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 evaluation has received considerable attention and (partial) integrations have been proposed starting from both the partial evaluation and abstract interpretation perspectives. In this work we present what we argue is the first generic algorithm for efficient and precise integration of abstract interpretation and partial evaluation from an abstract interpretation perspective. Taking as starting point state-of-the-art algorithms for context-sensitive, polyvariant abstract interpretation and (abstract) partial evaluation of logic programs, 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. Also, our framework is parametric w.r.t. different control strategies and abstract domains. Different combinations of these parameters correspond to existing algorithms for program analysis and specialization. Our approach efficiently computes strictly more precise results than those achievable by each of the individual techniques. The algorithm is one of the key components of CiaoPP, the analysis and specialization system of the Ciao compiler.

1 Introduction and Motivation

The relationship between abstract interpretation [5] and partial evaluation [14] has received considerable attention. See, for instance, the relationship established in a general context in [4, 13, 6] and the work in the context of partial evaluation of logic programs (also known as *partial deduction* [21, 11]) of [8, 10, 18, 15, 24, 26, 9, 19, 25, 16]). In order to motivate our proposal, we use the running "challenge" example of Fig. 1. It is a simple Ciao [3] program which uses Peano's arithmetic.[4]

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

```
:- module(_,[main/2],[assertions]).
:- entry main(s(s(s(L))),R) : (ground(L),var(R)).
main$_1$(X,X2):-formula$_{1,1}$(X,X1), formula$_{1,2}$(X1,X2), ground$_{1,3}$(X2).
formula$_2$(X,W):-ground$_{2,1}$(X),var$_{2,2}$(W),two$_{2,3}$(T),minus$_{2,4}$(X,T,X2),twice$_{2,5}$(X2,W).
minus$_4$(X,0,X).
minus$_5$(s(X),s(Y),R) :- minus$_{5,1}$(X,Y,R).
minus$_6$(0,s(_Y),_R).
twice$_7$(X,_Y) :- var$_{7,1}$(X).
twice$_8$(X,Y) :- ground$_{8,1}$(X), tw$_{8,2}$(X,Y).
tw$_9$(0,0).
tw$_{10}$(s(X),s(s(NX))) :- tw$_{10,1}$(X,NX).
```

Fig. 1. Running Example

The `entry` declaration is used to inform that all calls to the exported predicate `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`. A call `formula(X,W)` performs mode tests `ground(X)` and `var(W)` on its input arguments and returns $W = (X - 2) \times 2$. Predicate `two/1` returns `s(s(0))`, i.e., the natural number 2. A call `minus(A,B,C)` returns $C = A - B$. 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. For initial queries satisfying the `entry` declaration, all calls to the tests ground$_{1,3}$(X), ground$_{2,1}$(X), and var$_{2,2}$(W) will definitely succeed. Thus, they can be replaced by *true*, even if we do not know the concrete values of variable `L` at compile time. Also, the calls to ground$_{8,1}$(X) will succeed, while the calls to var$_{7,1}$(X) will fail, and can thus be replaced by *fail*. These kinds of optimizations require abstract information from analysis (e.g., groundness and freeness).

The example illustrates four difficulties and challenges. First, the benefits of (1) *exploiting abstract information in order to abstractly execute certain atoms. Furthermore, this 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 ground$_{2,1}$(X) and var$_{2,2}$(W) will also succeed. The reason is that on success of minus$_{2,4}$(X,T,X2), `X2` cannot be guaranteed to be ground since minus$_6$/3 succeeds with a free variable in its third argument position. It can be observed, however, that for all calls to `minus/3` in executions described by the `entry` declaration the third clause for `minus/3` is useless. It will never contribute to a success of `minus/3` since this predicate is always called with a value greater than zero in its first 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). This unfolding allows concluding that in our particular context all calls to `minus/3` succeed with a ground third argument. This illustrates the importance of (2) *performing unfolding steps in order to prune away useless branches, and that this may result in improved success information.* By the time execution reaches `twice`$_{2,5}$`(X2,W)`, we hopefully know that `X2` is ground. In order to determine that upon success of `twice`$_{2,5}$`(X2,W)` (and thus on success of `formula`$_{1,1}$`(X,W)`) `W` is ground, we need to perform a fixpoint computation. Since, for example, the success substitution for `formula`$_{1,1}$`(X,X1)` is indeed the call substitution for `formula`$_{1,2}$`(X1,X2)`, the success of the second test `ground`$_{2,1}$`(X)` (i.e., the one reachable from `formula`$_{1,2}$`(X1,X2)`) cannot be established unless we propagate success substitutions. This illustrates the importance of (3) *propagating (abstract) success information, and performing fixpoint computations when needed, and that this simultaneously may result in an improved unfolding.* Finally, whenever we call `formula(X,W)`, the argument `W` is a variable, a property which cannot be captured if we restrict ourselves to downwards-closed domains (i.e., domains capturing properties such that once a property holds, it will keep on holding in every state accessible in forwards execution). This indicates (4) *the usefulness of having information on non* downwards-closed *properties.*

