DIFFERENTIAL METHODS IN LOGIC PROGRAM ANALYSIS

MARIA GARCIA DE LA BANDA, KIM MARRIOTT, PETER STUCKEY AND HARALD SØNDERGAARD

▷ Program analysis based on abstract interpretation has proven very useful in compilation of constraint and logic programming languages. Unfortunately, the traditional goal-dependent framework is inherently imprecise. This is because they handle call and return in such a way that dataflow information may be re-asserted unnecessarily, leading to a loss of precision for many description domains. For a few specific domains, the literature contains proposals to overcome the problem, and some implementations use various unpublished tricks that sometimes avoid the precision loss.

The purpose of this paper is to map the landscape of goal-dependent, goal-independent, and combined approaches to generic analysis of logic programs. This includes formalising existing methods and tricks in a way that is independent of specific description domains. Moreover, we suggest new methods for overcoming the loss of precision—altogether eight different semantics are considered and compared. We provide theoretical results determining the relative accuracy of the approaches. These show that two of our new semantics are uniformally more accurate than existing approaches. Experiments that we have performed (for two description domains) with implementations of the eight different approaches enable a discussion of their relative runtime performances. We discuss the expected effect on other domains as well and conclude that our new methods can be trusted to yield significantly more accurate analysis for a small extra implementation effort, without compromising the efficiency of analysis. ◁

Address correspondence to Kim Marriott, Dept. of Computer Science, Monash University, Clayton Vic. 3168, Australia.

This work has been supported by a grant from the Australian Research Council.

THE JOURNAL OF LOGIC PROGRAMMING

655 Avenue of the Americas, New York, NY 10010 0743-1066/94/$7.00

1. INTRODUCTION

Abstract interpretation based program analysis has proven very useful in compilation of logic programming languages. Several compilers have demonstrated that global program analysis can give large performance improvement for the sequential compilation of Prolog (see for example [21, 22]) and significantly improve automatic parallelisation (see for example [2]). Recently, Kelly *et al.* [14] have shown that global program analysis is also important for constraint logic programming (CLP) languages. Initial theoretical research in abstract interpretation of logic programs has resulted in several generic "frameworks" for logic programming. These formal frameworks are very similar, and can be instantiated to a particular analysis by giving several parametric functions over the description domain. We refer to these frameworks jointly as the goal-dependent traditional approach.

The traditional goal-dependent frameworks [1, 13, 16] share the same "abstract execution" mechanism. The output of the analysis is a program annotated with the information obtained at each program point. Processing mimics normal execution of programs to a certain extent. The basic idea is to process a call to a predicate as follows. First, the current constraint description is restricted to the variables in the calling atom and the call is evaluated with this restricted description as input description. The answer to the call is obtained by processing the literals in the atom's definition with the input description. However, on return from the call, the answer to the call is combined with the original unrestricted description so as to give information about all of the variables. This combination is necessary because of the restriction that was performed at the call. We shall see an example shortly.

In its most straightforward definition, the combination of descriptions approximates conjunction of arbitrary constraints. However, many analysers implement a specialised combination which is more accurate and/or efficient, but more complex, than arbitrary conjunction. This specialised combination is based on the observation that the answers already contain all the information about the variables in the analysed atom, and the role of the combination is just to *extend* this information to the rest of variables in the call.

From an efficiency point of view, the above traditional frameworks suffer the drawback that the answers to an atom can be computed several times, each corresponding to a different call pattern. This re-computation can increase the accuracy but also reduce the efficiency of the analysis. For this reason, Jacobs and Langen [11] proposed a goal-independent method which avoids the problem. Their method (referred to as condensing) is based on the observation that given an answer D to an atom A with current constraint $true$, an answer to A with current constraint D' can be computed simply by conjoining D and D'. The advantage is that we only need to compute the answers to each atom in the program for one calling pattern, namely the most general one. The answers to other calling patterns can be obtained from this by conjunction. The disadvantage is that we now need two phases in the analysis: a first phase to compute the most general answers to each atom, and a second, goal-dependent, phase which uses this information to annotate the program for a given goal.

It has long been part of the program analysis folklore that goal-independent analyses are more efficient but less accurate than goal-dependent analyses. So when Jacobs and Langen [11] proposed their method, they focussed on defining the characteristics that the abstract domain operations should satisfy in order not to lose accuracy. Some empirical evaluations have been conducted to test the relative efficiency and accuracy of different domains, see for example Codish *et al.* [3].

We show, however, that in general the two approaches are incomparable with respect to accuracy, and in a certain sense, the traditional goal-dependent framework is inherently less precise than the two-phase goal-independent based approach. This is because in the goal-dependent framework, in effect, the original information about the call variables has been added twice—first in the call to the atom, and then again when the answer is combined with the original call. For many description domains (including the concrete domain) this leads to a loss of information because the domains are not "idempotent": adding the same information twice leads to a different and less precise result. To appreciate the problem, consider the well-known 'append' program:

$$app(X, Y, Z) \leftarrow X = nil, Y = Z$$
$$app(X, Y, Z) \leftarrow X = cons(U, X'), Z = cons(U, Z'), app(X', Y, Z')$$

and the goal

$$X = cons(A, cons(B, nil)),^{①} X = Y,^{②} app(X, Y, Z)^{③}$$

which concatenates the list $[A, B]$ with itself. Imagine that we are analysing this program and goal with descriptions which capture information about "structure sharing" in the solved form of the constraints. Descriptions are sets of variable pairs where a variable pair XY indicates that, in the solved form, variables X and Y may be bound to terms which share variables. A pair of the form XX has a special meaning, namely that the variable X may be bound to a non-linear term, that is, one in which some variable occurs more than once. Similar descriptions are commonly used in analyses for compile-time garbage collection and for determining independence of atoms when parallelising logic programs.

Let us sketch the analysis of *app*—the details will be made clear later. When analysing the goal, at the program point ① we have the description $\{XA, XB\}$, indicating that X possibly "shares" with A and also with B, but A and B do not share. After adding constraint $X = Y$, at program point ②, we have $\{XA, XB, YA, YB, XY\}$. Restricting this to the variables that occur in the call to *app*, we have the description $\{XY\}$. The call *app* is evaluated with this description and gives the description $\{XY, XZ, YZ, XX, YY, ZZ\}$ as its answer. Notice that we need to include the possibility that some variable has become non-linear. In the traditional goal-dependent framework, this answer is combined with the original call $\{XA, XB, YA, YB, XY\}$. This gives rise to the imprecise information that A and B possibly share after the call to *app*. Of course, in reality A and B do not share. This lack of precision occurs because, in effect, the description $\{XY\}$ (due to the constraint $X = Y$) has been added twice, once before the call to *app* and then again on return from *app*.

Given that the goal-independent and goal-dependent approaches are incomparable, it is natural to wish to combine the accuracy of the two approaches. That is the goal of this paper. A partial solution for the case of sharing analysis is given by Mulkers with her "twofold sharing domain" [18]. In Mulkers's approach dataflow information is kept in two parts—a description of the call and a description of the new constraints which have been added while processing the atom. Using this approach, analysis of the above example will give the more precise information that

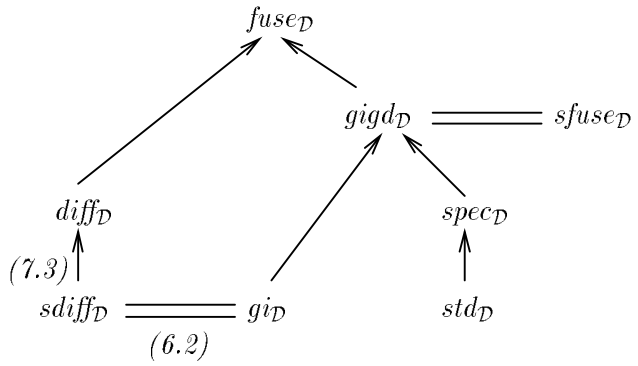

FIGURE 1.1. Relative accuracy of semantics

A and B definitely do not share. This idea is also used by Dumortier [6, 7]. Our aim is to formalise and develop this and allied approaches as *general* methods.

Let us call the Mulkers approach the *simple differential* approach. We show that the simple differential approach is equivalent to the goal-independent analysis in terms of annotated information. This follows from the fact that the component of the description which keeps track of the new constraints exactly mimics a goal-independent analysis, while the input component mimics the annotation phase. In this sense, the advantages of the simple differential approach are that it performs the analysis and annotation in a single phase, and that the traditional generic frameworks do not have to be modified, the approach being implemented simply as yet another description domain (a product domain). We then prove that for some particular description domains, the simple differential approach is at least as accurate as the traditional frameworks, uniformly. This result enlarges the number of description domains amenable to the technique of Jacobs and Langen. However, as in the case of goal-independent based analysis, for some other domains the two approaches are incomparable with respect to accuracy. Furthermore, the efficiency associated to the goal-independent based approaches disappears since, again, the answers to an atom are being re-computed for each particular call.

For these reasons we also introduce a highly versatile approach, referred to as the *differential* approach, which combines all the approaches mentioned above. In its ultimate form it is as least as accurate as any of the other approaches, uniformly.

Figure 1.1 summarises the relative accuracy of the various semantics defined throughout the paper. An arrow $s_1 \leftarrow s_2$ indicates that s_1 is more precise than s_2, while double lines indicate equalities. Where the relationships are conditional on the abstract domain satisfying the conditions of some theorem, the theorem number is given in parentheses next to the arc. The reader is urged to refer back to this figure occasionally to put results in context.

We have implemented all the methods discussed here, and we provide an empirical evaluation of them, for two analysis domains: the structure sharing domain used above and the *ASub* domain [20] which describes pairwise variable sharing. Our results show that differential methods can provide significantly more accurate analyses for a small extra implementation effort, without significantly compromising the efficiency of analysis, and sometimes improving it.

The plan of the paper is as follows: Section 2 introduces terminology and the goals of program analysis based on abstract interpretation. Section 3 considers two variants of so-called goal-dependent analysis while Sections 4 and 5 deal with methods based on so-called goal-independent analysis. Sections 6 and 7 discuss various differential approaches. Section 8 gives results of the empirical evaluation, both concerning efficiency and accuracy. Section 10 contains a concluding discussion.

2. BACKGROUND

In this section we revise the usual operational semantics of constraint logic programs, the basic theory of abstract interpretation, and the goals of program analysis.

A *(constraint logic) program* is a finite set of rules of the form $H \leftarrow B$, where the *head*, H, is an atom and the *body*, B, is a sequence of literals. A *goal* is a (possibly empty) sequence of literals. *Literals* are divided into two classes: the *primitive constraints*, $Prim$, and the programmer-defined *atoms*, $Atom$. Primitive constraints are predefined in the sense that they have an intended meaning or interpretation which, for efficiency, is built into the solver for the language. We will typically use Horn clause programs in examples, and in this case, a primitive constraint is a term equation of the form $x = t$ where x is a variable and t a term. For simplicity we require atoms and terms to be of the form $p(x_1, .., x_n)$ and either x_1 or $f(x_1, .., x_n)$, respectively, where the x_i are distinct variables. Unless otherwise stated, we will assume that the programs are pure, that is, they do not contain calls to non-logical built-ins such as var.

A *constraint* is a (possibly existentially quantified) conjunction of primitive constraints. We will usually consider constraints modulo logical equivalence. We let Con denote the set of all constraints. We let $\overline{\exists}_S e$ denote the constraint e restricted to the variables in S. That is, $\overline{\exists}_S e$ is $\exists V_1 \exists V_2 ... \exists V_n e$ where $\{V_1, V_2, ..., V_n\} = vars(e) \setminus vars(S)$ and the function $vars$ takes a syntactic object and returns the set of (free) variables occurring in it.

A *renaming*, ρ, is a bijection between variables. We let ρ^{-1} denote the inverse of ρ. We naturally extend renamings to mappings between atoms, rules, and constraints. We also allow renamings to distribute over sets and tuples. Thus we may write $\rho(X)$ for the set $\{\rho(x) \mid x \in X\}$ and $\rho\langle x_1, \ldots, x_n\rangle$ for $\langle \rho(x_1), \ldots, \rho(x_n)\rangle$ when convenient. The set of renamings is denoted by Ren. Syntactic objects s and s' are said to be *variants* if there is a renaming ρ such that $\rho(s) = s'$. If s and s' are variants, $rename(s, s')$ returns a renaming ρ such that $\rho(s) = s'$. The *definition of an atom A in program P*, $defn_P(A)$, is the set of rules in P such that each has a variant of A as its head.

The operational semantics of a constraint logic program is in terms of *answers* to its *derivations* which are reduction sequences of *states* where a state is a tuple consisting of the current constraint and the current literal sequence, or "goal". The *operational semantics*, $op(P, G)$, gives the set of answers to goal G for program P. The reader is referred to Jaffar and Maher [12] for more details.

Dataflow analysis is the process of statically (at compile-time) inferring information about the properties of the variables and data structures in a program.

The purpose of this process is to provide information which improves the task performed by compilers, program transformation tools, etc. The key idea behind abstract interpretation based dataflow analysis is that operations and data items in the *concrete* execution of the goal are mimicked by abstract operations and *descriptions* of the data items in the analysis. So the approximation implicitly defines a relation between the concrete and the abstract semantics of the program.

Unfortunately, the operational semantics defined above is not a useful basis for the analysis of logic programs. The reason is that for most purposes of program analysis, such as compiler optimisations, we need detailed information about what happens at each program point. Consider the idea of a logic program interpreter which answers queries by returning not only a set of answer constraints, but also a thoroughly annotated version of the program: For each program point ⓧ, it lists the constraints (projected onto the variables of the rule where ⓧ appears) obtained at ⓧ at some stage during evaluation of the given query. Since control may return to a program point many times during evaluation, each annotation is naturally a (possibly infinite) set of constraints. This idea leads to the notion of a *collecting semantics*, a semantics which gives very precise dataflow information, but which is of course not finitely computable in general. However, if we replace the possibly infinite sets of constraints by more crude descriptions then we may obtain a dataflow analysis which terminates in finite time. The aim of program analysis is, therefore, to take a program and goal and to annotate each program point ⓧ with an approximate description of the constraints which will be encountered at that point when the goal is executed.

Example 2.1. Consider the program

$$\begin{array}{l} le(X,Y) \leftarrow {}^{①}X = Y^{②} \\ le(X,Y) \leftarrow {}^{③}Y = s(Z),^{④}\, le(X,Z)^{⑤} \end{array}$$

and goal $Y = s(a), le(X,Y)$. In the collecting semantics

$$\begin{array}{l} ① = \{Y = a,\quad Y = s(a)\} \\ ② = \{Y = a \wedge X = a,\quad Y = s(a) \wedge X = s(a)\} \\ ③ = \{Y = a,\quad Y = s(a)\} \\ ④ = \{Y = s(a) \wedge Z = a\} \\ ⑤ = \{Y = s(a) \wedge Z = a \wedge X = a\}. \end{array}$$

If we let the set S of variables stand for "the set of constraints that ground all variables in S" then our analyser should give us the following information at the various program points:

$$\begin{array}{l} ① = \{Y\} \\ ② = \{Y, X\} \\ ③ = \{Y\} \\ ④ = \{Y, Z\} \\ ⑤ = \{Y, Z, X\}. \end{array}$$

Though not very precise, this information tells us that given that the initial call to le has a ground second argument, every subsequent call to le will have a ground second argument, and all answers to le will have both arguments ground.

