, 9, 1].

The program analysis graph is implicitly represented in the algorithm by means of two data structures, the *answer table* ($\mathcal{AT}$) and the *dependency table* ($\mathcal{DT}$). The answer table contains entries of the form $A : CP \rightsquigarrow AP$ which are interpreted as the answer (success) pattern for $A : CP$ is AP. For instance, there exists an entry of the form $\mathtt{main}(\mathtt{s}^3(\mathtt{X}),\mathtt{X2}) : \{\mathtt{X/G},\mathtt{X2/V}\} \rightsquigarrow \{\mathtt{X/G},\mathtt{X2/G}\}$ associated to the atom discussed above. Dependencies indicate direct relations among OR-nodes. An OR-node $A_F : CP_F$ *depends on* another OR-node $A_T : CP_T$ iff in the body of some clause for $A_F : CP_F$ there appears the OR-node $A_T : CP_T$. The intuition is that in computing the answer for $A_F : CP_F$ we have used the answer pattern for $A_T : CP_T$. In our algorithm we store *backwards* dependencies,[4] i.e., for each OR-node $A_T : CP_T$ we keep track of the set of OR-nodes which depend on it. That is to say, the keys in the dependency table are OR-nodes and the information associated to each node is the set of other nodes which depend on it, together with some additional information required to iterate when an answer is modified (updated). Each element of a *dependency set* for an atom $B : CP_2$ is of the form $\langle H : CP \Rightarrow [H_k : CP_1]\ k, i\rangle$. It should be interpreted as follows: the OR-node $H : CP$ through the literal at position k, i depends on the OR-node $B : CP_2$. Also, the remaining information $[H_k : CP_1]$ informs that the head of this clause is H_k and the substitution (in terms of all variables of clause k) just before

[4] In the implementation, for efficiency, both forward and backward dependencies are stored. We do not include them in the algorithm for simplicity of the presentation.

Algorithm 2 Abstract Interpretation with Specialized Definitions

```
1: procedure ABS_INT_WITH_SPEC_DEFS(P, {A_1 : CP_1, ..., A_n : CP_n})
2:     Create_Table(AT); Create_Table(DT)
3:     Create_Table(GT); Create_Table(ST)
4:     for j = 1..n do
5:         PROCESS_CALL_PATTERN(A_j : CP_j, ⟨A_j : CP_j ⇒ [A_j : CP_j], j, entry⟩)
6: function PROCESS_CALL_PATTERN(A : CP, Parent)
7:     CP_1 ← Widen_Call(AT, A : CP)
8:     if not IsIn(AT, A : CP_1) then
9:         Insert(AT, A : CP_1, ⊥)
10:        Insert(DT, A : CP_1, ∅)
11:        (A', A'_1) ← SPECIALIZED_DEFINITION(P, A : CP_1)
12:        A'' ← ren(A, {A'/A'_1})
13:        for all ren. apart clause C_k = H_k ← B_k ∈ P s.t. H_k unifies with A'' do
14:            CP_k ← Atranslate(A'' : CP_1, C_k)
15:            PROCESS_CLAUSE(A : CP_1 ⇒ [H_k : CP_k] B_k, k, 1)
16:    Deps ← Look_up(DT, A : CP_1) ⋃ {Parent}
17:    Insert(DT, A : CP_1, Deps)
18:    return Look_up(AT, A : CP_1)
19: procedure PROCESS_CLAUSE(H : CP ⇒ [H_k : CP_1] B, k, i)
20:    if CP_1 ≠ ⊥ then
21:        if B = (L, R) then
22:            CP_2 ← Arestrict(CP_1, L)
23:            AP_0 ← PROCESS_CALL_PATTERN(L : CP_2, ⟨H : CP ⇒ [H_k : CP_1], k, i⟩)
24:            CP_3 ← Aconj(CP_1, Aextend(AP_0, CP_1))
25:            PROCESS_CLAUSE(H : CP ⇒ [H_k : CP_3] R, k, i + 1)
26:        else
27:            CP_2 ← Arestrict(CP_1, B)
28:            AP_0 ← PROCESS_CALL_PATTERN(B : CP_2, ⟨H : CP ⇒ [H_k : CP_1], k, i⟩)
29:            CP_3 ← Aconj(CP_1, Aextend(AP_0, CP_1))
30:            AP_1 ← Atranslate(H_k : CP_3, H ← true)
31:            AP_2 ← Look_up(AT, H : CP)
32:            AP_3 ← Alub(AP_1, AP_2)
33:            if AP_2 ≠ AP_3 then
34:                Insert(AT, H : CP, AP_3)
35:                Deps ← Look_up(DT, H : CP)
36:                PROCESS_UPDATE(Deps)
37: procedure PROCESS_UPDATE(Updates)
38:    if Updates = {A_1, ..., A_n} with n ≥ 0 then
39:        A_1 = ⟨H : CP ⇒ [H_k : CP_1], k, i⟩
40:        if i ≠ entry then
41:            B ← get_body(P, k, i)
42:            REMOVE_PREVIOUS_DEPS(H : CP ⇒ [H_k : CP_1] B, k, i)
43:            PROCESS_CLAUSE(H : CP ⇒ [H_k : CP_1] B, k, i)
44:            PROCESS_UPDATE(Updates − {A_1})
```