Example 1. `CiaoPP`, which implements our proposed abstract interpretation with specialized definitions, produces the following specialized code for the example of Fig. 1 (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).
```

Thus, our proposal can indeed eliminate all calls to mode tests `ground/1` and `var/1`, and fully unfold predicates `two/1` and `minus/3` so that they no longer appear in the residual code. In addition, the algorithm also produces an accurate analysis for such a program. In particular, the success information for `sp_main(X,X2)` guarantees that `X2` is 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. This is because, during the computation of the specialized definitions, the branches corresponding to values of `X` smaller than 3 are detected to be failing and the residual code is indeed equivalent to the one achieved with the more precise `entry` declaration. This illustrates how our proposal is useful for improving the results of the analysis even in cases where there are no initial constants in the query which can be propagated through the program.

The above results cannot be achieved unless all four points mentioned before are addressed by a program analysis/specialization system. For example, if we use traditional partial deduction (PD) with the corresponding *Generalize* and

Unfold rules followed by abstract interpretation and *abstract specialization* as described in [24, 25] we only obtain a comparable program after four iterations of the: "PD + abstract interpretation + abstract specialization" cycle. 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.

2 Preliminaries

This section introduces some preliminary concepts on abstract interpretation [5] and partial deduction [21]. We assume some basic knowledge on the terminology of logic programming (see for example [20] for details). Very briefly, 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.

2.1 The Notions of Unfolding and Resultant

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. If $\mathcal{R}(G) = A_R$ we say that A_R is the *selected* atom in G. The operational semantics of definite programs is based on derivations [20]. 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, the goal $\leftarrow \theta(A_1, \ldots, A_{R-1}, B_1, \ldots, B_m, A_{R+1}, \ldots, A_k)$ 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 non-deterministic 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 G_n is not empty and it is not possible to perform a derivation step from it. We will also allow *incomplete* derivations in which, though possible, no further resolution step is performed.

Given an atom A, an *unfolding rule* [21, 11] 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 set of resultants for the computed SLD tree is called a *partial deduction* (PD) for the initial goal.

2.2 Abstract Interpretation

Abstract interpretation [5] provides a general formal framework for computing safe approximations of program behaviour. Programs are interpreted using *abstract values* instead of *concrete values*. An abstract value is a finite representation of a, possibly infinite, set of concrete values in the concrete domain D. The set of all possible abstract values constitutes the *abstract domain*, denoted D_α, which is usually a complete lattice or cpo which is ascending chain finite. The subset relation $\subseteq$ induces a partial order on sets of concrete values. The $\subseteq$ relation induces the $\sqsubseteq$ relation on abstract values. 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 (e.g., program variable values) it represents, such that $\forall x \in 2^D : \ \gamma(\alpha(x)) \supseteq x$ and $\forall y \in D_\alpha : \ \alpha(\gamma(y)) = y$. Concrete values denote typically (but not exclusively) which data structures program variables are bound to in actual executions, i.e., the *substitutions*. Correspondingly, abstract values will be often referred to as *abstract substitutions*. The following operations on abstract substitutions are domain-dependent and will be used in our algorithms:

- $\mathsf{Arestrict}(\lambda, E)$ 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 of two substitutions;
- $\mathsf{Alub}(\lambda_1, \lambda_2)$ performs the abstract disjunction ($\sqcup$) of two substitutions.

An *abstract atom* of the form $A : CP$ is a concrete atom A which comes equipped with an *abstract substitution* CP which is defined over $vars(A)$ and provides additional information on the context in which the atom will be executed at run-time. We write $A : CP \sqsubseteq A' : CP'$ to denote that $\{\theta(A) | \theta \in \gamma(CP)\} \subseteq \{\theta'(A') | \theta' \in \gamma(CP')\}$. 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: $\mathsf{Atranslate}(A : CP, H \leftarrow B) = \mathsf{Arestrict}(\mathsf{Aunif}(A, H, \mathsf{Aextend}(CP, C)), C)$. As customary, the most general abstract substitution is represented as $\top$, and the least general (empty) abstract substitution as $\bot$.

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

3 Unfolding with Abstract Substitutions

We now present our notion of *abstract unfolding* —based on an extension of the SLD semantics which exploits abstract information— which is used later to generate specialized definitions. This will pave the way to overcoming difficulties (1) and (2) posed in Section 1.

3.1 SLD with Abstract Substitutions

Our extended semantics handles *abstract goals* of the form $G : CP$, i.e., a concrete goal G equipped with an *abstract substitution* CP. The first definition captures derivation steps.

Definition 1 (derivation step). *Let $G : CP$ be an abstract goal where $G = \leftarrow A_1, \ldots, A_R, \ldots, A_k$ and CP is an abstract substitution defined over $vars(G)$. 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 there exist $\theta = mgu(A_R, H)$ and $CP_u \neq \bot$, where:*

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

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 goal: $\mathtt{formula}(\mathtt{s}^4(\mathtt{X}), \mathtt{X2}) : \{\mathtt{X/G}, \mathtt{X2/V}\}$ which appears during the analysis of our running example (c.f. Fig. 2). We abbreviate as $\mathtt{s}^{\mathtt{n}}(\mathtt{X})$ the successive application of `n` symbols `s` to variable `X`. We have used sharing-freeness as abstract domain in the analysis though, for simplicity, we will represent the results using traditional "modes": the notation `X/G` (resp. `X/V`) indicates that variable `X` is ground (resp. free). After applying a derivation step using the only rule for `formula`, we derive:

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

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

The second extension we present makes use of the availability of abstract substitutions to perform *abstract executability* [24] 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` the abstract execution table contains the information $\texttt{ground(X)} : \{\texttt{X/G}\} \rightsquigarrow \texttt{true}$. For `var`, it contains $\texttt{var(X)} : \{\texttt{X/V}\} \rightsquigarrow \texttt{true}$.[5]

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*

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

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

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

since both calls succeed by using the abstract executability table described above.

3.2 Abstract Unfolding

In our framework, resultants for abstract atoms will be obtained using abstract unfolding in a similar way as it is done in the concrete setting using unfolding (see Sect. 2.1).

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 definitions 1 and 2 to $A : CP$.*

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 [17] for details).

Example 4. Consider an unfolding rule *AUnfold* based on homeomorphic embedding [17] 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:

[5] In `CiaoPP` *assertions* express such information in a domain-independent manner.

