Disjunctive Relational Abstract Interpretation for Interprocedural Program Analysis

Rémy Boutonnet, Nicolas Halbwachs

▶ **To cite this version:**

Rémy Boutonnet, Nicolas Halbwachs. Disjunctive Relational Abstract Interpretation for Interprocedural Program Analysis. 20th International Conference on Verification, Model Checking, and Abstract Interpretation, Jan 2019, Cascais, Portugal. pp.136-159, 10.1007/978-3-030-11245-5_7. hal-02006429

HAL Id: hal-02006429
https://hal.science/hal-02006429v1

Submitted on 4 Feb 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Disjunctive relational abstract interpretation for interprocedural program analysis*

Rémy Boutonnet, Nicolas Halbwachs

Univ. Grenoble Alpes, CNRS, Grenoble INP**, VERIMAG, 38000 Grenoble, France,
{remy.boutonnet, nicolas.halbwachs}@univ-grenoble-alpes.fr

Abstract. Program analysis by abstract interpretation using relational abstract domains — like polyhedra or octagons — easily extends from state analysis (construction of reachable states) to relational analysis (construction of input-output relations). In this paper, we exploit this extension to enable interprocedural program analysis, by constructing relational summaries of procedures. In order to improve the accuracy of procedure summaries, we propose a method to refine them into disjunctions of relations, these disjunctions being directed by preconditions on input parameters.

1 Introduction

Linear Relation Analysis (LRA [17]) — or polyhedral abstract interpretation — is a classical method for discovering invariant linear inequalities among the numerical variables of a program. This method is still one of the most powerful numerical program analysis techniques, because of the expressivity of the discovered properties. However, it is not applicable to large monolithic programs, because of its prohibitive complexity, in terms of number of involved variables — in spite of recent progress in polyhedra algorithmics [22,49,37]. An obvious solution consists in using it in a modular way: the analysis of reasonably small procedures can provide, once and for all, a *summary* as an input-output relation; this summary can be reused in the analysis of programs calling the procedure. The *relational* nature of LRA is, of course, beneficial in this process.

On the other hand, the numerous works on interprocedural analysis, often concluded that such a "bottom-up" approach — where a procedure is analyzed before its callers — generally results in very imprecise summaries, because the procedure is considered independently of its calling context. One can object that this imprecision can be also due to the poor expressivity of the used domains, in particular those commonly used in compilers (e.g., data-flow analysis [32]).

So interprocedural analysis can provide a solution to the prohibitive cost of LRA, which, in turn, can provide a convenient expressive power for expressing more accurate procedure summaries.

* This work has been partially supported by the European Research Council under the European Union's Seventh Framework Programme(FP/2007-2013) / ERC Grant Agreement nr. 306595 "STATOR".

** Institute of Engineering Univ. Grenoble Alpes

This idea of using LRA to synthesize input-output relations is quite straightforward and not new. In particular, it is systematically applied in the tool PIPS [3,23], which considers each basic statement as an elementary relation, and synthesizes the input-output relation of a full program by composing these relations bottom-up. In this paper, we specialize the approach to the synthesis of procedure summaries. An easy way for building a relational summary of a procedure consists in duplicating the parameters to record their initial value, then performing a standard LRA of the body, which provides the summary as the least upper bound (convex hull) of the results at return points of the procedure.

However, it appears that conjunctions of linear constraints, i.e., convex polyhedral relations, are too restrictive. Obviously, procedures may exhibit very different and irregular behaviors according to the values of conditions appearing in tests. For instance,

– in many cases, whether an outermost loop is entered at least once or not is very relevant for the global behavior of the procedure;
– when a procedure has several return points, they are likely to correspond to quite different behaviors;
– for a simple recursive procedure, the base case(s) should be distinguished from those which involve recursive calls.

So it is natural to look for summaries that are *disjunctions* of polyhedral relations. However, algorithms for manipulating polyhedra do not extend easily to general disjunctions of polyhedra. A solution consists in using *trace partitioning* [9,28,38,47]. This solution is used in [24,26], where the partitioning is directed by formulas on Boolean variables. Here, we will propose such a partitioning directed by well-chosen *preconditions* on input parameters.