the call to $B : CP_2$ is CP_1. Such information avoids reprocessing atoms in the clause k to the left of position i. For instance, the dependency set for $\mathtt{f(C,A)} : \{\mathtt{A/V}, \mathtt{C/G}\}$ is $\{\langle \mathtt{main(s^3(X),X2)} : \{\mathtt{X/G}, \mathtt{X2/V}\} \Rightarrow [\mathtt{main(s^4(B),A)} : \{\mathtt{B/G}, \mathtt{A/V}, \mathtt{C/G}\}]2, 2\rangle\}$. It indicates that the OR-node $\mathtt{f(C,A)} : \{\mathtt{A/V}, \mathtt{C/G}\}$ is only used in the OR-node $\mathtt{main(s^3(X),X2)} : \{\mathtt{X/G}, \mathtt{X2/V}\}$ via literal 2,2 (see Example 1). Thus, if the answer pattern for $\mathtt{f(C,A)} : \{\mathtt{A/V}, \mathtt{C/G}\}$ is ever updated, then we must reprocess the OR-node $\{\mathtt{main(s^3(X),X2)} : \{\mathtt{X/G}, \mathtt{X2/V}\}$ from position 2,2.

Algorithm 2 proceeds as follows. The procedure ABS_INT_WITH_SPEC_DEFS initializes the four tables used by the algorithm and calls PROCESS_CALL_PATTERN for

each abstract atom in the initial set. The function PROCESS_CALL_PATTERN applies, first of all (L7), the *Widen_Call* function to $A : CP$ taking into account the set of entries already in $\mathcal{AT}$. This returns a substitution CP_1 s.t. $CP \sqsubseteq CP_1$. The most precise *Widen_Call* function possible is the identity function, but it can only be used with abstract domains with a finite number of abstract values. This is the case with *sharing–freeness* and thus we will use the identity function in our example. If the call pattern $A : CP_1$ has not been processed before, it places (L9) $\bot$ as initial answer in $\mathcal{AT}$ for $A : CP$ and sets to empty (L10) the set of OR-nodes in the graph which depend on $A : CP_1$. It then computes (L11) a specialized definition for $A : CP_1$. We do not show in Algorithm 2 the definition of SPECIALIZED_DEFINITION, since it is identical to that in Algorithm 1. In the graph, we show within an oval box the calls to SPECIALIZED_DEFINITION which appear during the execution of the running example (see the details in Sect. 4). The clauses in the specialized definition are linked to the box with a dotted arc. Then it launches (L13-15) calls to PROCESS_CLAUSE for the clauses in the specialized definition w.r.t. which $A : CP_1$ is to be analyzed. Only after this, the *Parent* OR-node is added (L16-17) to the dependency set for $A : CP_1$.