The *collecting semantics*, $coll(P, G)$, returns a function which maps each program point in a program P to the set of constraints (projected onto the variables of the rule) which are encountered at that program point when executing goal G.

Correctness of dataflow analysis is usually formalised in terms of *abstract interpretation* [4]. In this context, the concrete domain $\mathcal{E}$ (sets of term equations in the above example) and the description domain $\mathcal{D}$ (sets of variables in the example) are usually required to be complete lattices related by a pair of *adjoined* functions. That is, there should be monotonic functions $\alpha : \mathcal{E} \to \mathcal{D}$ and $\gamma : \mathcal{D} \to \mathcal{E}$ such that for all $D \in \mathcal{D}$ and for all $E \in \mathcal{E}$,

$$\alpha(E) \leq_{\mathcal{D}} D \Leftrightarrow E \leq_{\mathcal{E}} \gamma(D).$$

The intention is that $\alpha(E)$ is the most precise approximation that correctly describes every constraint in E, while γ is the "semantic function" for approximations. In Example 2.1 we have:

$$\begin{aligned} \alpha(E) &= \{v \mid \text{every } e \in E \text{ grounds } v\} \\ \gamma(D) &= \{e \mid e \text{ grounds every } v \in D\}. \end{aligned}$$

The notion of approximation is made precise as follows: $D \in \mathcal{D}$ *approximates* $E \in \mathcal{E}$ iff $E \leq \gamma(D)$. We write this $D \propto E$. For *syntactic objects*, such as programs and primitive constraints, $S \propto S'$ iff $S = S'$. We can then extend $\propto$ to function spaces as follows. Consider $f : \mathcal{D}_1 \times \cdots \times \mathcal{D}_n \to \mathcal{D}$ and $g : \mathcal{E}_1 \times \cdots \times \mathcal{E}_n \to \mathcal{E}$. We define $f \propto g$ iff for all $D_1 \in \mathcal{D}_1$, ..., $D_n \in \mathcal{D}_n$ and for all $E_1 \in \mathcal{E}_1$, ..., $E_n \in \mathcal{E}_n$, if $D_i \propto E_i$ for $i = 1, .., n$ then $f(D_1, ..., D_n) \propto g(E_1, ..., E_n)$.

We call a description of $\wp(Con)$ a *constraint description*. A domain $\mathcal{D}$ of constraint descriptions is *downwards closed* if for all $D \in \mathcal{D}$ and $e \in Con$, whenever $D \propto \{e\}$ we also have $D \propto \{e' \mid e' \models e\}$. Similarly D is *upwards closed* if for all $D \in \mathcal{D}$ and $e \in Con$, whenever $D \propto \{e\}$ we also have $D \propto \{e' \mid e \models e'\}$.

An important feature of P. and R. Cousot's "adjoined" framework is that given an operation on the concrete domain, there is a (unique) best operation on the description domain which approximates it. Here "best" is with respect to precision. We shall refer to such "best" operations as the *induced* operations.

In the next two sections we will revise the two main approaches to abstract interpretation based analysis of (constraint) logic programs: goal-dependent analysis and goal-independent based analysis.

3. GOAL-DEPENDENT ANALYSIS

The top-down goal-dependent approach [1, 13, 16] is probably the most common approach to the analysis of (constraint) logic programs. In this section we will define a number of semantics that use this approach in terms of a general goal-dependent semantics.

This semantics is defined implicitly in terms of the following five functions on constraints. The function $conj(E, E')$ "conjoins" two sets of constraints, E and E', by producing all possible conjunctions. The function $add(p, E)$ adds a primitive constraint, p, to each constraint in the set E. The function $lub(\mathbf{E})$ collates a set of sets $\mathbf{E} \subseteq \wp(Con)$ of constraints into a single set of constraints. The functions $restrict(W, E)$ and $extend(W, E)$ respectively restrict the constraints in the set E

to the variables in set W and extend the range of the constraints in E to the variables in W.

$$
\begin{array}{ll}
conj(E, E') & = \{e \wedge e' \mid e \in E \text{ and } e' \in E'\} \setminus \{\mathit{false}\} \\
add(p, E) & = conj(\{p\}, E) \\
lub(\mathbf{E}) & = \bigcup \mathbf{E} \\
restrict(W, E) & = \{\overline{\exists}_W e \mid e \in E\} \\
extend(W, E) & = E
\end{array}
$$

We define the standard semantics in terms of a "general goal-dependent" semantics. This provides a useful skeleton for some of the subsequent semantic definitions as well.

Definition 3.1. The *general goal-dependent semantics* for description domain $\mathcal{D}$ and a program P, $gen_{\mathcal{D}}$, is the least fixpoint of the following equations:

$$
\begin{array}{lll}
gen_{\mathcal{D}}(P, G) & = \mathbf{Body}(G, vars(G), true_{\mathcal{D}}) & \\
\mathbf{Rule}(H \leftarrow B, D) & = let\ W = vars(H \leftarrow B)\ in & \\
 & \quad let\ Ans = \mathbf{Body}(B, W, extend_{\mathcal{D}}(W, D))\ in & \\
 & \quad outrestrict_{\mathcal{D}}(vars(H), Ans) & \\
\mathbf{Body}(nil, W, D) & = D & \text{and } annotate(pp(nil), D) \\
\mathbf{Body}(L : B, W, D) & = \mathbf{Body}(B, W, \mathbf{Lit}(L, W, D)) & \text{and } annotate(pp(L), D) \\
\mathbf{Lit}(L, W, D) & = comb_{\mathcal{D}}(W, D, vars(L), \mathbf{Atom}(L, D)) & \text{when } L \in Atom \\
\mathbf{Lit}(L, W, D) & = add_{\mathcal{D}}(L, D) & \text{when } L \in Con \\
\mathbf{Atom}(A, D) & = let\ Proj = inrestrict_{\mathcal{D}}(vars(A), D)\ in & \\
 & \quad lub_{\mathcal{D}}\{\mathbf{Rule}(R, Proj) \mid R \in defn_P(A)\} &
\end{array}
$$

The intention is that this definition is *parametric*. The domain $\mathcal{D}$ can be varied, together with the following (monotonic) auxiliary functions:

$$
\begin{array}{ll}
comb_{\mathcal{D}} & : \mathcal{P}(Vars) \times \mathcal{D} \times \mathcal{P}(Vars) \times \mathcal{D} \to \mathcal{D} \\
add_{\mathcal{D}} & : Prim \times \mathcal{D} \to \mathcal{D} \\
lub_{\mathcal{D}} & : \mathcal{P}(\mathcal{D}) \to \mathcal{D} \\
inrestrict_{\mathcal{D}} & : \mathcal{P}(Vars) \times \mathcal{D} \to \mathcal{D} \\
outrestrict_{\mathcal{D}} & : \mathcal{P}(Vars) \times \mathcal{D} \to \mathcal{D} \\
extend_{\mathcal{D}} & : \mathcal{P}(Vars) \times \mathcal{D} \to \mathcal{D}
\end{array}
$$

In this way, the same set of semantic equations can be used to define a "standard" semantics, as well as a series of non-standard semantics, that is, dataflow analyses [16].

The first equation says that, given program P, the meaning of a goal G is the description obtained by treating G as a rule body. The description $true_{\mathcal{D}}$ denotes $\alpha_{\mathcal{D}}(\{true\})$. For simplicity, we assume that the analysis starts with $true_{\mathcal{D}}$ as the initial calling pattern. Although it is straightforward to modify the semsntics to start from an arbitrary calling pattern, we prefer not to do it so as to maintain a high degree of congruence between the concrete and the abstract semantics.

Rule bodies are treated according to the two equations for **Body**. For an empty body, the current description is returned. A non-empty body has its literals processed one by one, from left to right. In the case of a primitive constraint, processing simply means "adding" it to the current constraint. Otherwise it is an atom, and the result is found by collating all possible results returned from the entire set of rules. However, care must be exercised to ensure that descriptions only involve local variables. When an atom "calls" a rule, the current description is restricted so as to mention only the calling atom's variables (last equation). Then, upon return, the resulting description is combined with the original unconstrained call description (third last equation). For this to be possible, it is usually necessary to know the entire set of variables in the calling rule, and the parameter W records this.

To find the result of a rule $H \leftarrow B$ in the context of a current description D, the description is first extended to include all rule variables. The extended description is input to the body B, and the corresponding description is the result of the rule, after suitable restriction.

In the general semantics there is an implicit global variable which is the function mapping program points to their set of descriptions. This function is modified by the call $annotate(pp(L), D)$ which takes the least upper bound[1]of the description D and the current description of the program point $pp(L)$ before the literal L. For simplicity, we assume that each literal and end of the clause (nil) is uniquely associated with a program point. This could be formalised using an extra argument in all functions and modifying the functions so that they also return the annotation function. We have chosen not to do so because it confuses the essence of the semantics[2]. Note that we regard the general goal-dependent semantics as returning this annotation function.

The general semantic equations together with the parametric functions specify a dataflow analysis. The analysis can be implemented using memoization or tabulation. The least fixpoint is reached via the Kleene sequence and in which only those values of a denotation actually required are computed. In such an implementation termination is guaranteed, provided the number of constraint descriptions with a particular domain of variables is finite and the calls are treated modulo variable renaming. A number of generic abstract interpretation engines based on essentially the above semantic equations have been built, for example [19, 15, 14]. The engines provide sophisticated fixpoint algorithms and data structures.

Specific analyses are obtained by defining a description domain $\mathcal{D}$ and providing the seven parametric functions over $\mathcal{D}$. The following specialisation of the general semantics captures the usual theoretical viewpoint of goal-dependent analysis.

Definition 3.2. The *standard goal-dependent semantics* for description domain $\mathcal{D}$ and a program P, $std_{\mathcal{D}}$ is obtained from the general semantics by requiring that

$$comb_{\mathcal{D}}(W, D, W', D') = conj_{\mathcal{D}}(D, extend_{\mathcal{D}}(W, D')),$$

and that the auxiliary functions $conj_{\mathcal{D}}$, $add_{\mathcal{D}}$, $lub_{\mathcal{D}}$, $inrestrict_{\mathcal{D}}$, $outrestrict_{\mathcal{D}}$ and $extend_{\mathcal{D}}$ approximate $conj$, add, lub, $restrict$, $restrict$ and $extend$, respectively. The standard semantics *induced* for $\mathcal{D}$ is the semantics obtained by inducing the auxiliary functions from $comb$, add, lub, $restrict$, $restrict$ and $extend$ respectively. Note that the induced semantics is completely defined by the de-

[1] Instead of taking a least upper bound, we could collect sets of descriptions. This makes no difference to our subsequent results.

[2] Also for reasons of legibility we have taken liberties with the definition, which strictly speaking is not denotational, since **Atom** is defined with reference to the entire programs and therefore fails to be compositional. It is straightforward to fix this problem along the lines of [16], but we prefer not to complicate the definition further.

scription domain $\mathcal{D}$. The standard semantics induced when $\mathcal{D}$ is $\mathcal{P}(Con)$ is written *conc* and called the *concrete top-down semantics.*

It follows from standard results in abstract interpretation theory that for any description domain $\mathcal{D}$, $std_{\mathcal{D}}$ approximates the collecting semantics [16].

Proposition 3.1. If $std_{\mathcal{D}}$ is induced from description domain D, then for all programs P and goals G: $std_{\mathcal{D}}(P, G) \propto coll(P, G)$.

Example 3.1. Consider the abstract domain SS for structure sharing. Descriptions are sets of pairs of variables. The pair XY indicates that in the solved form of the constraints, variables X and Y may share a common structure, that is, point to the same piece of memory. The pair XX indicates that the solved form of X may have two different paths to the same piece of memory (also referred to as X being non-linear). Note that the pairs are symmetric and that $true_{SS}$ is the empty set. The structure sharing domain was introduced by Søndergaard [20] as a component of an analysis for determining when dropping the occur check in unification is safe. Structure sharing information is useful, for example, in compile-time garbage collection, as the term pointed to by a variable can be reclaimed if the variable and all variables with which it shares structure are not subsequently referred to in the text of the program.

The abstract operations for structure sharing are defined as follows:

$$
\begin{array}{ll}
conj_{SS}(D, D') & = \{XY \mid \text{there is a path from } X \text{ to } Y \text{ using arcs} \\
 & \qquad\qquad\qquad \text{alternately from } D \text{ and } D'\} \\
add_{SS}(p, D) & = conj_{SS}(\{XT_i \mid 1 \leq i \leq n\}, D) \\
 & \qquad\quad \text{where } p \text{ is } X = f(T_1, \ldots, T_n) \text{ or } X = T_1 \\
lub_{SS}(\mathbf{D}) & = \bigcup \mathbf{D} \\
inrestrict_{SS}(W, D) & = \{XY \in D \mid X, Y \in W\} \\
outrestrict_{SS}(W, D) & = \{XY \in D \mid X, Y \in W\} \\
extend_{SS}(W, D) & = D.
\end{array}
$$

Consider the following simple program and goal g

$$
\begin{array}{ll}
g & \leftarrow X = cons(A, Z), Y = cons(B, Z), ^{①}\, p(X, Y)^{②} \\
p(X, Y) & \leftarrow\, ^{③}X = Y^{④}
\end{array}
$$

Throughout the example we will ignore variable Z for simplicity. The description at point ① is $\{XA, YB, XY\}$. Projecting this description we arrive at the description at ③ of $\{XY\}$. Adding the constraint $X = Y$, at ④ we obtain

$$add_{SS}(X = Y, \{XY\}) = conj_{SS}(\{XY\}, \{XY\}) = \{XY, XX, YY\}.$$

At ② the result is $comb_{SS}(\{X, Y, A, B\}, \{XA, YB, XY\}, \{X, Y\}, \{XY, XX, YY\})$ which is

$$\{XA, YB, XY, XX, YY, XB, YA, AA, BB, AB\}.$$

Note that the possible sharing approximated by the pairs XX, XY, AA and BB in the above description, is in fact not possible.

For simplicity, the standard semantics has been the approach used in the theoretical definition of many abstract domains and analyses. In contrast, most implementations have used a more accurate and/or efficient, but more complex, definition of abstract combination. This is because in practice, safety of the analysis does not require the abstract combination operation $comb_{\mathcal{D}}(W, D, W', D')$ to be defined in terms of a general abstract conjunction operation. Correctness of the implementation definition is based on a deeper understanding of the relationship between D and D'.