Contributions: While being mainly interested in LRA, we consider a more general framework. We provide a general formalization of relational abstract interpretation, that we didn't find elsewhere. As its use for computing procedure summaries often provides too rough results, we propose an approach to build disjunctive summaries, based on precondition partitioning. The choice of partitions is a heuristic process. We propose a method based on successive partition refinements, guided, on one hand, by the reachability of control points, and on the other hand, by the partitioning of summaries of called procedures. The method has been implemented in a prototype analyzer. Our experiments give encouraging results.

The paper is organized as follows. To situate our work, we first survey the abundant literature on interprocedural program analysis (Section 2). Since our approach can be applied in a more general context than LRA, we will develop each aspect in a stratified fashion: first, we consider the very general framework, then we present a specialization to LRA, before an application on a running example. Section 3 is concerned with concrete relational semantics of programs, and introduces the notations in the general framework and for numerical programs, together with our running example. Sections 4 and 5 deal with relational abstract interpretation and its use for building procedure summaries relative to a precondition. In view of the results on our example, in Section 6, we propose to

compute disjunctive summaries directed by a partition of preconditions. In Section 7, we present a way of partitioning preconditions by successive refinements. The application of our method to recursive procedures is illustrated in Section 8. Section 9 briefly presents our prototype implementation, and some experiments are described in Section 10. Section 11 gives the conclusion and sketches some future work.

2 Related Work

Interprocedural analysis originated in side-effects analysis, from works of Spillman [51], Allen [1,2] and Barth [6].

Interprocedural analyses can be distinguished according to the order in which procedures are traversed. In top-down analyses, procedures are analyzed following their invocation order [2], from callers to callees, while in bottom-up analyses, procedures are analyzed according to the inverse invocation order, from the callees up to the callers, by computing procedure summaries. Hybrid analyses [53] combine top-down and bottom-up analyses. We are interested in bottom-up approaches since each procedure is analyzed only once, regardless of the calling contexts, in possibly much smaller variable environments, thereby allowing a modular analysis with potential scalability improvements for numerical analyses such as LRA.

Sharir and Pnueli [48] introduced the functional approach and the call strings approach for distributive data flow frameworks. The functional approach computes procedure summaries, either from the bottom-up composition of individual propagation functions or by propagating data flow properties in a top-down fashion and by tabulating properties obtained at the exit node of a procedure with the associated property at entry. In the call strings approach, data flow properties are tagged by a finite string which encodes the procedure calls encountered during propagation. Call strings are managed as stacks and updated during propagation through a procedure call or return.

Reps et al. [46] proposed an algorithm belonging to the family of functional approaches to solve data flow problems with finite semilattices and distributive propagation functions in polynomial time, by recasting these data flow problems into graph reachability problems. Jeannet et al. [30,50] proposed a method reminiscent of the call strings approach, for the relational numerical analysis of programs with recursive procedures and pointers to the stack. It is a top-down approach based on an abstraction of the stack. An implementation is available in the Interproc tool [25]. Abstract states are partitioned according to Boolean conditions, but not according to possible input abstract states of a procedure. Yorsh et al. [52] proposed a bottom-up approach for finite distributive data flow properties and described how precise summaries for this class of properties can be constructed by composition of summaries of individual statements.

A relational abstraction of sets of functions for shape analysis is proposed in [27], considering functions of signature $D_1 \to D_2$, provided that abstractions A_1 of $\mathcal{P}(D_1)$ and A_2 of $\mathcal{P}(D_2)$ exist, and that A_1 is of finite cardinality. This

abstraction is relational since it is able to express relations between images of abstract elements mapped by a set of functions, but the abstraction A_1 is required to be of finite cardinality, thus excluding numerical abstract domains such as convex polyhedra.

Gulwani et al. [20] proposed a backward analysis to compute procedure summaries as constraints that must be satisfied to guarantee that some generic assertion holds at the end of a procedure. A generic assertion is an assertion with context variables which can be instantiated by symbols of a given theory. Procedure summaries are obtained by computing weakest preconditions of generic assertions. These generic assertions must be given prior to the analysis, thus forbidding the automatic discovery of procedure properties.