The function PROCESS_CLAUSE performs the success propagation and constitutes the core of the analysis. First, the current answer (AP_0) for the call to the literal at position k, i of the form $B : CP_2$ is (L24 and L29) conjoined (Aconj), after being extended (Aextend) to all variables in the clause, with the description CP_1 from the program point immediately before B in order to obtain the description CP_3 for the program point after B. If B is not the last literal, CP_3 is taken as the (improved) calling pattern to process the next literal in the clause in the recursive call (L25). This corresponds to left-to-right success propagation and is marked in Fig. 2 with a dashed horizontal arrow. If we are actually processing the last literal, CP_3 is (L30) adapted (Atranslate) to the initial call pattern $H : CP$ which started PROCESS_CLAUSE, obtaining AP_1. This value is (L32) disjoined (Alub) with the current answer, AP_2, for $H : CP$ as given by Look_up. If the answer changes, then its dependencies, which are readily available in $\mathcal{DT}$, need to be recomputed (L36) using PROCESS_UPDATE. This procedure restarts the processing of all body postfixes which depend on the calling pattern for which the answer has been updated by launching new calls to PROCESS_CLAUSE. There is no need of recomputing answers in our example. The procedure REMOVE_PREV_DEPS eliminates (L42) entries in $\mathcal{DT}$ for the clause postfix which is about to be re-computed. We do not present its definition here due to lack of space. Note that the new calls to PROCESS_CLAUSE may in turn launch calls to PROCESS_UPDATE. On termination of the algorithm a global fixpoint is guaranteed to have been reached. Note that our algorithm also stores in the dependency sets calls from the initial entry points (marked with the value *entry* in L5). These do not need to be reprocessed (L40) but are useful for determining the specialized version to use for the initial queries after code generation.

5.1 Termination of Abstract Interpretation with Specialized Definitions

Termination of Algorithm 2 comprises several levels. First, termination of the algorithm requires the local termination of the process of obtaining a specialized definition. This corresponds to ensuring termination of function SPECIALIZED_DEFINITION in Algorithm 1. Second, we need to guarantee that the number of call patterns for which a specialized definition is computed is finite. This corresponds to global termination of specialization algorithms. In terms of our algorithm, this is equivalent to having a finite number of entries in $\mathcal{ST}$. The *AGeneralize* function should be able to guarantee it. Third, it is required that the set of call patterns for which an answer pattern is to be computed be finite. This corresponds to control of multivariance in context-sensitive analysis. In terms of our algorithm, this is equivalent to having a finite number of en-

tries in $\mathcal{AT}$. The *Widen_Call* function should be able to guarantee it. Fourth and final, it is required that the computation of the answer pattern for each entry in $\mathcal{AT}$ needs a finite number of iterations. This is guaranteed since we consider domains which are ascending chain finite. Another way of looking at this problem is that, intuitively, the combined effect of terminating *AUnfold* and *AGeneralize* operators guarantee that the set of specialized definitions which Algorithm 2 will compute for an initial set of atoms is finite. These two problems have received considerable attention by the PD community (see, e.g., [14]). Since Algorithm 2 performs analysis of the program composed of the set of specialized definitions, once we have guaranteed the finiteness of the program to be analyzed, a terminating *Widen_Call* together with an abstract domain which is ascending chain finite guarantee termination of the whole process.

6 Interpreting the Results of the Algorithm

We first discuss whether we can interpret the results of Algorithm 2 in terms of analysis. We denote by $success(A : CP, P)$ the set of computed answers for initial queries described by the abstract atom $A : CP$ in a program P, i.e., $success(A : CP, P) = \{\theta'' \mid \exists\theta \in \gamma(CP) \wedge \exists\theta' \text{ s.t. } \theta' \text{ is a computed answer for } A\theta \text{ and } \theta'' = \theta\theta'|_{vars(A)}\}$.

Theorem 1 (correctness of success substitutions). *Let P be a program and let $S = \{A_1 : CP_1, \ldots, A_n : CP_n\}$ be a set of abstract atoms. After termination of* ABS_INT_WITH_SPEC_DEFS(P, S), $\forall A_i : CP_i \in S \;.\; \exists(A_i : CP'_i \rightsquigarrow AP_i) \in \mathcal{AT}$ *s.t.* $CP_i \sqsubseteq CP'_i \wedge success(A_i : CP_i, P) \subseteq \gamma(AP_i)$.