Imagine that we are evaluating the atom A after a program point with associated constraint set E in the standard collecting semantics. On return from A we will call $comb(W, E, vars(A), E')$ where E' has been obtained by first restricting E to $vars(A)$, adding constraints, and then restricting back to $vars(A)$. In other words, for some set of constraints E'' (those added by evaluating A),

$$E' = conj(restrict(vars(A), E), restrict(vars(A), E'')).$$

For correctness it suffices that the abstract operation *comb* is correct for this particular use—it does not need to safely approximate $conj(E_1, E_2)$ for arbitrary E_1 and E_2.

Note that E' already contains all the information about the variables in A, and the role of the combination is just to extend this information to the rest of variables in W. This property carries over to an abstract domain, where if $D' \propto E'$ then D' already contains all the information about the variables in A. Naively conjoining the descriptions D and D' may unnecessarily weaken the information about the variables in A. This is because, first, the information in D about the variables in A which is also in D' is considered twice, and second, D may contain information about the variables in A which is no longer valid and should not affect the result of the combination.

Definition 3.3. Let $W' \subseteq W \subseteq Var$, $D, D' \in \mathcal{D}$ and $E, E'' \in \wp(Con)$, and let $E' = conj(restrict(W', E), restrict(W', E''))$. The *specialised abstract combine* operation $comb^{spec}_{\mathcal{D}}$ is *safe* if, whenever $D \propto E$ and $D' \propto E'$, we also have

$$comb^{spec}_{\mathcal{D}}(W, D, W', D') \propto conj(E, E').$$

We can modify our standard semantics so that it is defined in terms of a combine operation satisfying the above condition, rather than in terms of conjunction. This formalises the approach taken in some of the implementations of common domains such as *Sharing* (set sharing), *ASub* (pair sharing), and *Def* (definiteness dependencies), as for example, those embedded in PLAI [19] and GAIA [15].

Definition 3.4. The *specialised semantics*, $spec_{\mathcal{D}}$, is identical to the standard semantics except that the abstract combination $comb_{\mathcal{D}}$ is replaced by $comb^{spec}_{\mathcal{D}}$ which is required to be safe. The specialised semantics *induced* for D is the semantics obtained by inducing the auxiliary functions from *add*, *lub*, *restrict*, *restrict* and *extend* respectively, and using the least $comb^{spec}_{\mathcal{D}}$ which is safe.

In general, for a given description domain the specialised semantics will always be correct and at least as precise as the standard semantics.

Proposition 3.2. Let $spec_{\mathcal{D}}$ and $std_{\mathcal{D}}$ be induced from description domain $\mathcal{D}$. For all programs P and goals G: $spec_{\mathcal{D}}(P,G) \propto coll(P,G)$ and $spec_{\mathcal{D}}(P,G) \leq std_{\mathcal{D}}(P,G)$.

Example 3.2. Consider the structure sharing abstract domain SS. The specialised abstract combine operation may be defined by:

$$\begin{aligned} comb_{SS}^{spec}(W,D,W',D') &= \text{let } D'' = D \setminus \{XY \mid X,Y \in W'\} \text{ in} \\ &\quad\; D'' \cup new_{SS}(D'',D') \\ new_{SS}(D,D') &= \bigcup_{XY \in D'} \{UV \mid (U = X \vee UX \in D) \wedge (Y = V \vee YV \in D)\}. \end{aligned}$$

Now consider the following program and goal g, almost identical to that in Example 3.1.

$$\begin{array}{ll} g & \leftarrow X = cons(A,Z), Y = cons(B,Z), ^{①}\, q(X,Y)^{②} \\ q(X,Y) & \leftarrow\, ^{③} true^{④} \end{array}$$

Again we will ignore variable Z for simplicity. Using both the standard and the specialised semantics for SS, we have the description $\{XA, YB, XY\}$ at ①, $\{XY\}$ at ③, and hence also at ④. With the standard semantics the description at ② is the result obtained from $comb_{SS}(\{X,Y,A,B\}, \{XA, YB, XY\}, \{X,Y\}, \{XY\})$, that is, $\{XA, YB, XY, XX, YY, XB, YA, AB\}$. With the specialised semantics the description at ② is given by

$$\begin{array}{ll} & comb_{SS}^{spec}(\{X,Y,A,B\}, \{XA, YB, XY\}, \{X,Y\}, \{XY\}) \\ = & \{XA, YB\} \cup new_{SS}(\{XA, YB\}, \{XY\}) \\ = & \{XA, YB, XY, XB, YA, AB\}. \end{array}$$

The result with the specialised semantics is more accurate than that obtained with the standard semantics as it does not contain the pairs XX and YY. Even more importantly, because the definition of $comb_{SS}^{spec}$ does not require examining paths of arbitrary length, the specialised semantics can be computed more efficiently than the standard semantics. For the program and goal from Example 3.1 no improvement is obtained.

The specialised abstract combination function can also take advantage of cases where the return description D' invalidates information in the call. For upwards closed domains such as SS this is only possible with built-ins and other non-logicals, but for non-upwards closed domains it can occur without them.

Example 3.3. Using the structure sharing domain, SS, the built-in $X == Y$ is treated as follows. Since X and Y must be bound to the same variable or term, they can only share with the same things. Thus any existing sharing pair XA can be removed unless YA also exists.

$$\begin{array}{ll} g & \leftarrow q(X,Y,A), ^{①}\, p(X,Y,A)^{②} \\ q(X,Y,A) & \leftarrow X = Y \\ q(X,Y,A) & \leftarrow X = f(A,Y) \\ p(X,Y,A) & \leftarrow\, ^{③} X == Y^{④} \end{array}$$

At ① and ③ the description is $\{XY, XA\}$. After the built-in at ④, the description is $\{XY\}$ since the XA sharing is not possible. Upon return, the description at ② using the standard approach is again $\{XY, XA, XX, YY, YA\}$, while the specialised combination yields $\{XY\}$.

Example 3.4. Consider the abstract domain $ASub$ [20] which describes pairwise variable sharing. A description $[G, VS]$ has two components, a groundness description G as in Example 2.1, and a structure sharing description VS as in the abstract domain SS. Groundness information is used to eliminate any pair in the sharing part which involves a ground variable, hence only approximating variable sharing as opposed to arbitrary structure sharing. Clearly, $true_{ASub}$ is simply $[\emptyset, \emptyset]$. Note that the results of the previous examples (using SS) would be unaltered if we used $ASub$, since no groundness is present. Consider the following program and goal g:

$$
\begin{array}{ll}
g & \leftarrow X = Y, ① \, r(X,Y) ② \\
r(X,Y) & \leftarrow ③ X = a ④ \\
r(X,Y) & \leftarrow ⑤ Y = a ⑥
\end{array}
$$

At ①, ③ and ⑤ the description is $[\emptyset, \{XY\}]$. At ④ the description is $[\{X\}, \emptyset]$, since X has become ground and therefore the sharing is removed. Similarly, at ⑥ the description is $[\{Y\}, \emptyset]$. The lub of these two descriptions is $[\emptyset, \emptyset]$. Using the standard approach, the description at ② is $[\emptyset, \{XY\}]$. Using specialised combination, the pair XY can be removed since it does not appear in the lub description.

4. GOAL-INDEPENDENT BASED ANALYSIS

In goal-dependent analysis, the answers to a literal can be computed several times, once for each different abstract calling pattern. This re-computation can increase the accuracy but also reduce the efficiency of the approach. Goal-independent based analysis avoids this problem by noticing that in the operational semantics, if E is the set of answers to the goal G, then the set of answers to $e \wedge G$ is just $add(e, E)$ [11].

For program analysis, this means that when finding the answers to an atom A for a particular calling pattern of constraints, E, we can first find the answers, E', to A for the calling pattern $\{true\}$, and then conjoin E with E' to give the answers for the calling pattern E. This has the advantage that we only need to compute the answers to each atom in the program for one calling pattern; the answers to other calling patterns can be obtained from this by a simple conjunction. The disadvantage is that we now need two phases in the analysis: a first phase to compute the answers to each atom for $true_{\mathcal{D}}$, and a second, goal-dependent, phase which uses this information to annotate the program for a given goal. This approach to program analysis is formalised in the following semantic equations.

Definition 4.1. The *goal-independent based semantics* for description domain $\mathcal{D}$ and a program P, $gi_{\mathcal{D}}$, is the least fixpoint of the following two equation sets.

The first goal-independent set computes the answers to an atom.

$$
\begin{array}{lll}
\mathbf{Rule_{gi}}(H \leftarrow B) & = & let\ W = vars(H \leftarrow B)\ in \\
 & & let\ Ans = \mathbf{Body_{gi}}(B, W, extend_{\mathcal{D}}(W, true_{\mathcal{D}}))\ in \\
 & & outrestrict_{\mathcal{D}}(vars(H), Ans) \\
\mathbf{Body_{gi}}(nil, W, D) & = & D \\
\mathbf{Body_{gi}}(L : B, W, D) & = & \mathbf{Body_{gi}}(B, W, \mathbf{Lit_{gi}}(L, W, D)) \\
\mathbf{Lit_{gi}}(L, W, D) & = & conj_{\mathcal{D}}(D, extend_{\mathcal{D}}(W, \mathbf{Atom_{gi}}(L)))\ \text{when}\ L \in Atom \\
\mathbf{Lit_{gi}}(L, W, D) & = & add_{\mathcal{D}}(L, D) \qquad \text{when}\ L \in Con \\
\mathbf{Atom_{gi}}(A) & = & lub_{\mathcal{D}}\{\mathbf{Rule_{gi}}(R) \mid R \in defn_P(A)\}
\end{array}
$$

The second set annotates the program for a particular goal.

$$
\begin{array}{lll}
gi_{\mathcal{D}}(P, G) & = & \mathbf{AnnBody}(G, vars(G), true_{\mathcal{D}}) \\
\mathbf{AnnRule}(H \leftarrow B, D) & = & let\ W = vars(H \leftarrow B)\ in \\
 & & \mathbf{AnnBody}(B, W, extend_{\mathcal{D}}(W, D)) \\
\mathbf{AnnBody}(nil, W, D) & = & annotate(pp(nil), D) \\
\mathbf{AnnBody}(L : B, W, D) & = & \mathbf{AnnLit}(L, W, D)\ \text{and} \\
 & & \mathbf{AnnBody}(B, W, \mathbf{Lit_{gi}}(L, W, D)) \\
\mathbf{AnnLit}(L, W, D) & = & annotate(pp(L), D)\ \text{and} \\
 & & \mathbf{AnnAtom}(L, W, D) \quad \text{when}\ L \in Atom \\
\mathbf{AnnLit}(L, W, D) & = & annotate(pp(L), D) \qquad \text{when}\ L \in Con \\
\mathbf{AnnAtom}(A, W, D) & = & \text{for each}\ R \in defn_P(A), \\
 & & \mathbf{AnnRule}(R, inrestrict_{\mathcal{D}}(vars(A), D)).
\end{array}
$$

The goal-independent based semantics *induced* for D is the semantics obtained by inducing the auxiliary functions from *conj*, *add*, *lub*, *restrict*, *restrict* and *extend* respectively. The goal-independent based semantics induced from D chosen to be $\mathcal{P}(Con)$ is written *concgi* and called the *concrete goal-independent based semantics.*

Example 4.1. Recall the program from Example 2.1:

$$
\begin{array}{l}
le(X, Y) \leftarrow {}^{①}X = Y^{②} \\
le(X, Y) \leftarrow {}^{③}Y = s(Z),^{④}\, le(X, Z)^{⑤}
\end{array}
$$

with goal $Y = s(a), le(X, Y)$. If this is analysed using the goal-independent based approach with the simple groundness descriptions used in Example 2.1 we obtain in the first phase that the answers to $le(X, Z)$ are described by $\emptyset$. In the second phase we will compute the annotations

$$
\begin{array}{l}
① = \{Y\} \\
② = \{Y, X\} \\
③ = \{Y\} \\
④ = \{Y, Z\} \\
⑤ = \{Y, Z\}.
\end{array}
$$

Annotations ①, ②, ③ and ④ are computed as in the standard goal-dependent analysis. Annotation ⑤ is computed by conjoining the answer information for $le(X, Z)$ with the calling pattern at ④. That is, at ⑤ we have

$$conj_{gnd}(\{Y, Z\}, \emptyset) = \{Y, Z\} \cup \emptyset = \{Y, Z\}.$$

Note that the standard semantics finds that X is ground at ⑤. The goal-independent approach does not find this because of the weak description of the answers to $le(X, Z)$.

At first glance, it seems clear that the goal-independent based approach to analysis will be more efficient but less accurate than the goal-dependent approach. For this reason, when Jacobs and Langen [11] proposed this method (referred to as condensation) they focused on defining the characteristics that the abstract domain functions should satisfy in order not to lose accuracy.

Codish *et al.* [3] conducted empirical testing on a variant of the approach defined above. They determined the relative efficiency of the goal-independent based approach versus the specialised goal-dependent approach on three different domains. Their results indicated that relative efficiency depended on the domain and the number of different calling patterns to each atom. The accuracy for domains which do not satisfy all the required characteristics was usually decreased by the method. The few cases in which accuracy was improved were attributed to the change in the order in which constraints are analysed.

However, the relationship between the efficiency and precision of the two approaches is more complex. It is clear from Example 4.1 that the goal-dependent approach can be more accurate. However, the goal-independent based approach may also be more accurate, and, in a certain sense, is inherently more accurate.

The first point to note is that the concrete goal-independent based semantics is as accurate as the collecting semantics. This is because the first phase of the semantics is exactly the S-semantics [8] which has been shown to be equivalent to the operational semantics.

Theorem 4.1. For all programs P and goals G: $concgi(P, G) = coll(P, G)$.

In contrast, the concrete goal-dependent and specialised semantics are inherently imprecise. Imagine that the program point immediately before atom A has description D. Then A will be evaluated with the call pattern $restrict(vars(A), D)$ and on return from A with result D' say, D' is conjoined with D to give the description at the program point after A. The problem is that the constraints in $restrict(vars(A), D)$ are essentially being added twice, and the $conj$ operation is not idempotent. The following example is from Marriott *et al.* [16].

Example 4.2. Consider the following program with goal $g(X, Y, Z)$.

$$
\begin{array}{ll}
g(X,Y,Z) & \leftarrow {}^{①}p(X,Y),{}^{②}\, r(X,Y,Z)^{③} \\
p(X,Y) & \leftarrow X = a \\
p(X,Y) & \leftarrow Y = a \\
r(X,Y,Z) & \leftarrow Y = Z
\end{array}
$$

In the operational semantics the goal has two answers: $X = a \wedge Y = Z$ and $Y = a \wedge Y = Z$. In the concrete standard semantics we have $\{true\}$ at ① and the two constraints $\{X = a,\ \ Y = a\}$ at ②. Projecting onto X, Y and Z gives the same set. On return from r, we have the two answers $\{X = a \wedge Y = Z,\ \ Y = a \wedge Y = Z\}$. Combining this with $\{X = a,\ \ Y = a\}$ yields the set $\{X = a \wedge Y = Z,\ \ Y = a \wedge Y = Z,\ \ X = a \wedge Y = a \wedge Y = Z\}$ at ③. Thus, in the concrete standard semantics the goal has an extra answer.