```
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 as they appear in rules 5 and 6 of Ex. 1.

It is important to note that SLD resolution with abstract substitutions is not restricted to the left-to-right computation rule. For the case of derivation steps (Def. 1), it is well-known that non-leftmost 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 [1] and its references. For the case of abstract execution (Def. 2), the execution of non-leftmost atoms can be incorrect if the abstract domain used captures properties which are not downwards closed. A simple solution in this case is to allow only leftmost abstract execution steps for non-downwards closed domains.

4 Specialized Definitions

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 [21]. In order to ensure termination of this global process, the so-called *global* control defines a *Generalize* operator (see, e.g., [17]) 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 deduction is produced. However, the residual program is not generated until such iterative process terminates.

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 be able to generate residual code for each 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. As a consequence, the accuracy of the analyzer may be increased and difficulty (2) described in Sect. 1 overcome.

4.1 Abstract Partial Deduction

Algorithm 1 presents an APD algorithm. The main difference with standard algorithms for APD is that the resultants computed by *AUnfold* (L23) are added to the program during execution of the algorithm (L27) 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 (L26) prior to adding them to the program (L27). 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

Algorithm 1 Abstract Partial Deduction with Specialized Definitions

```
1: procedure PARTIAL_EVALUATION_WITH_SPEC_DEFS($P, \{A_1 : CP_1, \ldots, A_n : CP_n\}$)
2:     Create_Table($\mathcal{GT}$); Create_Table($\mathcal{ST}$)
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($\mathcal{GT}, A : CP$) then
7:         $(A_1, A'_1) \leftarrow$ SPECIALIZED_DEFINITION($P, A : CP$)
8:         $A_1 : CP_1 \leftarrow$ Look_up($\mathcal{GT}, A : CP$)
9:         for all renamed apart clause $C_k = H_k \leftarrow B_k \in P$ s.t. $H_k$ unifies with $A'_1$
           do
10:            $CP_k \leftarrow$ 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:         $CP_L \leftarrow$ Arestrict($CP, L$)
15:         PROCESS_CALL_PATTERN($L : CP_L$)
16:         PROCESS_CLAUSE($CP,\ R$)
17: function SPECIALIZED_DEFINITION($P, A : CP$)
18:     $A' : CP' \leftarrow AGeneralize(\mathcal{ST}, A : CP)$
19:     Insert($\mathcal{GT}, A : CP, A' : CP'$)
20:     if IsIn($\mathcal{ST}, A' : CP'$) then
21:         $A'' \leftarrow$Look_up($\mathcal{ST}, A' : CP'$)
22:     else
23:         $Def \leftarrow AUnfold(P, A' : CP')$
24:         $A'' \leftarrow$ new_filter($A'$)
25:         Insert($\mathcal{ST}, A' : CP', A''$)
26:         $Def' \leftarrow \{(H' \leftarrow B) \mid (H \leftarrow B) \in Def \wedge H' = \text{ren}(H, \{A'/A''\})\}$
27:         $P \leftarrow P \bigcup Def'$
28:     return $(A', A'')$
```

AGeneralize function and contains entries $A : CP \rightsquigarrow A' : CP'$ where $A' : CP'$ is a generalization of $A : CP$, in the sense that $A = A'\theta$ and $(A : CP) \sqsubseteq (A' : CP')$.

Let us briefly discuss some *AGeneralize* functions which can be used within our algorithms when using it as a specializer. 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. Then we define $AGeneralize_\alpha(\mathcal{ST}, A : CP) = (A', CP')$ where $A' = Generalize(\mathcal{ST}, A)$ and $CP' = \text{Atranslate}(A : CP, A' \leftarrow true)$. Function $AGeneralize_\alpha$ 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 $AGeneralize_\gamma$ 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 but different abstract substitution.

Procedure PARTIAL_EVALUATION_WITH_SPEC_DEFS (L1-4) initiates the computation. It first initializes the tables and then calls PROCESS_CALL_PATTERN for each abstract atom $A_j : CP_j$ 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 specialized definition for it (L7) and then process all clauses in its specialized definition by means of calls to PROCESS_CLAUSE (L9-11). For simplicity of the presentation, we assume that clause bodies returned by SPECIALIZED_DEFINITION are represented as lists rather than conjunctions. 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. In contrast, the order in which pending call patterns (atoms) are handled is usually not fixed in APD algorithms. They are often all put together in a set. The purpose of the two procedures PROCESS_CLAUSE and PROCESS_CALL_PATTERN is to traverse the clauses in the left-to-right order and add the corresponding call patterns. In principle, this does not have additional advantages w.r.t. existing APD algorithms because success propagation has not been integrated yet. However, the reason for our presentation is to be as close as possible to our analysis algorithm with success propagation, which enforces a depth-first, left-to-right traversal of program clauses. Correctness of Algorithm 1 can be established using the framework for APD in [16].