Intuitively, correctness holds since Algorithm 2 computes an abstract and–or graph and, thus, we inherit a generic correctness result for success substitutions. However, now we analyze the call patterns in S w.r.t. specialized definitions rather than their original definition in P. Since the transformation rules in Definitions 1 and 2 are semantics preserving, then analysis of each specialized definition is guaranteed to produce a safe approximation of its success set, which is also a safe approximation of the success of the original definition.

6.1 The Framework as a Specializer

We now discuss whether the set of specialized definitions can be used as a specialized program. Algorithm 2 differs from Algorithm 1 in several ways. First, the specialized definition for $A : CP$ (L11) is not computed w.r.t the original call pattern $A : CP$ but rather w.r.t. $A : CP_1$, where $CP_1 = Widen\_Call(\mathcal{AT}, A : CP)$. This is required in order to guarantee termination of the analysis side of the algorithm since, as we discuss below, analysis is multivariant in nature and the same specialized definition can be analyzed for (a possibly infinite) number of different abstract atoms. Second, the abstract substitution CP_k used to process clause k (L14) is not induced by the abstract atom returned from $AGeneralize(\mathcal{ST}, A : CP)$, as in Algorithm 1 (which would limit the analysis side to be monovariant on each specialized definition), but rather it is induced by $A : CP_1$. This on one hand enables a higher degree of multivariance in analysis, and thus more accurate results, and on the other hand poses more difficulties in interpreting the results of the algorithm as a specialized program.

Let us now formulate the conditions under which it is *feasible* to perform code generation on the results of Algorithm 2. We use $\langle P', \mathcal{AT}, \mathcal{DT}, \mathcal{GT}, \mathcal{ST}\rangle = AISP(P, S)$ to denote that on termination of Algorithm 2 for a program P and a set of atoms S we have obtained a program P' and tables $\mathcal{AT}$, $\mathcal{DT}$, $\mathcal{GT}$, and $\mathcal{ST}$. We denote by $spec\_defs(P, \mathcal{ST})$ the subset of clauses in P which correspond to specialized definitions, as stored in $\mathcal{ST}$ and we define it as $spec\_defs(P, \mathcal{ST}) = \{(H \leftarrow B) \in P \mid \exists(\_ : \_ \rightsquigarrow A') \in \mathcal{ST} \text{ s.t. } H \text{ unifies with } A'\}$. Each non-root OR-node in the

analysis graph has been generated by a call of the form PROCESS_CALL_PATTERN($B : CP_2, \langle H : CP \Rightarrow [H_k : CP_1], k, i\rangle$) (L23 or L28 in Algorithm 2). Thus, each non-root OR-node is uniquely identified by a pair of the form $(B : CP_2, \langle H : CP \Rightarrow [\_ : \_], k, i\rangle)$. We can classify the OR-nodes in an analysis graph according to the program point they correspond to, i.e., k, i. We denote by $OR\_nodes(k, i)$ the set of OR-nodes of the form $(\_ : \_, \langle \_ : \_ \Rightarrow [\_ : \_], k, i\rangle)$. Also, we denote by $SD((B : CP_2, Id), \mathcal{DT}, \mathcal{GT})$ the abstract atom $B' : CP'_2$ which has been used for generating the specialized definition w.r.t. which the atom $(B : CP_2, Id)$ has been analyzed, and it is defined as $SD((B : CP_2, \langle H : CP \Rightarrow [\_ : \_], k, i\rangle), \mathcal{DT}, \mathcal{GT}) = B' : CP'_2$ s.t. $\exists (B : CP_1 \rightsquigarrow Deps) \in \mathcal{DT}$ s.t. $(H : CP \Rightarrow [\_ : \_], k, i) \in Deps \wedge \exists (B : CP_1 \rightsquigarrow B' : CP'_2) \in \mathcal{GT}$