In program analysis this problem is exacerbated because for many description domains adding a description twice is *very* different to adding it once. The following example illustrates that real analysis domains suffer a large loss of precision. In such circumstances, the goal-independent based approach will be more precise than the standard or specialised goal-dependent semantics. This is confirmed by the experimental results reported in Section 8.

Example 4.3. Recall the program and goal from Example 3.2.

$$\begin{array}{ll} g & \leftarrow X = cons(A,Z), Y = cons(B,Z), ^{①}\, q(X,Y)^{②} \\ q(X,Y) & \leftarrow {}^{③}true^{④} \end{array}$$

Again we will ignore variable Z for simplicity. Using the SS domain and the goal-independent semantics we obtain in the first phase that the answer to $q(X,Y)$ is just the empty set. In the annotation phase, we again have $\{XA, YB, XY\}$ at ① and $\{XY\}$ at ③ and ④. The result at ② is given by

$$conj_{SS}(\{XA, YB, XY\}, \emptyset) = \{XA, YB, XY\}.$$

This is more accurate than the descriptions obtained with either the standard or specialised goal-dependent approaches. Accuracy was lost in the goal-dependent approaches because the input to the call to q, $\{XY\}$, could not be distinguished from new structure sharing that may have arisen in the call.

In the following sections we look at methods to combine the goal-independent-based and goal-dependent approaches so as to further improve precision.

5. IMPROVING GOAL-INDEPENDENT BASED ANALYSIS

As seen in the previous section, in order to use goal-independent analysis information to annotate a program, a separate annotation phase is required. This annotation phase has similarities with a goal-dependent analysis, but it only calculates calling pattern information in terms of the goal-independent answer information. It never uses the information generated in annotating the program to provide (possibly more accurate) answer information.

We can do this by simply replacing the annotation phase in the goal-independent based analysis by a goal-dependent analysis which uses both the goal-independent and goal-dependent answers to determine the result of calling an atom. The greatest lower bound of the two answers is used to obtain a more accurate result. To define such an approach we only need change the definition of **Atom** in the general goal-dependent semantics to make use of the goal-independent information. Note that a first phase is still required to compute the goal-independent information.

Definition 5.1. The *gigd semantics* for description domain $\mathcal{D}$ and a program P, $gigd_{\mathcal{D}}$, is the least fixpoint of the equations of the general goal-dependent semantics with the definition of **Lit** replaced by

$$\begin{array}{ll}\mathbf{Lit}(L,W,D) = comb^{spec}_{\mathcal{D}}(W,D,vars(A),Atom(L,D)) & \\ \qquad\qquad\qquad\quad \sqcap \mathbf{Lit}_{\mathbf{gi}}(L,W,D) & \text{when } L \in Atom \\ \mathbf{Lit}(L,W,D) = add_{\mathcal{D}}(L,D) & \text{when } L \in Con\end{array}$$

The *gigd* semantics induced for domain D, $gigd_{\mathcal{D}}$, is the semantics obtained by inducing the auxiliary functions $comb^{spec}_{\mathcal{D}}$, $add_{\mathcal{D}}$, $inrestrict_{\mathcal{D}}$, $outrestrict_{\mathcal{D}}$, $extend_{\mathcal{D}}$, $conj_{\mathcal{D}}$, and $lub_{\mathcal{D}}$, $extend_{\mathcal{D}}$, $conj_{\mathcal{D}}$.

Example 5.1. The following program (with goal g) combines features of Examples 3.4 and 4.3.

$$\begin{array}{ll} g & \leftarrow X = f(Y,Z), ① \; p(X,Y), ② \; r(X,Y) ③ \\ p(X,Y) & \leftarrow true \\ r(X,Y) & \leftarrow X = a \\ r(X,Y) & \leftarrow Y = a \end{array}$$

Consider the *ASub* domain. Goal-independent analysis determines the answers to $p(X,Y)$ and $r(X,Y)$ to both be $[\emptyset,\emptyset]$ since they introduce no sharing, or definite groundness. The three analysis methods: $spec_{Asub}$, gi_{Asub} and $gigd_{Asub}$ proceed as follows:

pgm point	$spec_{Asub}$	gi_{Asub}	$gigd_{Asub}$
①	$[\emptyset,\{XY,XZ\}]$	$[\emptyset,\{XY,XZ\}]$	$[\emptyset,\{XY,XZ\}]$
②	$[\emptyset,\{XY,XZ,YZ\}]$	$[\emptyset,\{XY,XZ\}]$	$[\emptyset,\{XY,XZ\}]$
③	$[\emptyset,\{XZ,YZ\}]$	$[\emptyset,\{XY,XZ\}]$	$[\emptyset,\{XZ\}]$

Clearly the *gigd* approach surpasses the other two approaches.

Theorem 5.1. For fixed monotonic auxiliary operations $comb^{spec}_{\mathcal{D}}$, $add_{\mathcal{D}}$, $inrestrict_{\mathcal{D}}$, $outrestrict_{\mathcal{D}}$, $extend_{\mathcal{D}}$, $conj_{\mathcal{D}}$, and $lub_{\mathcal{D}}$, we have: $gigd_{\mathcal{D}}(P,G) \leq gi_{\mathcal{D}}(P,G)$ and $gigd_{\mathcal{D}}(P,G) \leq spec_{\mathcal{D}}(P,G)$ for all goals G and programs P.

The *gigd* semantics is loosely related to that proposed by Codish *et al.* [3]. Codish *et al.* noticed that with little additional work, the annotation phase of a goal-independent based analysis could become a simplified form of goal-dependent analysis. For each call, they used either the goal-independent or the goal-dependent information to compute the answer rather than both as in the *gigd* semantics. In essence, they chose to use goal-independent answer information only for recursive atoms with calling patterns that had previously been encountered. Hence they in effect short-circuited the evaluation of recursive goals. The reason for doing this was to maintain the accuracy of goal-dependent analysis as far as possible while keeping the efficiency of goal-independent based analyses for recursive predicates. This contrasts with the *gigd* semantics which performs both goal-dependent and goal-independent analysis to find the answers to each call. This gives better accuracy at the cost of some efficiency.

6. SIMPLE DIFFERENTIAL ANALYSIS

The *gigd* semantics is one approach to the combination of goal-dependent and goal-independent based approaches. Because of its two phase nature, it can be viewed as a modification of the goal-independent based approach. In the next two sections we propose different approaches that, while still taking advantage of the goal-independent information, do not require two phases, and thus can be viewed as a modification of the goal-dependent approach.

We begin by describing the *simple* differential approach which performs goal-independent based analysis in a goal-dependent framework. In this approach, descriptions are kept in two components—a description of the call to the rule, D_{in}, and a description of the new constraints which have been added while processing the rule, D_δ. The description of the current program point is obtained by conjoining these descriptions together. The idea is the following. Assume that the analysis is processing the rule

$$H \leftarrow {}^{⓪}A_1, ..., A_{x-1}^{ⓩ}, A_x^{ⓧ}, ..., A_n \qquad (z = x - 1)$$

and originally H was called with the description D_{in}. The description at ⓪ will be $\langle D_{in}, true_{\mathcal{D}}\rangle$ indicating that the call is D_{in} and that, so far, no constraints have been added while processing this rule. If processing $A_1, \ldots, A_{x-1}$ adds the constraints described by D_δ, then the description at ⓩ will be $\langle D_{in}, D_\delta\rangle$. The call to A_x proceeds as follows. The call to A_x, D'_{in} say, is $restrict(vars(A_x), D_{tot})$ where $D_{tot} = conj_{\mathcal{D}}(D_{in}, D_\delta)$. Processing A_x will give rise to an answer of the form $\langle D'_{in}, D'_\delta\rangle$, indicating that evaluation of A_x has added constraints described by D'_δ. On return from A_x, we must add the information about new constraints to D_δ. The description at ⓧ is therefore $\langle D_{in}, D''_\delta\rangle$ where D''_δ is the abstract conjunction of D_δ and D'_δ. Note that the input description D'_{in} is only used for computing information about program points—it is ignored when computing the answer.

Definition 6.1. Let D be a description domain. The *differential abstract domain* $\delta\mathcal{D}$ is just $\mathcal{D} \times \mathcal{D}$. Abstract operations on $\delta\mathcal{D}$ are given by:

$$\begin{array}{l}
true_{\delta\mathcal{D}} = \langle true_{\mathcal{D}}, true_{\mathcal{D}}\rangle \\
add_{\delta\mathcal{D}}(p, \langle D_{in}, D_\delta\rangle) = \langle D_{in}, add_{\mathcal{D}}(p, D_\delta)\rangle \\
lub_{\delta\mathcal{D}}(\mathbf{Y}) = \langle lub_{\mathcal{D}}(\mathbf{Y}_{in}), lub_{\mathcal{D}}(\mathbf{Y}_\delta)\rangle \quad \text{where } \mathbf{Y}_{in} = \{D_{in} \mid \langle D_{in}, D_\delta\rangle \in \mathbf{Y}\} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{and } \mathbf{Y}_\delta = \{D_\delta \mid \langle D_{in}, D_\delta\rangle \in \mathbf{Y}\} \\
inrestrict_{\delta\mathcal{D}}(W, \langle D_{in}, D_\delta\rangle) = \langle inrestrict_{\mathcal{D}}(W, conj_{\mathcal{D}}(D_{in}, D_\delta)), true_{\mathcal{D}}\rangle \\
outrestrict_{\delta\mathcal{D}}(W, \langle D_{in}, D_\delta\rangle) = \langle D_{in}, outrestrict_{\mathcal{D}}(W, D_\delta)\rangle \\
extend_{\delta\mathcal{D}}(W, \langle D_{in}, D_\delta\rangle) = \langle extend_{\mathcal{D}}(W, D_{in}), extend_{\mathcal{D}}(W, D_\delta)\rangle \\
comb_{\delta\mathcal{D}}(W, \langle D_{in}, D_\delta\rangle, W', \langle D'_{in}, D'_\delta\rangle) = \langle D_{in}, conj_{\mathcal{D}}(D_\delta, extend_{\mathcal{D}}(W, D'_\delta))\rangle
\end{array}$$

The *simple differential semantics*, $sdiff_{\mathcal{D}}$, for description domain $\mathcal{D}$ and a program P is the standard goal-dependent semantics using $\delta\mathcal{D}$ and the above abstract operations.

The simple differential semantics *induced* for $\mathcal{D}$ is the semantics obtained by inducing the auxiliary functions on $\mathcal{D}$ from $conj$, add, lub, $restrict$, $restrict$ and $extend$. The simple differential semantics induced when $\mathcal{D}$ is $\mathcal{P}(Con)$ is written *concdiff* and called the *concrete differential semantics.*

Example 6.1. Recall the program from Example 4.2 :

$$\begin{array}{ll} g(X,Y,Z) & \leftarrow {}^{①}p(X,Y),^{②}\, r(X,Y,Z)^{③} \\ p(X,Y) & \leftarrow {}^{④}X = a^{⑤} \\ p(X,Y) & \leftarrow {}^{⑥}Y = a^{⑦} \\ r(X,Y,Z) & \leftarrow {}^{⑧}Y = Z^{⑨} \end{array}$$

and the goal $g(X,Y,Z)$. In the concrete differential semantics we will have $\langle\{true\},\{true\}\rangle$ at ① indicating that the calling constraint was $\{true\}$ and that no constraints have been added so far. We will conjoin these to give $\{true\}$ which is the "in" component of the call to p. Thus the description at ④ and at ⑥ is $\langle\{true\},\{true\}\rangle$. Processing $X = a$ adds the constraint to the "diff" component. Thus, the description at ⑤ is $\langle\{true\},\{X = a\}\rangle$. Similarly, the description at ⑦ is $\langle\{true\},\{Y = a\}\rangle$. On return from p we lub these together, giving $\langle\{true\},\{X = a, Y = a\}\rangle$. This gives $\langle\{true\},\{X = a, Y = a\}\rangle$ at ② because

$$conj(\{true\},\{X = a, Y = a\}) = \{X = a, Y = a\}.$$

Projecting onto X, Y and Z gives $\{X = a, Y = a\}$ so this is the call to $r(X,Y,Z)$. Thus at ⑧ we have $\langle\{X = a, Y = a\},\{true\}\rangle$. Processing r we add the constraint $Y = Z$ to the "diff" component giving $\langle\{X = a, Y = a\},\{Y = Z\}\rangle$. On return from r, we combine $\langle\{true\},\{X = a, Y = a\}\rangle$ with $\langle\{X = a, Y = a\},\{Y = Z\}\rangle$. The "in" component remains unchanged from ②—indeed it never changes in a rule body. The "diff" component is obtained by conjoining the two "diff" components. This gives the description $\langle\{true\},\{X = a \wedge Y = Z,\ \ Y = a \wedge Y = Z\}\rangle$ at ③.

In the above example the simple differential approach is more precise than the standard approach for the concrete domain and it gives the same result as the operational semantics. We shall see that this is always true. This is because the simple differential semantics is equivalent to the goal-independent semantics in terms of answer information. This is due to the fact that the differential component of the description exactly mimics a goal-independent analysis.

Theorem 6.1. For all atoms A, if $\mathbf{Atom}(A,\langle D_{in}, D_\delta\rangle) = \langle D'_{in}, D'_\delta\rangle$ in the simple differential semantics then $D'_\delta = \mathbf{Atom_{gi}}(A)$.

PROOF. By induction, examining the steps performed by the goal-independent analysis at each stage, which are identical to those performed on the differential component. □

Notice that the differential semantics annotates a program with tuples of the form $\langle D_{in}, D_\delta\rangle$. In terms of useful annotation information, this represents the description $conj_{\mathcal{D}}(D_{in}, D_\delta)$. Hence, in an abuse of notation, when comparing a differential semantics, such as $sdiff_{\mathcal{D}}(P,G)$, with some other semantics, $sem_{\mathcal{D}}(P,G)$ we will use the expression $diff_{\mathcal{D}}(P,G) \leq sem_{\mathcal{D}}(P,G)$ to mean that at each program point ⓧ if the annotation given by $diff_{\mathcal{D}}(P,G)$ is $\langle D_{in}, D_\delta\rangle$, and that given by $sem_{\mathcal{D}}(P,G)$ is D_{sem}, then $conj_{\mathcal{D}}(D_{in}, D_\delta) \leq D_{sem}$.