4.2 Integration with an Abstract Interpreter

For the integration we propose, the most relevant part of the algorithm comprises L17-28, as it is the code fragment which is *directly* executed from our abstract interpreter. The remaining procedures (L1-L16) will be overridden by more accurate ones later on. The procedure of interest is SPECIALIZED_DEFINITION. It performs (L18) 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 (L19) in the generalization table $\mathcal{GT}$. It is required that $(A : CP) \sqsubseteq (A' : CP')$. If $A' : CP'$ has been previously treated (L20), then its specialized definition A'' is looked up in $\mathcal{ST}$ (L21) and returned. Otherwise, a specialized definition *Def* is computed by using the *AUnfold* operator (L23).

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 (L24). Then, function ren is applied to rename the clause heads using atom A' (L26). 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 $\texttt{main}(\texttt{s}^3(\texttt{X}), \texttt{X2}) : \{\texttt{X/G}, \texttt{X2/V}\}$, then $\texttt{tw}(\texttt{B}, \texttt{C}) : \{\texttt{B/G}, \texttt{C/V}\}$ and finally $\texttt{formula}(\texttt{C}, \texttt{A}) : \{\texttt{C/G}, \texttt{A/V}\}$. 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

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

where only the head is renamed. The renaming of the body literals is done in a later code-generation phase.

It is important to note that Algorithm 1 does not perform success propagation yet (difficulty 3). In L16, it becomes apparent that all atom(s) in R will be analyzed with the same call pattern CP as L, which is to their left in the clause. This may clearly lead to substantial precision loss. In the above example, Alg. 1 is not able to obtain the three abstract atoms above due to the absence of success propagation. For instance, the abstract pattern $\texttt{formula}(\texttt{C}, \texttt{A}) : \{\texttt{C/G}, \texttt{A/V}\}$ which is necessary in order 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. In addition, this lack of success propagation makes it difficult or even impossible to work with non downwards closed domains (difficulty 4), since CP may contain information which holds before execution of the leftmost atom L but which can no longer hold after that. In fact, in our example CP contains the info `C/V`, which becomes false after execution of $\texttt{tw}(\texttt{B}, \texttt{C})$, 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.

5 Abstract Interpretation with Specialized Definitions

The main idea in *abstract interpretation with specialized definitions* is that a generic abstract interpreter is equipped with a generator of specialized definitions. Such generator provides, upon request, the specialized definitions to be analyzed by the interpreter. Certain data structures, which take the form of tables in the algorithms (i.e., the specialization, generalization, answer and dependency arc tables) will be used to communicate between the two processes and achieve a smooth interleaving. The input to the whole process is a program together with a set of calling patterns for it. The output is a specialized program together with the analysis results inferred for it. The scheme can be parameterized with different (abstract) unfolding rules, generalization operators, abstract domains and widenings. The different instances give rise to interesting analysis and specialization methods, some of which are well known and others are novel (see Section 7).

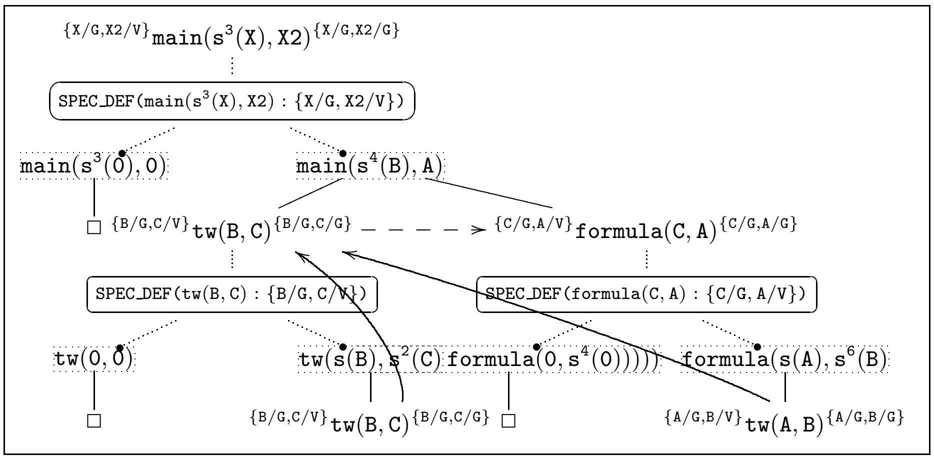
{X/G,X2/V}main(s^3(X),X2){X/G,X2/G}
SPEC_DEF(main(s^3(X),X2) : {X/G,X2/V})
main(s^3(0),0)
main(s^4(B),A)
{B/G,C/V}tw(B,C){B/G,C/G}
{C/G,A/V}formula(C,A){C/G,A/G}
SPEC_DEF(tw(B,C) : {B/G,C/V})
SPEC_DEF(formula(C,A) : {C/G,A/V})
tw(0,0)
tw(s(B),s^2(C))
formula(0,s^4(0))))
formula(s(A),s^6(B))
{B/G,C/V}tw(B,C){B/G,C/G}
{A/G,B/V}tw(A,B){A/G,B/G}

Fig. 2. Analysis Graph computed by ABS_INT_WITH_SPEC_DEF

Algorithm 2 presents our final algorithm for abstract interpretation with specialized definitions. This algorithm extends both the APD Algorithm 1 and the abstract interpretation algorithms in [23, 12]. The main improvement w.r.t. Algorithm 1 is the addition of success propagation, which requires computing a global fixpoint. It is an important objective for us to be able to compute an accurate fixpoint in an efficient way. The main improvements w.r.t the algorithms in [23, 12] are the following. (1) It interleaves program analysis and specialization in a way that is efficient, accurate, and practical. (2) Algorithm 2 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. (3) It incorporates a hardwired efficient graph traversal strategy which eliminates the need for maintaining priority queues explicitly [12]. (4) The algorithm includes a widening operation for calls, *Widen_Call*, which limits the amount of multivariance in order to keep the number of call patterns analyzed finite. This is required in order to be able to use abstract domains with an infinite number of elements, such as regular types. (5) 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.