Definition 4 (feasible specialized program). *Let P be a program and S a set of atoms. Let $\langle P', \mathcal{AT}, \mathcal{DT}, \mathcal{GT}, \mathcal{ST}\rangle = AISP(P, S)$. The program spec_defs$(P', \mathcal{ST})$ is* feasible *iff* $\forall k, i \in spec\_defs(P', \mathcal{ST})\ .\ \forall (B : CP_2^1, Id^1), (B : CP_2^2, Id^2) \in OR\_nodes(k, i)$

$$\begin{aligned} SD((B : CP_2^1, Id^1), \mathcal{DT}, \mathcal{GT}) = B^{1'} : CP_2^{1'} \ \wedge \\ SD((B : CP_2^2, Id^2), \mathcal{DT}, \mathcal{GT}) = B^{2'} : CP_2^{2'} \ \wedge \\ B : CP_2^1 \sqsubseteq B^{2'} : CP_2^{2'} \ \wedge \\ B : CP_2^2 \sqsubseteq B^1 : CP_2^{1'} \end{aligned}$$

Correctness of *Widen_Call* and *AGeneralize* guarantee than any call pattern $B : CP_2$ is analyzed w.r.t. a specialized definition which is correct, i.e., for any possible $\mathcal{DT}$ and $\mathcal{GT}$, $B : CP_2 \sqsubseteq SD((B : CP_2, Id), \mathcal{DT}, \mathcal{GT})$. This condition is indeed sufficient for correctness of the analysis. However, Definition 4 requires that whenever more than one OR-node correspond to the same program point k, i, the specialized definition w.r.t which each OR-node has been analyzed is a safe approximation not only of itself but also of all the OR-nodes for k, i. In spite of the complexity of the formulation, it is relatively easy to find *AGeneralize* functions which guarantee that the resulting specialized program will be feasible.

Example 6. We present two *AGeneralize* functions which can be used in Alg. 2 to ensure a feasible specialized program. In both of them, the decision on whether to lose information in a call *AGeneralize*$(\mathcal{ST}, A : CP)$ is based on the concrete part of the atom, A. This allows easily defining *AGeneralize* operators in terms of existing *Generalize* operators. Let *Generalize* be a (concrete) generalization function. Let $AGeneralize_\alpha(\mathcal{ST}, A : CP) = (A', CP')$ where $A' = Generalize(\mathcal{ST}, A)$ and $CP' = \mathsf{Atranslate}(A : CP, A' \leftarrow true)$. Function $AGeneralize_\alpha$ always produces feasible programs since it only assigns the same specialized definition for different abstract atoms when we know that after adapting the analysis info of both $A_1 : CP_1$ and $A_2 : CP_2$ to the new atom A' the same entry substitution CP' will be obtained in either case. Similarly, we define $AGeneralize_\gamma(\mathcal{ST}, A : CP) = (A', CP')$ where $A' = Generalize(\mathcal{ST}, A)$ and $CP' = \top$. The function $AGeneralize_\gamma$ is also correct since it assigns generalizations taking into account the concrete part of the abstract atom only, which is the same for all OR-nodes which correspond to a literal k, i. These functions are in fact two extremes. In $AGeneralize_\alpha$ we try to keep as much abstract information as possible, whereas in $AGeneralize_\gamma$ we lose all abstract information. The latter is useful when we do not have an unfolding system which can exploit abstract information or when we do not want the specialized program to have different implemented specialized definitions for atoms with the same concrete part (modulo renaming) but different abstract substitution.

Now, code generation for a feasible program is done by simply traversing the bodies of the specialized definitions and for each literal L at position k, i we find out,

using function SD, an atom $A' : CP'$ whose specialized definition we can use at k, i. Then, we look up in $\mathcal{ST}$ the atom A' w.r.t. which to rename the body literal L. A code generation algorithm can be found in the appendix (Algorithm 3).

Finally, since Algorithm 2 computes a fixpoint, the answer and specialization tables may contain entries which correspond to *spurious* call patterns. It is however straightforward to remove them.

7 Discussion and Related Work

We have presented a generic framework for the analysis and specialization of logic programs which is currently the basis of the analysis/specialization system implemented in the `CiaoPP` preprocessor. We argue that, in contrast to other approaches, the fact that our method can be used both as a specializer and analyzer gives us more accuracy and efficiency than the individual techniques. Indeed, the versatility of our framework (and of our implementation) can be seen by recasting well-known specialization and analysis frameworks as instances in which the different parameters: unfolding rule, widen call rule, abstraction operator, and analysis domain, take the following values.