Although they are essentially the same analysis, the goal-independent based analysis and the simple differential differ slightly in what annotation they produce.

This is because the order of *conj* and *add* operations is slightly changed.

Example 6.2. Consider analysing the following program using the *ASub* description domain:

$$\begin{array}{lll} g & \leftarrow & X = a, ① \; p(X,Y,Z) ② \\ p(X,Y,Z) & \leftarrow & ③ X = Y, ④ \; Y = Z ⑤ \end{array}$$

The annotations given by the goal-independent based and simple differential semantics are given below, together with the conjunction of the differential components.

pgm point	gi_{Asub}	$\langle D_{in}, D_\delta \rangle$	$conj_{Asub}(D_{in}, D_\delta)$
①	$[\{X\}, \emptyset]$	$\langle [\emptyset, \emptyset], [\{X\}, \emptyset] \rangle$	$[\{X\}, \emptyset]$
③	$[\{X\}, \emptyset]$	$\langle [\{X\}, \emptyset], [\emptyset, \emptyset] \rangle$	$[\{X\}, \emptyset]$
④	$[\{X, Y\}, \emptyset]$	$\langle [\{X\}, \emptyset], [\emptyset, \{XY\}] \rangle$	$[\{X\}, \emptyset]$
⑤	$[\{X, Y, Z\}, \emptyset]$	$\langle [\{X\}, \emptyset], [\emptyset, \{XY, XZ, YZ\}] \rangle$	$[\{X\}, \{YZ\}]$
②	$[\{X\}, \{YZ\}]$	$\langle [\emptyset, \emptyset], [\{X\}, \{YZ\}] \rangle$	$[\{X\}, \{YZ\}]$

Note that the answer to the call $\mathbf{Atom_{gi}}$ is $[\emptyset, \{XY, XZ, YZ\}]$, exactly what appears in the differential component at ⑤.

Because the only difference between the annotation resulting from the goal-independent based semantics and the simple differential semantics results from a different order of conjoining information, they are identical if the abstract operations give the same result regardless of order.

Definition 6.2. A constraint description domain $\mathcal{D}$ is *order independent* if the induced conjunction operator, $conj_{\mathcal{D}}$, and the induced constraint addition operator, $add_{\mathcal{D}}$, satisfy the following conditions for any descriptions D, D', D'' and constraints e:

(a) $conj_{\mathcal{D}}(true_{\mathcal{D}}, D) = D$ ($true_{\mathcal{D}}$ is a left-identity for $conj_{\mathcal{D}}$),

(b) $conj_{\mathcal{D}}(conj_{\mathcal{D}}(D, D'), D'') = conj_{\mathcal{D}}(D, conj_{\mathcal{D}}(D', D''))$ ($conj_{\mathcal{D}}$ is associative),

(c) $add_{\mathcal{D}}(e, conj_{\mathcal{D}}(D, D')) = conj_{\mathcal{D}}(add_{\mathcal{D}}(e, D), D')$

The *SS* abstract domain is an example of an abstract domain which is order independent.

Proof of the following theorem relies on commutatiivity of the induced conjunction operator.

Lemma 6.1. Every induced conjunction operator is commutative.

PROOF. Let $conj_{\mathcal{D}}$ be the induced conjunction operator. As *conj* is commutative, it follows that

$$conj_{\mathcal{D}}(D, D') \sqcap conj_{\mathcal{D}}(D', D)$$

approximates *conj*. As $conj_{\mathcal{D}}$ is the best (hence least) approximation of *conj*,

$$conj_{\mathcal{D}}(D, D') = conj_{\mathcal{D}}(D, D') \sqcap conj_{\mathcal{D}}(D', D) = conj_{\mathcal{D}}(D', D). \quad \Box$$

Theorem 6.2. Let $sdiff_{\mathcal{D}}$ and $gi_{\mathcal{D}}$ be induced from a description domain $\mathcal{D}$ which is order independent. For all programs P and goals G, $sdiff_{\mathcal{D}}(P,G) = gi_{\mathcal{D}}(P,G)$.

PROOF. The proof is by induction on the simple differential semantics and the annotation phase of the goal independent semantics. The proof is based on the following invariant: at each program point, if $\langle D_{in}, D_\delta\rangle$ is the annotation given by the simple differential semantics and D_{gi} is the annotation given by the goal independent semantics, then $D_{gi} = conj_{\mathcal{D}}(D_{in}, D_\delta)$.

The base case is the initial call to $gi_{\mathcal{D}}(P,G)$. This calls **AnnBody** with G which annotates the initial program point (at the start) of G with $true_{\mathcal{D}}$. Using the simple differential semantics the starting calling pattern is $true_{\delta\mathcal{D}} = \langle true_{\mathcal{D}}, true_{\mathcal{D}}\rangle$ and so the initial program point is annotated with $\langle true_{\mathcal{D}}, true_{\mathcal{D}}\rangle$. As $\mathcal{D}$ is order independent, $conj_{\mathcal{D}}(true_{\mathcal{D}}, true_{\mathcal{D}}) = true_{\mathcal{D}}$. Thus the invariant holds.

Now consider the annotation of literal L_i in the goal

$$L_1 \ldots, ⓩ, L_i, ⓘ, \ldots \qquad\qquad z = i-1$$

Let the annotation at ⓧ be D^x_{gi} with the goal independent semantics and $\langle D^x_{in}, D^x_\delta\rangle$ with the simple differential semantics. There are two cases to consider.

The first case is when L_i is a constraint. In this case,

$$\begin{array}{lll}
D^i_{gi} & = add_{\mathcal{D}}(L_i, D^{i-1}_{gi}) & \text{(from the definition of the goal-} \\
 & & \text{independent semantics)} \\
 & = add_{\mathcal{D}}(L_i, conj_{\mathcal{D}}(D^{i-1}_{in}, D^{i-1}_\delta)) & \text{(from the induction invariant)} \\
 & = add_{\mathcal{D}}(L_i, conj_{\mathcal{D}}(D^{i-1}_\delta, D^{i-1}_{in})) & \text{(from Lemma 6.1)} \\
 & = conj_{\mathcal{D}}(add_{\mathcal{D}}(L_i, D^{i-1}_\delta), D^{i-1}_{in}) & \text{(from order independence)} \\
 & = conj_{\mathcal{D}}(D^i_\delta, D^i_{in}) & \text{(from the definition of the simple} \\
 & & \text{differential semantics)} \\
 & = conj_{\mathcal{D}}(D^i_{in}, D^i_\delta) & \text{(from Lemma 6.1.)}
\end{array}$$

The second case is when L_i is an atom. In this case,

$$\begin{array}{lll}
D^i_{gi} & = conj_{\mathcal{D}}(D^{i-1}_{gi}, extend_{\mathcal{D}}(W, \mathbf{Atom_{gi}}(L_i))) & \text{(from the definition of the goal-} \\
 & & \text{independent semantics)} \\
 & = conj_{\mathcal{D}}(D^{i-1}_{gi}, extend_{\mathcal{D}}(W, D'_\delta)) & \text{(from Theorem 6.1 where } \langle D'_{in}, D'_\delta\rangle \\
 & & \text{is } \mathbf{Atom}(L_i, \langle D^{i-1}_{in}, D^{i-1}_\delta\rangle\ )\text{)} \\
 & = conj_{\mathcal{D}}(conj_{\mathcal{D}}(D^{i-1}_{in}, D^{i-1}_\delta), extend_{\mathcal{D}}(W, D'_\delta)) & \text{(from the induction invariant)} \\
 & = conj_{\mathcal{D}}(D^{i-1}_{in}, conj_{\mathcal{D}}(D^{i-1}_\delta, extend_{\mathcal{D}}(W, D'_\delta))) & \text{(from order independence)} \\
 & = conj_{\mathcal{D}}(D^i_{in}, D^i_\delta) & \text{(from the definition of the simple} \\
 & & \text{differential semantics.)}
\end{array}$$

In this case, we must also prove that the calls to **AnnRule** and **Rule** have the "same" calling pattern. **AnnRule** is called with the calling pattern

$$inrestrict_{\mathcal{D}}(vars(L_i), D^{i-1}_{gi}),$$

while **Rule** will be called with the calling pattern

$$\begin{array}{ll}
\quad inrestrict_{\delta\mathcal{D}}(vars(L_i), \langle D^{i-1}_{in}, D^{i-1}_\delta\rangle) & \\
= \langle inrestrict_{\mathcal{D}}(vars(L_i), conj_{\mathcal{D}}(D^{i-1}_{in}, D^{i-1}_\delta)), true_{\mathcal{D}}\rangle & \text{(from the definition of the} \\
 & \text{simple differential semantics)} \\
= \langle inrestrict_{\mathcal{D}}(vars(L_i), D^{i-1}_{gi}), true_{\mathcal{D}}\rangle & \text{(from the induction invariant.)}
\end{array}$$

Now from Lemma 6.1 and order independence we have that

$$inrestrict_{\mathcal{D}}(vars(L_i), D_{gi}^{i-1}) = conj_{\mathcal{D}}(inrestrict_{\mathcal{D}}(vars(L_i), D_{gi}^{i-1}), true_{\mathcal{D}}),$$

as required. □

Example 6.3. Consider again the domain SS and the program from Example 3.1:

$$\begin{array}{ll} g & \leftarrow X = cons(A, Z), Y = cons(B, Z), ① \; p(X, Y) ② \\ p(X, Y) & \leftarrow ③ \; X = Y ④ \end{array}$$

The goal-independent analysis phase finds the answer to the goal $p(X, Y)$ is $\{XY\}$.
At ① the goal-independent annotation gives $\{XA, YB, XY\}$, while the differential gives $\langle \emptyset, \{XA, YB, XY\}\rangle$. At ③ the descriptions are $\{XY\}$ and $\langle\{XY\}, \emptyset\rangle$ respectively. The descriptions at ④ are $\{XY, XX, YY\}$ and $\langle\{XY\}, \{XY\}\rangle$. The resulting descriptions at ② are given for the goal-independent annotation phase by conjoining the description at ① with the answer to $p(X, Y)$ as follows:

$$conj_{SS}(\{XA, YB, XY\}, \{XY\}) = \{XA, YB, XY, XX, YY, XB, YA, AB\}$$

On the other hand, for the simple differential approach, the result of the abstract function $comb_{\delta SS}(\{X, Y, A, B\}, \langle\emptyset, \{XA, YB, XY\}\rangle, \{X, Y\}, \langle\{XY\}, \{XY\}\rangle)$ is

$$\langle\emptyset, \{XA, YB, XY, XX, YY, XB, YA, AB\}\rangle.$$

Note that the resulting descriptions are more accurate than those obtained by the standard or specialised approaches since they do not contain AA and BB.

Because the concrete domain is order independent, we have the following corollary relating the concrete differential semantics to the collecting semantics.

Corollary 6.1. For all programs P and goals G $concdiff(P, G) = coll(P, G)$.

The differential semantics is clearly less efficient than the goal-independent semantics in general because it performs the analysis part for the differential components once for every new calling pattern that is generated rather than exactly once for each predicate. However, the differential approach has many advantages. From an implementation viewpoint, it performs the analysis and annotation in a single phase rather than the two phases required by the goal-independent based approaches. Since it is defined within the goal-dependent framework, an implementation for goal-dependent analysis can trivially be modified to provide a differential analysis. From a theoretical viewpoint, it is easier to relate the steps in a differential analysis to those in a standard goal-dependent analysis. Hence, as we shall shortly see, theorems relating the two approaches are easier to prove. Most importantly, because of its similar form to goal-dependent analysis it is easy to extend the differential semantics to create more accurate analysis. This will be examined in the next section.

Jacobs and Langen [11] give sufficient conditions, based on the characteristics of the abstract operations, which ensure that goal-independent analysis is as accurate as the standard top-down analysis. Since their semantics uses a different set of abstract operations, comparison is difficult. However, their conditions essentially correspond to the idempotence and associativity of $conj_{\mathcal{D}}$, and its distributivity over $lub_{\mathcal{D}}$, $add_{\mathcal{D}}$, $restrict_{\mathcal{D}}$ and $extend_{\mathcal{D}}$. Therefore, if these conditions are met by the induced abstract operations, the above theorem implies that the differential semantics is also more accurate than the standard semantics. These conditions are quite restrictive. The only useful domain we know of whose induced abstract operations satisfy the conditions is Pos [16].

Owing to the straightforward link between the goal-dependent and differential semantics, we can provide a different, less restrictive, set of conditions that ensure that the simple differential (and hence the goal-independent) approach is uniformly more accurate than the standard goal-dependent approach. We show that the simple differential semantics is more precise for "upwards closed" description domains where conjunction is associative. For example, although structure sharing SS does not satisfy the Jacobs and Langen conditions ($conj_{SS}$ is not distributive over lub_{SS}), the simple differential semantics induced for SS is uniformly more accurate than the standard semantics. Because the associativity condition guarantees that the simple differential semantics is identical to the goal-independent based semantics, the following theorem also extends the results of Jacobs and Langen.

Theorem 6.3. If $sdiff_{\mathcal{D}}$ and $std_{\mathcal{D}}$ are induced from a description domain $\mathcal{D}$ which is upwards closed and order independent, then $sdiff_{\mathcal{D}}(P,G) \leq std_{\mathcal{D}}(P,G)$ for all goals G and programs P.

PROOF. The proof is by structural induction, maintaining the following invariant: at each program point where the description given by the differential semantics is $\langle D_{in}, D_\delta \rangle$ and the description given by the standard semantics is D_{std}, we have that $conj_{\mathcal{D}}(D_{in}, D_\delta) \leq D_{std}$.

The proof is similar to that of Theorem 6.2 except for the point at which combination is performed. Consider the following program fragment:

$$\ldots,\ ①,\mathtt{p}(\tilde{v}),④,\ldots$$

$$\mathtt{p}(\tilde{v})\mathtt{:-}\ B_1.②$$

$$\mathtt{p}(\tilde{v})\mathtt{:-}\ B_2.③$$

We assume for simplicity that p is defined by only two rules, the extension to an arbitrary number is straightforward.

Let $\langle D^x_{in}, D^x_\delta \rangle$ and D^x_{std} be the descriptions given at program point ⓧ by the simple differential and standard goal-dependent semantics respectively. By induction we have that at program points $x \in \{1,2,3\}$, $conj_{\mathcal{D}}(D^x_{in}, D^x_\delta) \leq D^x_{std}$. Given $conj_{\mathcal{D}}(D^x_{in}, D^x_\delta) \leq D^x_{std}$, then since $\mathcal{D}$ is upwards closed we have that $D^x_{in} \leq conj_{\mathcal{D}}(D^x_{in}, D^x_\delta)$. Hence $D^x_{in} \leq D^x_{std}$. Similarly $D^x_\delta \leq D^x_{std}$.