5.1 The Program Analysis Graph: Answer and Dependency Tables

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 the `CiaoPP` analyzer [23, 12]. 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". An OR-node of the form ${}^{CP}A^{AP}$ is interpreted as the answer (success) pattern for the abstract atom $A : CP$ is AP. The OR-node ${}^{\{\mathtt{X/G,X2/V}\}}\mathtt{main}(\mathtt{s}^3(\mathtt{X}),\mathtt{X2})^{\{\mathtt{X/G,X2/G}\}}$ in the example 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 its body. If a child OR-node is already in the tree, it is not expanded any further and the currently available answer is used. 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 heads of the clauses in the specialized definition are linked to the box with a dotted arc. 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 [23, 12, 2].

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 $A : CP \leadsto AP$ which are interpreted as the answer (success) pattern for $A : CP$ is AP. For instance, there exists an entry of the form $\mathtt{main}(\mathtt{s}^3(\mathtt{X}),\mathtt{X2}) : \{\mathtt{X/G},\mathtt{X2/V}\} \leadsto \{\mathtt{X/G},\mathtt{X2/G}\}$ associated to the OR-node 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, i.e., for each OR-node $A_T : CP_T$ we keep track of the set of OR-nodes which depend on it. I. e., 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 $[H_k : CP_1]$ encodes the fact that the head of this clause is H_k and the substitution (in terms of all variables of clause k) just before the call to $B : CP_2$ is CP_1. Such information avoids having to reprocess atoms in the clause k to the left of position i.

Example 6. For instance, the dependency set for $\mathtt{formula}(\mathtt{C},\mathtt{A}) : \{\mathtt{A/V},\mathtt{C/G}\}$ is $\{\langle \mathtt{main}(\mathtt{s}^3(\mathtt{X}),\mathtt{X2}) : \{\mathtt{X/G},\mathtt{X2/V}\} \Rightarrow [\ \mathtt{main}(\mathtt{s}^4(\mathtt{B}),\mathtt{A}) : \{\mathtt{B/G},\mathtt{A/V},\mathtt{C/G}\}\ ]\ 2, 2\rangle\}$ It indicates that the OR-node $\mathtt{formula}(\mathtt{C},\mathtt{A}) : \{\mathtt{A/V},\mathtt{C/G}\}$ is only used in the OR-node $\mathtt{main}(\mathtt{s}^3(\mathtt{X}),\mathtt{X2}) : \{\mathtt{X/G},\mathtt{X2/V}\}$ via literal 2,2 (see Example 1). Thus, if the

Algorithm 2 Abstract Interpretation with Specialized Definitions

```
1: procedure ABS_INT_WITH_SPEC_DEFS($P, \{A_1 : CP_1, \ldots, A_n : CP_n\}$)
2:     Create_Table($\mathcal{AT}$); Create_Table($\mathcal{DT}$); Create_Table($\mathcal{GT}$); Create_Table($\mathcal{ST}$);
3:     for $j = 1..n$ do
4:         PROCESS_CALL_PATTERN($A_j : CP_j, \langle A_j : CP_j \Rightarrow [A_j : CP_j], j, entry\rangle$)
5: function PROCESS_CALL_PATTERN($A : CP, Parent$)
6:     $CP_1 \leftarrow Widen\_Call(\mathcal{AT}, A : CP)$
7:     if not IsIn($\mathcal{AT}, A : CP_1$) then
8:         Insert($\mathcal{AT}, A : CP_1, \perp$); Insert($\mathcal{DT}, A : CP_1, \emptyset$)
9:         $(A', A'_1) \leftarrow$ SPECIALIZED_DEFINITION($P, A : CP_1$)
10:        $A'' \leftarrow$ ren($A, \{A'/A'_1\}$)
11:        for all renamed clause $C_k = H_k \leftarrow B_k \in P$ s.t. $H_k$ unifies with $A''$ do
12:            $CP_k \leftarrow$ Atranslate($A'' : CP_1, C_k$)
13:            PROCESS_CLAUSE($A : CP_1 \Rightarrow [H_k : CP_k]\ B_k, k, 1$)
14:    $Deps \leftarrow$ Look_up($\mathcal{DT}, A : CP_1$) $\bigcup\{Parent\}$; Insert($\mathcal{DT}, A : CP_1, Deps$)
15:    return Look_up($\mathcal{AT}, A : CP_1$)
16: procedure PROCESS_CLAUSE($H : CP \Rightarrow [H_k : CP_1]\ B, k, i$)
17:    if $CP_1 \neq \perp$ then
18:        if $B = [L|R]$ then
19:            $CP_2 \leftarrow$ Arestrict($CP_1, L$)
20:            $AP_0 \leftarrow$ PROCESS_CALL_PATTERN($L : CP_2, \langle H : CP \Rightarrow [H_k : CP_1], k, i\rangle$)
21:            $CP_3 \leftarrow$ Aconj($CP_1$, Aextend($AP_0, CP_1$))
22:            PROCESS_CLAUSE($H : CP \Rightarrow [H_k : CP_3]R, k, i + 1$)
23:        else
24:            $AP_1 \leftarrow$ Atranslate($H_k : CP_3, H \leftarrow true$); $AP_2 \leftarrow$ Look_up($\mathcal{AT}, H : CP$)
25:            $AP_3 \leftarrow$ Alub($AP_1, AP_2$)
26:            if $AP_2 \neq AP_3$ then
27:                Insert($\mathcal{AT}, H : CP, AP_3$)
28:                $Deps \leftarrow$ Look_up($\mathcal{DT}, H : CP$); PROCESS_UPDATE($Deps$)
29: procedure PROCESS_UPDATE($Updates$)
30:    if $Updates = \{A_1, \ldots, A_n\}$ with $n \geq 0$ then
31:        $A_1 = \langle H : CP \Rightarrow [H_k : CP_1], k, i\rangle$
32:        if $i \neq entry$ then
33:            $B \leftarrow$ get_body($P, k, i$)
34:            REMOVE_PREVIOUS_DEPS($H : CP \Rightarrow [H_k : CP_1]\ B, k, i$)
35:            PROCESS_CLAUSE($H : CP \Rightarrow [H_k : CP_1]\ B, k, i$)
36:            PROCESS_UPDATE($Updates - \{A_1\}$)
```