Polyvariant Abstract Interpretation: Our algorithm can behave as the analysis algorithm described in [9, 20] for polyvariant static analysis by defining a *AGeneralize* operator which returns always the base form of an expression (i.e., it loses all constants) and an *AUnfold* operator which performs a single derivation step (i.e., it returns the original definition). Thus, the resulting framework would always produce a residual program which coincides with the original one and can be analyzed with any abstract domain of interest.

Multivariant Abstract Specialization: The specialization power of the framework described in [23, 22] can be obtained by using the same *AGeneralize* described in the above point plus an *AUnfold* operator which always performs a derive step followed by zero or more abstract execution steps. It is interesting to note that in the original framework abstract executability is performed as an offline optimization phase while it is performed online in our framework.

Classical Partial Deduction: Our method can be used to perform classical PD in the style of [19, 8] by using an abstract domain with the single abstract value $\top$ and the identity function as *Widen_Call* rule. This corresponds to the $\mathcal{PD}$ domain of [13] in which an atom with variables represents all its instances. Let us note that, in spite of the fact that the algorithm follows a left-to-right computation flow, the process of generating specialized definitions (as discussed in Section 3) can perform *non-leftmost* unfolding steps and achieve optimizations as powerful as in PD.

Abstract Partial Deduction: Several approaches have been proposed which extend PD by using abstract substitutions [12, 6, 15, 13]. In essence, such approaches are very similar to the abstract partial deduction with call propagation shown in Algorithm 1. Though all those proposals identify the need of propagating success substitutions, they either fail to do so or propose means for propagating success information which are not fully integrated with the APD algorithm and, in our opinion, do not fit in as nicely as the use of and–or trees. Also, these proposals are either strongly coupled to a particular (downward closed) abstract domain, i.e., regular types, as in [6, 15] or do not provide the exact description of operations on the abstract domain which are needed by the framework, other than general correctness criteria [12, 13]. However, the latter allow conjunctive PD, which is not available in our framework.

The approach in [24]: was a starting step towards our current framework. There, the introduction of unfolding steps directly in the and–or graph was proposed in order to achieve transformations as powerful as those of PD while at the same time propagating abstract information. In contrast, we now resort to augmented SLD semantics for the specialization side of the framework while using AND-OR semantics for the analysis side of the framework. This has both conceptual, the hybrid approach we propose provides satisfactory answers to the four issues raised in Section 1, and practical advantages, since the important body of work in control of PD is directly applicable to the specialization side of our framework.

8 Conclusions

We have proposed a novel scheme for a seamless integration of the techniques of abstract interpretation and partial deduction. Our scheme is parametric w.r.t. the abstract domain and the control issues which guide the partial deduction process. Existing proposals for the integration use abstract interpretation as a *means* for improving partial evaluation rather than as a *goal*, at the same level as producing a specialized program. This implies that, as a result, their objective is to yield a set of atoms which determines a partial evaluation rather than to compute a safe approximation of its success. Unlike them, a main objective of our work is to improve success information by analyzing the specialized code, rather than the original one. We achieve this objective by smoothly *interleaving* both techniques which, on one hand, improves success information—even for abstract domains which are not related directly to partial evaluation. On the other hand, with more accurate success information, we can improve further the quality of partial evaluation. The overall method thus yields not only a specialized program but also a safe approximation of its behaviour.

Acknowledgments

The authors would like to thank John Gallagher and Michael Leuschel for useful discussions on the integration of abstract interpretation and partial deduction.

References

1. M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10:91–124, 1991.
2. C. Consel and S.C. Koo. Parameterized partial deduction. *ACM Transactions on Programming Languages and Systems*, 15(3):463–493, July 1993.
3. P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of POPL'77*, pages 238–252, 1977.
4. P. Cousot and R. Cousot. Systematic Design of Program Transformation Frameworks by Abstract Interpretation. In *Proc. of POPL'02*, pages 178–190. ACM, 2002.
5. 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.
6. 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.
7. J.P. Gallagher. Static Analysis for Logic Program Specialization. In *Workshop on Static Analysis WSA'92*, pages 285–294, 1992.
8. J.P. Gallagher. Tutorial on specialisation of logic programs. In *Proc. of PEPM'93*, pages 88–98. ACM Press, 1993.
9. 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.
10. N. D. Jones. Combining Abstract Interpretation and Partial Evaluation. In *Static Analysis Symposium*, number 1140 in LNCS, pages 396–405. Springer-Verlag, 1997.

11. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, New York, 1993.
12. M. Leuschel. Program Specialisation and Abstract Interpretation Reconciled. In *Joint International Conference and Symposium on Logic Programming*, June 1998.
13. M. Leuschel. A framework for the integration of partial evaluation and abstract interpretation of logic programs. *ACM Transactions on Programming Languages and Systems*, 26(3):413 – 463, May 2004.
14. M. Leuschel and M. Bruynooghe. Logic program specialisation through partial deduction: Control issues. *Theory and Practice of Logic Programming*, 2(4 & 5):461–515, July & September 2002.
15. M. Leuschel and S. Gruner. Abstract conjunctive partial deduction using regular types and its application to model checking. In *Proc. of LOPSTR*, number 2372 in LNCS. Springer, 2001.
16. 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.
17. Michael Leuschel and De Schreye. Logic program specialisation: How to be more specific. In *Proc. of PLILP'96*, LNCS 1140, pages 137–151, 1996.
18. J.W. Lloyd. *Foundations of Logic Programming*. Springer, second, extended edition, 1987.
19. J.W. Lloyd and J.C. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11(3–4):217–242, 1991.
20. G. Puebla and M. Hermenegildo. Optimized Algorithms for the Incremental Analysis of Logic Programs. In *Proc. of SAS'96*, pages 270–284. Springer LNCS 1145, 1996.
21. 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.
22. G. Puebla and M. Hermenegildo. Abstract Multiple Specialization and its Application to Program Parallelization. *J. of Logic Programming.*, 41(2&3):279–316, November 1999.
23. G. Puebla and M. Hermenegildo. Abstract Specialization and its Applications. In *Proc. of PEPM'03*, pages 29–43. ACM Press, 2003. Invited talk.
24. G. Puebla, M. Hermenegildo, and J. Gallagher. An Integration of Partial Evaluation in a Generic Abstract Interpretation Framework. In *Proc. of PEPM'99*, number NS-99-1 in BRISC Series, pages 75–85. University of Aarhus, Denmark, 1999.

A Algorithm for Code Generation

Algorithm 3 presents the code generation phase. Since the specialized definitions generated already have different predicate names, the heads of the new clauses do not

Algorithm 3 Code Generation

function CODEGEN$(P, \mathcal{DT}, \mathcal{GT}, \mathcal{ST})$
  return $\{(H_k \leftarrow B'_k) \mid \exists (H_k \leftarrow B_k) \in spec\_defs(P, \mathcal{ST}) \wedge$
    $B'_k =$RENAME_BODY$(B_k, k, 1, \mathcal{DT}, \mathcal{GT}, \mathcal{ST})$
function RENAME_BODY$(B, k, i, \mathcal{DT}, \mathcal{GT}, \mathcal{ST})$
  if $B = (L, R)$ **then**
    $L' \leftarrow$ RENAME_ATOM$(L, k, i, \mathcal{DT}, \mathcal{GT}, \mathcal{ST})$
    $R' \leftarrow$ RENAME_BODY$(R, k, i+1, \mathcal{DT}, \mathcal{GT}, \mathcal{ST})$
    $B' \leftarrow (L', R')$
  else
    $B' \leftarrow$ RENAME_ATOM$(B, k, i, \mathcal{DT}, \mathcal{GT}, \mathcal{ST})$
  return B'
function RENAME_ATOM$(L, k, i, \mathcal{DT}, \mathcal{GT}, \mathcal{ST})$
  $L' : CP' \leftarrow SD((L : \_, \langle \_ : \_ \Rightarrow [\_ : \_], k, i \rangle), \mathcal{DT}, \mathcal{GT})$
  return Look_up$(\mathcal{ST}, L' : CP')$

need to be renamed. Function RENAME_BODY simply traverses the body of the clauses in the specialized definitions and replaces atoms for predicates in the original program with atoms for predicates in the specialized definitions. Deciding which predicate to use is done by function RENAME_ATOM. Note that since (optionally) constants are filtered out by function new_filter, this renaming can remove constants from the original program.