By monotonicity of $outrestrict_{\mathcal{D}}$, $outrestrict_{\mathcal{D}}(\tilde{v}, D^x_\delta) \leq outrestrict_{\mathcal{D}}(\tilde{v}, D^x_{std})$, where $x \in \{2,3\}$. Let

$$\begin{aligned} lub_\delta &= lub_{\mathcal{D}}(\{outrestrict_{\mathcal{D}}(\tilde{v}, D^x_\delta) \mid x \in \{2,3\}\}) \\ lub_{std} &= lub_{\mathcal{D}}(\{outrestrict_{\mathcal{D}}(\tilde{v}, D^x_{std}) \mid x \in \{2,3\}\}). \end{aligned}$$

Then, by the monotonicity of $lub_{\mathcal{D}}$, $lub_\delta \leq lub_{std}$. Now $D^4_{std} = conj_{\mathcal{D}}(D^1_{std}, lub_{std})$, $D^4_{in} = D^1_{in}$, and $D^4_\delta = conj_{\mathcal{D}}(D^1_\delta, lub_\delta)$. Since $conj_{\mathcal{D}}$ is associative,

$$conj_{\mathcal{D}}(D^4_{in}, D^4_\delta) = conj_{\mathcal{D}}(conj_{\mathcal{D}}(D^1_{in}, D^1_\delta), lub_\delta).$$

Hence, by monotonicity of $conj_{\mathcal{D}}$, since $conj_{\mathcal{D}}(D^1_{in}, D^1_\delta) \le D^1_{std}$ and $lub_\delta \le lub_{std}$ we have that $conj_{\mathcal{D}}(D^4_{in}, D^4_\delta) \le conj_{\mathcal{D}}(D^1_{std}, lub_{std}) = D^4_{std}$. □

7. DIFFERENTIAL ANALYSIS

Simple differential analysis, introduced in the previous section, is a way of defining a goal-independent analysis and annotation in a way that is closely related to a goal-dependent analysis. In this section we show how we can make use of the extra calling pattern information carried around by the differential analysis to improve accuracy.

Example 7.1. Recall the goal and program from Example 6.2

$$\begin{array}{ll} g & \leftarrow X = a, ① \; p(X,Y,Z) ② \\ p(X,Y,Z) & \leftarrow ③ X = Y, ④ \; X = Z ⑤ \end{array}$$

The standard goal-dependent analysis for $Asub$ produces the following annotations:

$$\begin{array}{ll} ① & [\{X\}, \emptyset] \\ ③ & [\{X\}, \emptyset] \\ ④ & [\{X, Y\}, \emptyset] \\ ⑤ & [\{X, Y, Z\}, \emptyset] \\ ② & [\{X, Y, Z\}, \emptyset] \end{array}$$

Recall that, for the simple differential analysis, the corresponding annotation at ② is $\langle [\emptyset, \emptyset], [\{X\}, \{YZ\}] \rangle$. Clearly the simple differential approach loses significant accuracy by not taking calling pattern information regarding ground variables into account.

The above example motivates a less restrictive form of differential analysis where some calling pattern information is allowed to appear in the differential component. We will only require $conj_{\mathcal{D}}(D_{in}, D_\delta)$ to be a correct approximation of the constraints that can occur during program execution. This leads to a considerably more flexible approach to differential analysis.

Definition 7.1. A *differential semantics* is defined the same as the simple differential semantics except that

$$inrestrict_{\delta\mathcal{D}}(W, \langle D_{in}, D_\delta \rangle) = \langle D'_{in}, D'_\delta \rangle$$

where $conj_{\mathcal{D}}(D'_{in}, D'_\delta) \ge restrict_{\mathcal{D}}(W, conj_{\mathcal{D}}(D_{in}, D_\delta))$.

A *differential semantics*, $diff_{\mathcal{D}}$, for description domain $\mathcal{D}$ and a program P is obtained by replacing the $inrestrict_{\delta\mathcal{D}}$ by any operation which satisfies the above condition.

It is not difficult to show that any differential semantics is correct.

Theorem 7.1. Let $diff_{\mathcal{D}}$ be a differential semantics for description domain $\mathcal{D}$. Then, for all programs P and goals G, $diff_{\mathcal{D}}(P, G) \propto coll(P, G)$.

PROOF. Using standard abstract interpretation techniques, we can show that at each stage $conj_{\mathcal{D}}(D_{in}, D_\delta)$ describes the constraints collected in the operational semantics. □

Clearly the simple differential semantics is a special case of the differential semantics. But we can also show that the standard goal-dependent semantics is a special case.

Theorem 7.2. Let $\mathcal{D}$ be a description domain where $true_{\mathcal{D}}$ is a (left-)identity for $conj_{\mathcal{D}}$. Let $diff_{\mathcal{D}}$ be a differential semantics for description domain $\mathcal{D}$ where

$$inrestrict_{\delta\mathcal{D}}(W, \langle D_{in}, D_\delta\rangle) = \langle true_{\mathcal{D}}, inrestrict_{\mathcal{D}}(W, conj_{\mathcal{D}}(D_{in}, D_\delta))\rangle$$

Then, for all programs P and goals G, $diff_{\mathcal{D}}(P, G) = std_{\mathcal{D}}(P, G)$.

PROOF. It is straightforward to show by induction that if the annotation of a program point using the differential semantics is $\langle D_{in}, D_\delta\rangle$ and D_{std} is the annotation of the same point using the standard semantics then D_{in} is $true_{\mathcal{D}}$ and D_δ is D_{std}. Thus, as $true_{\mathcal{D}}$ is a (left-)identity for $conj_{\mathcal{D}}$, $conj_{\mathcal{D}}(D_{in}, D_\delta) = D_{std}$. □

Ideally, the operation $inrestrict_{\delta\mathcal{D}}$ should be defined so that calling pattern information that can improve accuracy is made part of the δ component, and no calling pattern information that can decrease accuracy by being combined with itself is part of the δ component. Conjunction of downwards closed information with itself not only cannot lose accuracy but may also gain accuracy. Therefore, this kind of information is a perfect candidate for inclusion in the differential component. On the other hand, upwards closed information should not be included since it might yield a loss of accuracy when combined with itself.

By separating information in the description into two components we can combine the advantage of goal-dependent analysis (calling pattern information) with the advantage of goal-independent analysis (not combining the same information twice).

Example 7.2. We can define a (non-simple) differential approach for $ASub$ by including the groundness information (which is downwards closed) in the differential component

$$\begin{aligned} inrestrict_{\delta ASub}(W, \langle D_{in}, D_\delta\rangle) &= \langle D'_{in}, D'_\delta\rangle \text{ where} \\ D'_{in} &= inrestrict_{ASub}(W, conj_{ASub}(D_{in}, D_\delta)) \\ D'_\delta &= [G_{in} \cup G_\delta, \emptyset] \end{aligned}$$

where $D_{in} = [G_{in}, S_{in}]$ and $D_\delta = [G_\delta, S_\delta]$.

Consider again the program from Example 6.2:

$$\begin{array}{lll} g & \leftarrow & X = a, ① \, p(X, Y, Z) ② \\ p(X, Y, Z) & \leftarrow & ③ X = Y, ④ \, Y = Z ⑤ \end{array}$$

Applying the differential semantics we obtain the description $\langle [\{X\}, \emptyset], [\{X\}, \emptyset]\rangle$ at ③. Hence at ⑤ we obtain the description $\langle [\{X\}, \emptyset], [\{X, Y, Z\}, \emptyset]\rangle$ and returning to ② we obtain the completely accurate $\langle [\emptyset, \emptyset], [\{X, Y, Z\}, \emptyset]\rangle$. Hence we do not lose any accuracy with respect to the standard goal-dependent semantics.

We can give conditions that ensure that a differential semantics is uniformly more accurate than the simple differential semantics. The definition of a differential semantics for *ASub* given in Example 7.2 satisfies the conditions of the following theorem.

Theorem 7.3. Let $\mathit{diff}_{\mathcal{D}}$ be a differential semantics for description domain D where for all W, D_{in} and D_δ, if

$$\mathit{inrestrict}_{\delta\mathcal{D}}(W, \langle D_{in}, D_\delta\rangle) = \langle D'_{in}, D'_\delta\rangle$$

then $D'_{in} \leq \mathit{inrestrict}_{\mathcal{D}}(W, \mathit{conj}_{\mathcal{D}}(D_{in}, D_\delta))$ and $D'_\delta \leq \mathit{true}_{\mathcal{D}}$. Then for all programs P and goals G we have that $\mathit{diff}(P,G) \leq \mathit{sdiff}(P,G)$.

PROOF. By induction we show that at each program point $D^{\mathit{diff}}_{in} \leq D^{\mathit{sdiff}}_{in}$ and $D^{\mathit{diff}}_{\delta} \leq D^{\mathit{sdiff}}_{\delta}$. The result follows from the definitions of *inrestrict* and the monotonicity of the other operations. □

A consequence of this theorem allows us to improve the simple differential for all abstract domains D in which $\perp$ represents unreachability, that is $\gamma(\perp) = \emptyset$. In such domains if the calling pattern is $\perp$ this means that the program point is unreachable. We can take advantage of this by defining $\mathit{inrestrict}_{\delta\mathcal{D}}$ as follows:

$$\mathit{inrestrict}_{\delta\mathcal{D}}(W, \langle D_{in}, D_\delta\rangle) = \langle D'_{in}, D'_\delta\rangle$$

where $D'_{in} = \mathit{restrict}_{\mathcal{D}}(W, \mathit{conj}_{\mathcal{D}}(D_{in}, D_\delta))$ and $D'_\delta = \mathit{true}_{\mathcal{D}}$, unless $D'_{in} = \perp$, in which case $D'_\delta = \perp$. Clearly this satisfies the conditions of Theorem 7.3. In fact, the definition simply propagates a $\perp$ in the first component to the second component. In practice, a similar operation is performed in any decent implementation of an abstract domain which is defined in terms of several components: whenever, as a result of an abstract operation, we determine that at least one component is $\perp$ (representing unreachability), the rest of the components can also be set to $\perp$.

Example 7.3. Consider the following program

$$\begin{array}{l} g(X,Y) \leftarrow {①} X = 3, {②} p(X,Y) {③} \\ p(X,Y) \leftarrow {④} X < 0, {⑤} Y = Z + 1, {⑥} p(Z,X) {⑦} \\ p(X,Y) \leftarrow {⑧} X > 0 {⑨} \end{array}$$

For a differential analysis in the concrete domain with calling pattern $\langle\{X = 3\}, \{\mathit{true}\}\rangle$ at ④, the description at ⑤ is $\langle\{X = 3\}, \{X < 0\}\rangle$. This represents the constraint $X = 3 \wedge X < 0$ which is unsatisfiable, hence the point is not reachable. Differential analysis, as defined, does not yet notice the unreachability. It is at ⑥, when determining the calling pattern for $p(Z,X)$ that we conjoin the two components and detect unreachability. Therefore, the differential analysis can ignore the remainder of the rule, eventually calculating the description at ③ as $\langle\{\mathit{true}\}, \{X = 3 \wedge X > 0\}\rangle$.

The goal-independent analysis phase cannot use calling pattern information. It calculates the fixpoint for the goal $p(X,Y)$ as $\{X > 0, X < 0 \wedge Y > 1\}$ and then obtains the answer $\{X = 3 \wedge X > 0\}$ at ③. Clearly this is less efficient and can lead to inaccuracies.

When using an abstract domain $\mathcal{D}$ where $\perp$ represents an unreachable program point, the differential semantics can be made even more accurate by replacing a description $\langle D_{in}, D_\delta \rangle$ by $\langle \perp, \perp \rangle$ whenever $conj_{\mathcal{D}}(D_{in}, D_\delta) = \perp$. Thus, unreachability is detected as soon as possible. However, this would require additional abstract conjunction, thus affecting efficiency.

We might imagine that, as defined, a differential approach can always be designed that is as least as accurate as both the specialised goal-dependent and goal-independent based approaches. Unfortunately, this is not the case because specialised semantics can take advantage of cases in which the return description invalidates information in the call.

Example 7.4. Consider the program from Example 3.4 together with goal g:

$$\begin{array}{ll} g & \leftarrow X = Y, ① \; r(X,Y) ② \\ r(X,Y) & \leftarrow ③ X = a ④ \\ r(X,Y) & \leftarrow ⑤ Y = a ⑥ \end{array}$$

For the $ASub$ abstract domain and using the (non-simple) differential approach, the description is $\langle [\emptyset, \{XY\}], [\{X\}, \emptyset] \rangle$ at ④. Similarly, at ⑥ the description is $\langle [\emptyset, \{XY\}], [\{Y\}, \emptyset] \rangle$. The lub of these two descriptions is $\langle [\emptyset, \{XY\}], [\emptyset, \emptyset] \rangle$. Hence, at ② we get $\langle [\emptyset, \emptyset], [\emptyset, \{XY\}] \rangle$. As illustrated in Example 3.4, using the specialised goal-dependent approach, the pair XY is removed.

This example shows that, owing to the specialised combination, the specialised goal-dependent approach is sometimes more accurate than the differential approach. So to achieve better accuracy we are interested in combining both approaches. One might expect that we could combine them directly, just by using the specialised combination rather than conjunction inside the differential semantics as follows:

$$comb^{spec}_{\delta\mathcal{D}}(W, \langle D_{in}, D_\delta \rangle, W', \langle D'_{in}, D'_\delta \rangle) = \langle D_{in}, comb^{spec}_{\mathcal{D}}(W, D_\delta, W', D'_\delta) \rangle.$$

But this is incorrect because the differential parts of the descriptions do not satisfy the condition for which the specialised abstract combine operation, $comb^{spec}_{\mathcal{D}}$, was defined.

Example 7.5. Consider analysing the following program and goal g using the abstract domain SS.

$$\begin{array}{ll} g & \leftarrow A = X, Y = U, V = B, ① \; p(X,Y,U,V) ② \\ p(X,Y,U,V) & \leftarrow ③ X = Y, U = V ④ \end{array}$$

At program point ① the inferred description is $\langle \emptyset, \{AX, YU, VB\} \rangle$, at ③ the description is $\langle \{YU\}, \emptyset \rangle$, at ④ the description is $\langle \{YU\}, \{XY, UV\} \rangle$. Using

$$comb^{spec}_{SS}(\{A,B,X,Y,U,V\}, \{AX, YU, VB\}, \{X,Y,U,V\}, \{XY, UV\})$$

we obtain at ② a description which misses the pair AB because it does not consider the path AX, XY, YU, UV, VB of length five. This is because the information given by the pair YU has not been propagated to $\{XY, UV\}$, a condition which is needed for the correctness of the specialised combine operation.

To fuse the two methods we need to include a new component D_{tot} in the descriptions, which holds a total description of the constraints. Hence descriptions are now of the form $\langle D_{in}, D_\delta, D_{tot}\rangle$. Operations to compute the first two components are the usual differential operations while operations to compute D_{tot} are just the usual (specialised semantics) abstract operations for $\mathcal{D}$, except that *comb* uses both a simple differential and a specialised approach.