answer pattern for `formula(C,A)` : {`A/V`, `C/G`} is ever updated, then we must reprocess the OR-node `main(s`^{3}`(X),X2)` : {`X/G`, `X2/V`} from position 2,2.

5.2 The Algorithm

Algorithm 2 presents our proposed algorithm. 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. PROCESS_CALL_PATTERN applies, first of all (L6), 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 for each set of variables. 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 (L8) $\bot$ as initial answer in $\mathcal{AT}$ for $A : CP$ and sets to empty the set of OR-nodes in the graph which depend on $A : CP_1$. It then computes (L9) 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. Then (L11-13) calls to PROCESS_CLAUSE are launched for the clauses in the specialized definition w.r.t. which $A : CP_1$ is to be analyzed. Then, the *Parent* OR-node is added (L14) 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 (L21) conjoined (Aconj), after being extended ($\mathsf{Aextend}$) 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 (L22). 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 (L24) adapted ($\mathsf{Atranslate}$) to the initial call pattern $H : CP$ which started PROCESS_CLAUSE, obtaining AP_1. This value is (L25) disjoined (Alub) with the current answer, AP_2, for $H : CP$ as given by $\mathsf{Look\_up}$. If the answer changes, then its dependencies, which are readily available in $\mathcal{DT}$, need to be recomputed (L28) 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_PREVIOUS_DEPS eliminates (L34) entries in $\mathcal{DT}$ for the clause postfix which is about to be recomputed. We do not present its definition here due to lack of space. Note that the new calls (L35) 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 L4). These do not need to be reprocessed (L32) but are useful for determining the specialized version to use for the initial queries after code generation.

The next theorem presents the correctness of the results of Algorithm 2 in terms of analysis. We use $\theta|_{\{X_1,\ldots,X_n\}}$ to denote the projection of substitution θ onto the set of variables $\{X_1, \ldots, X_n\}$. We denote by $\mathit{success}(A : CP, P)$ the set of computed answers for initial queries described by the abstract atom $A : CP$ in a program P.

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

Intuitively, correctness holds since Algorithm 2 computes an abstract and–or graph and, thus, we inherit a generic correctness result for success substitutions of [12]. 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.

5.3 The Framework as a Specializer

If we compose a terminating analysis strategy (abstract domain plus widening operator) with a terminating PD strategy (local control plus global control), then Algorithm 2 also terminates for such strategies. The set of specialized definitions computed during the execution of the algorithm is a specialization of the program w.r.t. the initial entries.

Theorem 2 (correctness of specialization). *Consider the Algorithm 2 parameterized with terminating operators AUnfold, Widen_Call and AGeneralize. Then, for any program P and set of abstract atoms S,* ABS_INT_WITH_SPEC-_DEFS(P, S) *terminates and the set of renamed specialized definitions is a correct specialization of P w.r.t. S.*

Intuitively, if we have a terminating *AUnfold* rule and the abstract domain is ascending chain finite, non-termination can only occur if the set of call patterns handled by the algorithm is infinite. Since the *Widen_Call* function guarantees that a given concrete atom A can only be analyzed w.r.t. a finite number of abstract substitutions CP, non-termination can only occur if the set of atoms has an infinite number of elements with different concrete parts. If the *AGeneralize* function guarantees that an infinite number of different concrete atoms cannot occur, then termination is guaranteed.

6 Experiments

In this section we show some experimental results aimed at studying two crucial points for the practicality of our proposal: the cost associated to computing specialized definitions and the optimization obtained by the process. We have implemented the abstract interpreter with specialized definitions as an extension of the generic abstract interpretation system of `CiaoPP`. The whole system is implemented in Ciao 1.13#5666 [3]. Execution times are given in milliseconds and measure *runtime*. They are computed as the arithmetic mean of five runs. All of our experiments have been performed on a Pentium M at 1.86GHz and 1GB RAM running Ubuntu Breezy Linux. The Linux kernel used is 2.6.12.