Definition 7.2. Let $\mathcal{D}$ be a description domain. The *fused abstract domain* $\phi\mathcal{D}$ is $\mathcal{D} \times \mathcal{D} \times \mathcal{D}$. Abstract operations on $\phi\mathcal{D}$ are given by:

$$\begin{array}{l}
true_{\phi\mathcal{D}} = \langle true_{\mathcal{D}}, true_{\mathcal{D}}, true_{\mathcal{D}}\rangle \\
add_{\phi\mathcal{D}}(p, \langle D_{in}, D_\delta, D_{tot}\rangle) = \langle D_{in}, add_{\mathcal{D}}(p, D_\delta), add_{\mathcal{D}}(p, D_{tot})\rangle \\
lub_{\phi\mathcal{D}}(\mathbf{Y}) = \langle lub_{\mathcal{D}}(\mathbf{Y}_{in}), lub_{\mathcal{D}}(\mathbf{Y}_\delta), lub_{\mathcal{D}}(\mathbf{Y}_{tot})\rangle \\
\qquad\qquad\qquad \text{where } \mathbf{Y}_{in} = \{D_{in} \mid \langle D_{in}, D_\delta, D_{tot}\rangle \in \mathbf{Y}\} \\
\qquad\qquad\qquad \text{and } \mathbf{Y}_\delta = \{D_\delta \mid \langle D_{in}, D_\delta, D_{tot}\rangle \in \mathbf{Y}\} \\
\qquad\qquad\qquad \text{and } \mathbf{Y}_{tot} = \{D_{tot} \mid \langle D_{in}, D_\delta, D_{tot}\rangle \in \mathbf{Y}\} \\
outrestrict_{\phi\mathcal{D}}(W, \langle D_{in}, D_\delta, D_{tot}\rangle) = \\
\qquad \langle D_{in}, outrestrict_{\mathcal{D}}(W, D_\delta), outrestrict_{\mathcal{D}}(W, D_{tot})\rangle \\
inrestrict_{\phi\mathcal{D}}(W, \langle D_{in}, D_\delta, D_{tot}\rangle) = \\
\qquad \langle inrestrict_{\mathcal{D}}(W, D_{tot}), true_{\mathcal{D}}, inrestrict_{\mathcal{D}}(W, D_{tot})\rangle \\
extend_{\phi\mathcal{D}}(W, \langle D_{in}, D_\delta, D_{tot}\rangle) = \\
\qquad \langle extend_{\mathcal{D}}(W, D_{in}), extend_{\mathcal{D}}(W, D_{diff}), extend_{\mathcal{D}}(W, D_{tot})\rangle \\
comb_{\phi\mathcal{D}}(W, \langle D_{in}, D_\delta, D_{tot}\rangle, W', \langle D'_{in}, D'_\delta, D'_{tot}\rangle) = \langle D_{in}, D''_\delta, D''_{tot}\rangle \\
\qquad\qquad\qquad \text{where } D''_\delta = conj_{\mathcal{D}}(D_\delta, extend_{\mathcal{D}}(W, D'_\delta)) \\
\text{and } D''_{tot} = conj_{\mathcal{D}}(D_{tot}, extend_{\mathcal{D}}(W, D'_\delta)) \sqcap comb^{spec}_{\mathcal{D}}(W, D_{tot}, W', D'_{tot})
\end{array}$$

The *simple fused semantics*, $sfuse_{\mathcal{D}}$, for description domain $\mathcal{D}$ and a program P is the standard goal-dependent semantics using $\phi\mathcal{D}$ and the above abstract operations.

The simple fused semantics *induced* for $\mathcal{D}$ is the semantics obtained by inducing the auxiliary functions from $comb^{spec}$, $conj$, add, lub, $restrict$, $restrict$ and $extend$ respectively.

In effect, this definition executes both the simple differential and the specialised approaches, yielding their greatest lower bound. It is therefore guaranteed to be as accurate as both of the approaches. It may be considered as an example of a product domain [5]. Again, in an abuse of notation, when comparing a fused semantics with some other semantics we will consider the annotations of the fused semantics to be only the last component of the tuple.

Because the differential component in the simple fuse semantics is equivalent to the associated information obtained by a goal-independent based analysis, the *sfuse* semantics and the *gigd* are equivalent.

Theorem 7.4. If $gigd_{\mathcal{D}}$ and $sfuse_{\mathcal{D}}$ are induced from a description domain $\mathcal{D}$, then $sfuse_{\mathcal{D}}(P, G) = gigd_{\mathcal{D}}(P, G)$ for all goals G and programs P.

At last, we are now in a position to define our last semantics for analysis. This is the fused semantics, *fuse*, which fuses any differential semantics with any goal-dependent semantics. In particular by using a more accurate differential semantics together with the specialised semantics we can create a fused semantics which is uniformly more accurate than the *gigd* semantics.

The *gigd* and *fuse* semantics are highly related. They differ only in the method in which they attempt to reduce loss of accuracy that results from combining information with itself. The *fuse* approach inherits the advantages and disadvantages compared to the *gigd* approach that the differential approach has compared to the goal-independent based approach. Using *fuse* can be more accurate than using *gigd*, but possibly slower since in effect differential computations may repeat steps performed only once in goal-independent analysis.

8. EXPERIMENTAL EVALUATION

In this section we experimentally compare the efficiency and accuracy of the different analysis methods discussed in the previous sections, for two different abstract domains: SS and $ASub$. The following versions of the abstract domains have been integrated in the general goal-dependent top-down semantics: the standard goal-dependent semantics (referred to as *std* in the tables), specialised goal-dependent semantics (*spec*), simple differential semantics (*sdiff*). We have also implemented the (non-simple) differential version (*diff*) for $ASub$ described in Example 7.2, that is, the groundness information is also included in the differential component. Finally, for each abstract domain we have implemented a fused semantics, *sfuse* for SS and *fuse* (based on *diff*) for $Asub$.

All these analysers have been integrated in the PLAI framework [19], in which a specialised version for $ASub$ was already available. The reliability of the comparison is based on the fact that, for each abstract domain, the different versions have been implemented in such a way that they reuse the abstract functions from the standard version.

Also, for both SS and $ASub$, we have implemented the following goal-independent based semantics:

- *gi*: implements a goal-dependent phase in which the answer to a particular call $Call$ is computed by abstractly conjoining $Call$ and the answer provided by the goal-independent phase.
- *cod*: implements a goal-dependent phase based on the approach of Codish *et al.* [3], that is, goal-independent information is only used for computing the answers to recursive predicates, the rest being computed using the specialised goal-dependent semantics.
- *gigd*: implements the *gigd* semantics in which both goal-dependent and goal-independent information are used to find the answers to each (recursive or non-recursive) call.

These semantics and the goal-independent phase have been implemented by suitably modifying PLAI.

For the evaluation we have selected a wide set of benchmarks which have been traditionally used in the evaluation of analysers[3]. These benchmarks have been

[3] A complete description of these benchmarks can be found, for example, in [2].

Program	*spec*	*std*	*sdiff*	*sfuse*	*gi*	*cod*	*gigd*	ratio
aiakl	0.29	1.08	1.64	1.77	1.57	1.51	2.32	0.84
ann	10.14	2.05	1.92	2.64	1.04	0.83	2.48	0.32
bid	1.14	1.42	1.38	1.66	1.13	1.14	1.74	0.33
boyer	6.69	1.68	2.20	2.57	0.85	0.66	2.04	0.15
browse	0.30	1.29	1.29	1.45	1.15	1.18	1.93	0.56
deriv	0.38	1.21	1.89	2.20	1.49	1.49	2.69	0.97
fib	0.02	0.81	1.02	1.19	1.27	1.40	1.34	0.67
grammar	0.11	0.94	1.28	1.44	1.59	1.59	1.63	0.59
hanoiapp	0.52	1.97	0.73	0.99	0.37	0.43	0.91	0.17
mmatrix	0.08	0.94	1.39	1.33	1.33	1.35	1.76	0.63
occur	0.33	1.30	2.30	2.73	1.32	1.17	2.04	0.40
peephole	4.91	1.34	1.59	1.83	1.02	0.97	2.13	0.54
progeom	0.86	1.08	0.99	1.22	0.77	0.98	1.18	0.25
qplan	9.40	2.30	3.05	3.47	1.33	1.11	2.24	0.34
qsortapp	0.32	1.06	1.44	1.61	0.88	0.91	1.62	0.35
query	0.18	1.03	1.25	1.33	1.52	1.65	1.69	0.64
rdtok	1.63	0.97	1.98	2.18	1.09	1.16	1.75	0.58
read	34.33	2.19	1.28	1.65	0.40	0.44	1.46	0.05
serialize	0.37	1.12	1.78	1.84	1.39	1.36	2.43	0.88
tak	4.10	1.45	1.79	0.31	0.08	0.13	0.16	0.02
warplan	9.80	1.78	1.10	1.41	0.67	0.81	1.45	0.08
zebra	0.26	1.01	1.29	1.53	1.33	1.43	1.62	0.52
		1.36	1.57	1.74	1.07	1.08	1.76	0.45

TABLE 8.1. Efficiency results for SS

normalised for our experiments. The efficiency results for SS and $ASub$ are shown in Tables 8.1 and 8.2.

For each benchmark, the information shown is the following: the first column (*spec*) shows the analysis times in milliseconds[4] for the analyser using the specialised version; the remaining columns except the last show the ratios of analysis time for various techniques compared to that of *spec*; the last column shows the ratio of the time taken by the goal-independent phase versus that of *spec*, which affects the goal-independent based analysers. The last row indicates the arithmetic mean of the ratios.

The results from the efficiency evaluation show that the time penalty for the more sophisticated methods, compared with the traditional approach, rarely exceeds 50%, and sometimes they are actually faster. In particular, the specialised goal-dependent method is almost uniformly faster than the traditional approach. Of the equally precise *sdiff* and *gi*, the latter is typically faster.

The accuracy results for SS and $ASub$ are given in Tables 8.3, 8.4, and 8.5. For the abstract domain SS, Table 8.3 shows, for each benchmark and analyser, the number of program variables which are known to be linear (Linearity) and the total number of pairs or edges between different variables inferred by each analyser

[4] SICStus 2.1, compactcode, SPARCstation 10, one processor.

Program	*spec*	*std*	*sdiff*	*diff*	*fuse*	*gi*	*cod*	*gigd*	ratio
aiakl	0.24	1.11	1.82	1.64	1.76	1.75	1.69	2.51	1.06
ann	4.34	1.59	1.93	1.61	2.00	1.47	1.40	2.29	0.69
bid	0.63	1.00	1.52	1.21	1.33	1.42	1.36	1.57	0.49
boyer	4.08	1.33	0.70	0.67	0.76	0.51	0.63	0.82	0.23
browse	0.23	1.09	1.60	1.08	1.17	1.48	1.53	1.78	0.79
deriv	0.19	1.10	2.64	1.41	1.54	2.44	2.48	3.23	1.90
fib	0.03	0.80	0.92	0.93	1.00	1.26	1.27	1.60	0.60
grammar	0.10	0.95	1.34	1.24	1.34	1.48	1.55	1.64	0.58
hanoiapp	0.13	0.99	2.27	1.24	1.32	1.35	1.37	1.89	0.72
mmatrix	0.07	1.05	1.49	1.25	1.37	1.42	1.49	1.88	0.82
occur	0.06	1.05	3.26	1.26	1.36	2.91	3.04	3.33	2.18
peephole	3.48	1.24	1.91	1.52	1.69	1.28	1.25	2.21	0.76
progeom	0.50	1.63	1.72	1.72	1.50	1.31	1.13	1.62	0.42
qplan	9.15	2.15	2.94	2.60	2.97	1.26	1.09	2.09	0.33
qsortapp	0.28	1.05	1.53	1.38	1.57	0.99	0.96	1.63	0.37
query	0.18	0.98	1.21	1.27	1.33	1.54	1.66	1.66	0.63
rdtok	1.68	1.01	1.75	1.20	1.31	0.97	1.06	1.75	0.56
read	9.96	1.03	1.34	1.31	1.43	0.43	0.44	1.34	0.15
serialize	0.35	1.07	1.67	1.59	1.69	1.32	1.29	2.20	0.84
tak	0.03	0.93	1.95	0.93	1.11	1.82	2.36	1.70	0.65
warplan	6.77	1.62	1.35	1.44	1.78	0.76	0.83	1.56	0.11
zebra	0.23	0.99	1.36	1.37	1.48	1.42	1.43	1.64	0.53
		1.17	1.74	1.36	1.49	1.39	1.42	1.91	0.70

TABLE 8.2. Efficiency results for $ASub$

(Pairs). Obviously, the more linear variables and the fewer pairs, the more exact the information. Similar information is shown for the abstract domain $ASub$ in Tables 8.4 and 8.5. In addition, the number of program variables which are known to be ground is also shown for $ASub$ in Table 8.4. For the benchmarks chosen, the results obtained by *gigd* for both SS and $ASub$ are uniformly better than the rest. We will take advantage of this fact for improving readability: the results for the rest of analysers will only appear if they are different from those obtained by *gigd*. Also, columns labelled with more than one analyser mean that the results for those analysers are identical and they are thus shown in just one column.

As expected, the experiments confirm that *spec* is uniformly more accurate than *std* for both SS and $ASub$. While in SS accuracy is improved for **ann, rdtok**, and **read**, in $ASub$ the results of *std* and *spec* are identical except for **progeom**. The improvements observed in the mentioned benchmarks for SS do not carry to $ASub$ because the variables involved are already ground and have thus disappeared from the abstraction. In the case of **progeom**, its information is also improved by *spec* for SS for some particular calling patterns, but the collapsing of the information masks this fact in the final results. The most common improvement observed is the accurate inference of linear information. This is mainly due to the combination of a calling pattern including the sharing XY between linear variables X and Y, with an answer which also includes the pair XY but still has X and Y as linear

	Linearity					Pairs				
Program	*std*	*spec*	*sdiff* *gi*	*sfuse* *gigd*	*cod*	*std*	*spec*	*sdiff* *gi*	*sfuse* *gigd*	*cod*
aiakl				455					314	
ann				1625		5376		5376	5374	5376
bid	374	374		376	374	625	625		616	625
boyer				557					2240	
browse	186	186		188	186				193	
deriv				252					285	
fib				54					0	
grammar				202					57	
hanoiapp	55	55		92	85	139	139		102	110
mmatrix				154					46	
occur				78					104	
peephole				1948					1946	
progeom	308	308		438	308	410	410		352	410
qplan				2372					5543	
qsortapp				188					133	
query				372					28	
rdtok	1424			1429					417	
read	1483	1485		1508	1485	3820	3814		3728	3814
serialize				433					353	
tak	40	40	49	96	40	139	139		91	139
warplan	439	439		448	439	1538	1538		1511	1538
zebra				882					465	

TABLE 8.3. Accuracy results for SS

variables. This happens in `progeom` and `read`, and from this, *std* loses further accuracy. In `ann`, the loss of accuracy is due to the combination of a calling pattern which contains a pair XY and an answer which contains XY and YZ but not XZ. While *std* infers the pair XZ, *spec* is able to eliminate such a spurious pair.

The results of comparing *spec* and the various differential versions are more involved and significantly depend on the abstract domain. With $ASub$, *spec* is usually more accurate thanks to groundness propagation. This happens in `ann`, `bid`, `browse`, `grammar`, `hanoiapp`, `occur`, `rdtok`, `serialize`, `tak`, and `warplan`. In all these benchmarks, accuracy is recovered when groundness information is included in the differential component, as it can be observed by the results of *diff*. However, there is is also a case in which *sdiff* yields considerably stronger sharing information than *spec*, namely `boyer`. In this case, *sdiff* takes advantage of the change in the order of the analysis in a piece of code which performs backwards groundness propagation. The cases of `qplan` and `progeom` are more involved. In `qplan` groundness propagation improves the accuracy of *std* and *spec* with respect to that of *sdiff*. This is noticeable by the results provided by *diff*. However, *sdiff* also betters the accuracy of *spec* in some places. As a result, the number of pairs inferred by *fuse* is better than those of *spec*, *sdiff* and *diff*. The case of `progeom` is similar: both *spec* and *sdiff* achieve improvements, thus the results of *fuse* are

	Groundness			Linearity					
Program	*sdiff* *gi*	*fuse* *spec* *std* *diff* *gigd*	*cod*	*std*	*spec*	*sdiff* *gi*	*diff*	*fuse* *gigd*	*cod*
aiakl		81						455	
ann	1002	1089	1002					2110	2071
bid	263	270	263			541		802	
boyer		84		603	603			2010	1857
browse	71	154	71			195		212	193
deriv		173						280	
fib		30						54	
grammar	36	37						202	
hanoiapp		56				94		128	94
mmatrix		96						154	
occur	100	114	100			156		158	156
peephole		1290						2083	
progeom		104		326	475	447	447	484	475
qplan	887	978						2619	
qsortapp		17						188	
query		211						372	
rdtok	516	582	516			1431		1441	1431
read		615						3044	
serialize	77	80	77					436	
tak	44	65	44					96	
warplan	176	214	176			539		548	539
zebra		181						882	

TABLE 8.4. Accuracy results for $ASub$

the best. We could conclude that, on average, *sdiff* and *gi* are the least accurate, owing to the importance of groundness information, and that *diff* and *spec* are quite similar, the former being more accurate only in `boyer` and `qplan`, the latter being more accurate only in `progeom`. As expected, *fuse* is uniformly more accurate than the rest.

In SS the importance of groundness information disappears and the number of times in which information is added twice significantly increases (note that in SS, rules which ground variables do not eliminate the input's sharing, and thus it is combined again when returning). As a result, the characteristics of *sdiff* are more valuable, and *sdiff* is at least as accurate as *spec* in all cases except `ann`, in which the specialised combination succeeds in eliminating a pair (which then appears in the next program point). In fact, the results obtained by *sdiff* are almost identical to those of *sfuse*, which is uniformly more accurate than all the other analysers. Also note that there is again one case (`tak`) in which there is a synergy between the

	Pairs					
Program	*std*	*spec*	*sdiff* *gi*	*diff*	*fuse* *gigd*	*cod*
aiakl					230	
ann			3411		3396	3620
bid			317		189	
boyer	2092	2092			1219	1398
browse			160		64	160
deriv					49	
fib					0	
grammar			41		40	
hanoiapp			28		23	28
mmatrix					11	
occur			7		4	7
peephole					775	
progeom	344	285	296	296	284	285
qplan	4710	4710	4929		4709	4710
qsortapp					127	
query					6	
rdtok			259		257	259
read					1000	
serialize			262		261	262
tak			2		0	2
warplan			1347		1333	1347
zebra					338	

TABLE 8.5. Accuracy results for $ASub$

improvements allowed by *spec* and *sdiff* which makes *sfuse* better than both *spec* and *sdiff*.

When the goal-independent based analyses are taken into account, we observe that while the results of *gigd* are always identical to those of *fuse* or *sfuse*, the results of *gi* are identical to those of *sdiff*. This confirms the theoretical results obtained in the previous sections. It also confirms that, although theoretically it is possible for *sdiff* and *fuse* to obtain better results than *gi* and *gigd*, respectively, thanks to the propagation of unreachability ($\perp$), such improvement is uncommon in practice for the domains under study. However, in domains which yield a higher percentage of $\perp$, such as those including freeness, types, and so on, such $\perp$-propagation is likely to significantly improve the results of *diff* and *fuse*. Regarding the results obtained by *cod* for $ASub$, we can conclude that, while the availability of the specialised combination allows it to improve its results with respect to *sdiff* in some cases (`bid`, `grammar`, `progeom`, and `qplan`), adding information twice makes it lose accuracy in some other cases (`ann`, `boyer`, and `browse`). The latter problem becomes more important in SS, owing to the lack of groundness propagation.

We conclude that the relative merits of the various approaches depend on the abstract domain. For the domains studied here, and weighing both efficiency and accuracy, *gi* appears to be very useful for SS, while *diff* gives good results for $ASub$. For cases where accuracy is the main concern, notice that *fuse* never appears to be excessively expensive.

9. OTHER APPLICATIONS

The example analyses considered in the previous sections may not do full justice to a differential approach. When we consider analysis of more sophisticated languages, or complex applications of the analysis information, the characteristics of a differential approach can become invaluable. We briefly present two examples.

In the analysis of programs with dynamic scheduling [10], the behaviour of the delayed atoms is approximated by a function from descriptions to descriptions. This function provides the connection between the variables currently being analysed and those over which delayed atoms are defined. Whenever a new constraint is analysed, the function must be evaluated upon the resulting constraint description, so that the effect of awakening delayed atoms is correctly included in the description. In a typical evaluation of a function, the delayed atoms only affect a small subset of the variables involved. This information is transmitted to the current set of variables through a long chain of operations which include combination. Therefore, a significant part of the information which has not changed will be combined with itself, possibly many times, throughout the chain. Since a differential approach need only combine new information, it can obtain more accurate results than those obtained using other approaches to combination. Note that it is only the evaluation of the function that needs to be performed in a differential manner; the analysis of literals which do not delay can follow the standard approach, if desired. Since the calling pattern information is required to determine when delayed atoms wake, we cannot use a goal-independent approach to obtain a comparable method.

The second example considers a particular application of the information inferred by abstract interpretation: automatic parallelisation based on *a posteriori* conditions. In this model, two goals g_1 and g_2 can be executed in parallel, for constraint store c if every partial answer of $\langle c, g_1\rangle$ is consistent with those of $\langle c, g_2\rangle$ [9]. At the abstract level, let D^1 and D^2 describe the partial answers of $\langle c, g_1\rangle$ and $\langle c, g_2\rangle$, respectively. Then, the condition stated above is satisfied if the abstract conjunction of D^1 and D^2 is definitely satisfiable. Unfortunately, in both the standard and specialised semantics, D^1 and D^2 describe constraints which are equal or stronger than c. Hence, in most cases their abstract conjunction will not be definitely satisfiable.

Using a differential approach, the description of the partial answers is $\langle D_{in}, D^1_\delta\rangle$ and $\langle D_{in}, D^2_\delta\rangle$, where D_{in} describes the constraint c. Then, the *a posteriori* condition holds if the abstract conjunction of D_{in}, D^1_δ and D^2_δ is definitely satisfiable. Note that the information about c is included only once.

Example 9.1. Consider the following constraint logic program:

$$\begin{array}{ll} g(X,Y,Z) & \leftarrow ① X = 3, ② \, p(X), r(X) \\ p(X) & \leftarrow true \\ r(X) & \leftarrow true \end{array}$$

and the *LSign* domain introduced by Marriott and Stuckey [17]. This domain is used to describe conjunctions of linear arithmetic constraints and gives information about possible failure. In *LSign* a linear arithmetic constraint is abstracted by replacing coefficients in the constraint by their sign. Signs are 0, $\oplus$, $\ominus$ and $\top$ which, respectively, describe 0, a positive coefficient, a negative coefficient and any coefficient. For example the conjunction $X + Z = 2 \wedge Y - Z = 3$ is described by $\{\oplus X + \oplus Z = \oplus, \oplus Y + \ominus Z = \oplus\}$.

Although $p(X)$ and $r(X)$ clearly satisfy the *a posteriori* condition, as $D^1 = D^2 = \{\oplus X = \oplus\}$ we cannot ensure that such condition is satisfied, since their abstract conjunction $\{\oplus X = \oplus, \oplus X = \oplus\}$ describes unsatisfiable constraints such as $X = 3 \wedge X = 2$. On the other hand, using a differential approach $D_{in} = \{\oplus X = \oplus\}$, $D^1_\delta = D^2_\delta = \emptyset$, and the condition holds.

10. CONCLUDING REMARKS

The present paper has investigated a variety of approaches to abstract interpretation of (constraint) logic programs, including differential methods. On the theoretical side we have arrived at a better understanding of goal-dependent and goal-independent based program analysis and their relative precision. We have introduced two approaches which theoretically are more accurate than either: the goal-independent goal-dependent (*gigd*) framework and the fused framework.

Implementors should also consider the issues raised in this paper carefully. Our empirical evaluation has shown that for very little additional implementation effort, a significant improvement in accuracy with small efficiency overhead can be achieved by using the differential approach.

REFERENCES

1. Bruynooghe, M., A Practical Framework for the Abstract Interpretation of Logic Programs, *Journal of Logic Programming* **10** (2): 91–124, 1991.

2. Bueno, F., García de la Banda, M., and Hermenegildo, M., Effectiveness of Global Analysis in Strict Independence-Based Automatic Parallelization, in: M. Bruynooghe (ed.), *Logic Programming: Proc. 1994 Int. Symp.*, pp. 320–336, MIT Press, 1994.

3. Codish, M., García de la Banda, M., Bruynooghe, M., and Hermenegildo, M., Goal Dependent versus Goal Independent Analysis of Logic Programs, in: F. Pfenning (ed.), *Proc. Fifth Int. Conf. Logic Programming and Automated Reasoning* (Lecture Notes in Artificial Intelligence 822), pp. 305–320, Springer-Verlag, 1994.

4. Cousot, P. and Cousot, R., Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints, in: *Proc. Fourth Ann. ACM Symp. Principles of Programming Languages*, pp. 238–252, ACM Press, 1977.

5. Cousot, P. and Cousot, R,. Systematic Design of Program Analysis Frameworks, in: *Proc. Sixth Ann. ACM Symp. Principles of Programming Languages*, pp. 269–282, ACM Press, 1979.

6. Dumortier, V., *Freeness and Related Analyses of Constraint Logic Programs Using Abstract Interpretation*, PhD Thesis, Catholic University of Leuven, Belgium, 1994.

7. Dumortier, V., Janssens, G., Bruynooghe, M., and Codish, M., Freeness Analysis in the Presence of Numerical Constraints, in: D. S. Warren (ed.), *Logic Programming: Proc. Tenth Int. Conf.*, pp. 100–115, MIT Press, 1993.

8. Falaschi, M., Levi, G., Martelli, M., and Palamidessi, C., A new declarative semantics for logic languages. in: R. Kowalski and K. Bowen (eds.), *Logic Programming: Proc. Fifth Int. Conf. Symp.*, pp. 993–1005, MIT Press, 1988.

9. García de la Banda, M., Hermenegildo, M., and Marriott, K., Independence in Constraint Logic Programs, in: D. Miller (ed.), *Logic Programming: Proc. 1993 Int. Symp.*, pp. 130–146, MIT Press, 1993.

10. García de la Banda, M., Marriott, K., and Stuckey, P., Efficient Analysis of Logic Programs with Dynamic Scheduling, in: J. Lloyd (ed.), *Logic Programming: Proc. 1995 Int. Symp.*, pp. 417–431, MIT Press, 1995.

11. Jacobs, D. and Langen, A., Static Analysis of Logic Programs for Independent And-parallelism, *Journal of Logic Programming* **13** (2&3): 291–31, 1992.

12. Jaffar, J. and Maher, M., Constraint Logic Programming: A Survey, *Journal of Logic Programming* **19/20**: 503–581, 1994.

13. Jones, N. and Søndergaard, H., A Semantics-Based Framework for the Abstract Interpretation of Prolog, in: S. Abramsky and C. Hankin (eds.), *Abstract Interpretation of Declarative Languages*, pp. 123–142, Ellis Horwood, 1987.

14. Kelly, A., Macdonald, A., Marriott, K., Søndergaard, H., Stuckey, P., and Yap, R., An Optimizing Compiler for CLP(R), in: U. Montanari and F. Rossi (eds.), *Principles and Practice of Constraint Programming—CP'95* (Lecture Notes in Computer Science 976), pp. 222–239, Springer-Verlag, 1995.

15. Le Charlier, B. and Van Hentenryck, P., Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog, *ACM Transactions on Programming Languages and Systems* **16** (1): 35–101, 1994.

16. Marriott, K., Søndergaard, H., and Jones, N., Denotational Abstract Interpretation of Logic Programs, *ACM Transactions on Programming Languages and Systems* **16** (3): 607–648, 1994.

17. Marriott, K. and Stuckey, P., Approximating Interaction between Linear Arithmetic Constraints, in: M. Bruynooghe (ed.), *Logic Programming: Proc. 1994 Int. Symp.*, pp. 571–585, MIT Press, 1994.

18. Mulkers, A., *Live Data Structures in Logic Programs*, Lecture Notes in Computer Science 675, Springer-Verlag, 1993.

19. Muthukumar, K. and Hermenegildo, M., Compile-Time Derivation of Variable Dependency Using Abstract Interpretation, *Journal of Logic Programming* **13** (2&3): 315–347, 1992.

20. Søndergaard, H., An Application of Abstract Interpretation of Logic Programs: Occur Check Reduction, in: B. Robinet and R. Wilhelm (eds.), *Proc. ESOP 86* (Lecture Notes in Computer Science 213), pp. 327–338, Springer-Verlag, 1986.

21. Taylor, A., LIPS on a MIPS: Results from a Prolog Compiler for a RISC, in: D. Warren and P. Szeredi (eds.), *Logic Programming: Proc. Seventh Int. Conf.*, pp. 174–185, MIT Press, 1990.

22. Van Roy, P. and Despain, A., The Benefits of Global Dataflow Analysis for an Optimizing Prolog Compiler, in: S. Debray and M. Hermenegildo (eds.), *Logic Programming: Proc. North American Conf. 1990*, pp. 501–515, MIT Press, 1990.