A relatively wide range of programs has been used as benchmarks. The program `running_ex` is that in Fig. 1. The rest are the same programs used in [12] as benchmarks for static analysis.[6] Thus, they do not necessarily contain static data which can be exploited by partial evaluation. Interestingly, some (first group of

[6] More details on such benchmarks can be found in [12].

		Traditional			SD_γ		$SD_{\alpha}-$		SD_α		Exec T
Bench	Abs	Ana	PD	Ana PD	SD_γ	SU	$SD_{\alpha}-$	SU	SD_α	SU	SU
running_ex	shfr	5	11	5	13	1.20	14	1.14	14	1.10	1.33
grammar	shfr	24	4	21	24	1.03	27	0.92	34	0.72	1.59
query	shfr	358	160	15	173	1.01	187	0.93	453	0.38	2.69
zebra	shfr	261	1523	1	1522	1.00	1604	0.95	6476	0.24	1148.08
aiakl	shfr	13	25	25	44	1.15	53	0.95	50	1.01	1.00
ann	shfr	432	159	452	558	1.10	625	0.98	604	1.01	1.00
boyer	shfr	154	90	161	232	1.08	271	0.93	241	1.04	1.00
progeom	shfr	9	26	14	37	1.10	39	1.03	41	0.98	0.99
warplan	shfr	318	63	311	410	0.91	607	0.62	553	0.68	1.01
witt	shfr	103	183	118	255	1.18	288	1.04	276	1.09	1.00
browse	eterms	33	18	36	50	1.07	71	0.75	65	0.83	1.00
deriv	eterms	149	5	151	151	1.03	160	0.97	161	0.97	1.00
fib	eterms	13	2	13	15	1.03	17	0.89	17	0.87	1.00
hanoiapp	eterms	61	5	65	73	0.96	101	0.70	97	0.73	1.00
mmatrix	eterms	68	4	69	71	1.04	74	0.99	72	1.03	1.00
occur	eterms	24	7	24	30	1.02	49	0.62	44	0.69	1.00
serialize	eterms	68	13	73	85	1.03	108	0.81	97	0.89	1.03
tak	eterms	5	3	5	7	1.21	9	0.95	9	0.95	1.00
Overall						1.03		0.90		0.41	

Table 1. Some implementations of AI with Specialized Definitions. Cost and efficiency

rows in Table 1) contain static data, while others (second and third groups of rows in Table 1) contain little or no static data. In `zebra` all the data is static and it can be potentially fully evaluated at compile-time.

As the analyzers within `CiaoPP` it derives from, our abstract interpreter with specialized definitions is parametric w.r.t. the abstract domain. In these experiments we have used mostly the *sharing+freeness* domain [22] (for the first and second group of rows in Table 1). We have selected this domain because it is on one hand well known and on the other orthogonal w.r.t. partial evaluation, in the sense that it does not contain any concrete information (as, for example, a depth-k or types domain would). We have also conducted experiments with the *eterms* domain [27] which infers regular types (third group of rows in the table).

For each benchmark, the columns under `Traditional` present the analysis (`Ana`) and partial deduction (`PD`) times using the standard algorithms. Column `Ana PD` provides the time taken by analysis of the specialized program (rather than the original one). Each of the following six columns presents the time taken by the abstract interpreter with specialized definitions, as well as the ratio (speedup/slowdown, `SU`) of this time w.r.t. `PD` + `Ana PD`. Columns marked $\texttt{SD}_\alpha$ are for the case where $AGeneralize_\alpha$ (Section 5) is used, whereas $\texttt{SD}_\gamma$ columns use $AGeneralize_\gamma$, with $\texttt{SD}_\alpha-$ representing the case where we only check for useless clauses once a derivation is fully computed, rather than at each derivation step. Finally, the last column represents the *speedup* in the execution time of the program after applying $\texttt{SD}_\alpha-$.

The last row summarizes the analysis times for the different benchmarks using a weighted mean, which places more importance on those benchmarks with relatively larger analysis times. We believe that this weighted mean is more informative than the arithmetic mean, as, for example, doubling the speed in which a large and complex program is analyzed (checked) is more relevant than achieving this for small, simple programs.

Overall, we first observe that the time taken by the abstract interpreter with specialized definitions compares well with that taken by a traditional PD phase followed by a traditional analysis phase (`Ana PD`). In the case of $\texttt{SD}_\gamma$ there is actually some speedup (1.03), presumably because fewer traversals of the program are required, whereas in the case of $\texttt{SD}_\alpha$ we observe a reasonable slowdown (0.41), with $\texttt{SD}_\alpha-$ representing an interesting tradeoff (0.90). The execution times of the resulting programs show significant speedups for the first group (in which concrete information is available for specialization) and (as expected) only very minor variations for the other programs. This shows that our system performs well as a specializer. At the same time, the analysis information obtained (which is of course one of the fundamental objectives of the process) is always at least as accurate as that obtained when performing analysis after a standalone specialization pass (`Ana PD`), and is more accurate for the programs in the first group, which shows that it also performs well as an analyzer.

7 Discussion and Related Work

The versatility of our approach can be seen by recasting well-known specialization and analysis frameworks as instances where the parameters unfolding rule, widen call rule, abstraction operator, and analysis domain, take different values.

From an analysis point of view, our algorithm can behave as the *polyvariant abstract interpretation* algorithm described in [12, 23] by defining an *AGeneralize* operator which returns 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). Also, the specialization power of the *multivariant abstract specialization* framework described in [25, 24] 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. However abstract executability is performed now online, during analysis, instead of offline.

From a partial evaluation perspective, our method can be used to perform *classical partial deduction* in the style of [21, 11] 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 [16] 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 at the global control level, the process of generating specialized definitions (as discussed in Section 3) can perform *non-leftmost* unfolding steps at the local control level and achieve the same optimizations as in PD. Several approaches for *abstract partial deduction* have been proposed which extend PD with SLDNF-trees by using abstract substitutions [15, 9, 19, 16]. In essence, such approaches are very similar to APD 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 within 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 [9, 19] or do not provide the exact description of operations on the abstract domain which are needed by the framework, other than general correctness criteria [15, 16]. However, the latter allow Conjunctive PD [7], which is not available in our framework yet. It remains as future work to investigate the extension of our framework in order to analyze conjunctions of atoms and in order to achieve optimizations like tupling and deforestation.

Finally, [26] was a very preliminary (and only informally published) step towards our current framework which identified the need for including unfolding in abstract interpretation frameworks in order to increase their power. Then, four different alternatives for doing so (Section 5.3) were discussed. The framework we propose in this work does not correspond to any of those alternatives and is in fact more powerful than any of them.

Acknowledgments

The authors would like to thank John Gallagher and Michael Leuschel for useful discussions. This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* project, by the MEC project TIN-2005-09207 *MERIT*, and the CAM project S-0505/TIC/0407 *PROMESAS*. Manuel Hermenegildo is also supported by the Prince of Asturias Chair in Information Science and Technology at UNM.

References

1. E. Albert, G. Puebla, and J. Gallagher. Non-Leftmost Unfolding in Partial Evaluation of Logic Programs with Impure Predicates. In *Proc. of LOPSTR'05*. Springer LNCS 3901, April 2006.
2. M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10:91–124, 1991.
3. F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla. The Ciao Prolog System. Reference Manual (v1.8). The Ciao System Documentation Series–TR CLIP4/2002.1, School of Computer Science, Technical University of Madrid (UPM), May 2002. System and on-line version of the manual available at `http://clip.dia.fi.upm.es/Software/Ciao/`.
4. C. Consel and S.C. Koo. Parameterized partial deduction. *ACM Transactions on Programming Languages and Systems*, 15(3):463–493, July 1993.
5. 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.
6. P. Cousot and R. Cousot. Systematic Design of Program Transformation Frameworks by Abstract Interpretation. In *POPL'02*, pages 178–190. ACM, 2002.

7. D. De Schreye, R. Glück, J. Jørgensen, M. Leuschel, B. Martens, and M.H. Sørensen. Conjunctive Partial Deduction: Foundations, Control, Algorihtms, and Experiments. *Journal of Logic Programming*, 41(2&3):231–277, 1999.
8. J. Gallagher, M. Codish, and E. Shapiro. Specialisation of Prolog and FCP Programs Using Abstract Interpretation. *NGC*, 6(2–3):159–186, 1988.
9. J. P. Gallagher and J. C. Peralta. Regular tree languages as an abstract domain in program specialisation. *HOSC*, 14(2,3):143–172, 2001.
10. J.P. Gallagher. Static Analysis for Logic Program Specialization. In *Workshop on Static Analysis WSA'92*, pages 285–294, 1992.
11. J.P. Gallagher. Tutorial on specialisation of logic programs. In *Proc. of PEPM'93*, pages 88–98. ACM Press, 1993.
12. M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental Analysis of Constraint Logic Programs. *ACM TOPLAS*, 22(2):187–223, March 2000.
13. N. D. Jones. Combining Abstract Interpretation and Partial Evaluation. In *Static Analysis Symposium*, number 1140 in LNCS, pages 396–405. Springer-Verlag, 1997.
14. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, New York, 1993.
15. M. Leuschel. Program Specialisation and Abstract Interpretation Reconciled. In *Joint International Conference and Symposium on Logic Programming*, June 1998.
16. M. Leuschel. A framework for the integration of partial evaluation and abstract interpretation of logic programs. *ACM TOPLAS*, 26(3):413 – 463, May 2004.
17. 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.
18. M. Leuschel and D. De Schreye. Logic program specialisation: How to be more specific. In *Proc. of PLILP'96*, LNCS 1140, pages 137–151, 1996.
19. 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.
20. J.W. Lloyd. *Foundations of Logic Programming*. Springer, second, extended edition, 1987.
21. J.W. Lloyd and J.C. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11(3–4):217–242, 1991.
22. K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *1991 International Conference on Logic Programming*, pages 49–63. MIT Press, June 1991.
23. G. Puebla and M. Hermenegildo. Optimized Algorithms for the Incremental Analysis of Logic Programs. In *SAS'96*, pages 270–284. Springer LNCS 1145, 1996.
24. G. Puebla and M. Hermenegildo. Abstract Multiple Specialization and its Application to Program Parallelization. *JLP*, 41(2&3):279–316, November 1999.
25. G. Puebla and M. Hermenegildo. Abstract Specialization and its Applications. In *Proc. of PEPM'03*, pages 29–43. ACM Press, 2003. Invited talk.
26. G. Puebla, M. Hermenegildo, and J. Gallagher. An Integration of Partial Evaluation in a Generic Abstract Interpretation Framework. In *PEPM'99*, number NS-99-1 in BRISC Series, pages 75–85. Univ. of Aarhus, Denmark, 1999.
27. C. Vaucheret and F. Bueno. More precise yet efficient type inference for logic programs. In *Proc. of SAS'02*, pages 102–116. Springer LNCS 2477, 2002.