Cousot and Cousot [15,16] describe the symbolic relational separate analysis for abstract interpretation, which uses relational domains, relational semantics and symbolic names to represent initial values of variables modified by a procedure. When instantiated with the convex polyhedra abstract domain, this approach computes procedure summaries which are input-output relations represented by a single convex polyhedron, with no ability to capture disjunctive behaviors in procedures. Recursive procedures are supported, as presented earlier in [13,21,14].

Müller-Olm et al. [42,40] proposed an interprocedural bottom-up analysis to discover all Herbrand equalities between program variables in polynomial time. This approach was extended to linear two-variables equalities [18] and to affine relations [41]. This approach considers only abstracted programs with affine assignments, ignoring conditions on branches and dealing conservatively with other assignments. We are proposing a more general approach, which is also able to capture some disjunctive behaviors.

In the PIPS tool [23,3], statements are abstracted by affine transformers [35,36] which are input-output relations represented by convex polyhedra. The summary of a whole procedure is obtained from the composition of statement transformers, in a bottom-up fashion. Recursive procedures are not supported and each procedure summary is a single affine input-output relation, preventing the expression of disjunctive behaviors.

Popeea et al. [43,44,45] presented an analysis to both prove user-supplied safety properties and to find bugs by deriving conditions leading either to success or failure in each procedure. Disjunctive numerical properties are handled by a complete decision procedure for linear arithmetic provided by the Omega Test [31]. Our approach is able to discover automatically some disjunctive behaviors of procedures without requiring user-provided assertions.

Kranz et al. [33] proposed a modular analysis of executables based on Heyting completion [19]. Unfortunately, in the convex polyhedra abstract domain, the pseudo-complement $a \Rightarrow b = \sqcup\{d \mid a \sqsubseteq d \sqsubseteq b\}$ of a relative to b is not available in general.

3 Concrete Relational Semantics

3.1 General Framework

In our general framework, a program or a procedure is just a transition system. We introduce below a few definitions and notations.

States and Relations: Let S be a set of states. Let 2^S be the powerset of S. Let $\mathcal{R} = 2^{S\times S}$ be the set of binary relations on S.

- We define src, tgt the projection functions $\mathcal{R} \mapsto 2^S$ such that: $\forall r \in \mathcal{R}$,

$$src(r) = \{s_0 \in S \mid \exists s_1 \in S, (s_0, s_1) \in r\}\ ,\ tgt(r) = \{s_1 \in S \mid \exists s_0 \in S, (s_0, s_1) \in r\}$$

- If $U \subseteq S$, we define Id_U the relation $\{(s,s) \mid s \in U\}$.
- If $r_1, r_2 \in \mathcal{R}$, we denote by $r_1 \circ r_2$ their composition:

$$r_1 \circ r_2 = \{(s,s') \mid \exists s'', (s,s'') \in r_1 \text{ and } (s'', s') \in r_2\}$$

Forward, Backward Relational Semantic Equations: Let $\rho \in \mathcal{R}$ be a transition relation on S. We are interested in computing an upper approximation of its transitive closure ρ^*, which can be defined as a least fixpoint:

$$\begin{aligned} \rho^* &= \mu r. Id_S \cup (r \circ \rho) \quad \text{(forward equation)} \\ &= \mu r. Id_S \cup (\rho \circ r) \quad \text{(backward equation)} \end{aligned}$$

Trace Partitioning: We use the classical "trace partitioning" technique [38,47]. Assume that the set S is finitely partitioned : $S = S_1 \oplus S_2 \oplus \ldots \oplus S_n$. This partitioning can reflect the control points in a program or a control-flow graph, but it can also be more "semantic", and express state properties, like preconditions. If $r \in \mathcal{R}$, for each $i, j \in \{1, \ldots, n\}$, we define $r(S_i, S_j) = r \cap (S_i \times S_j)$.

With these notations, the relations $\rho^*(S_i, S_j)$ can be defined by the following system of fixpoint equations (henceforth, we consider only forward computation, backward computation is symmetrical):

$$\begin{aligned} \forall j \neq i,\ \rho^*(S_i, S_j) &= \bigcup_{k=1}^{n} \rho^*(S_i, S_k) \circ \rho(S_k, S_j) \\ \rho^*(S_i, S_i) &= Id_{S_i} \cup \bigcup_{k=1}^{n} \rho^*(S_i, S_k) \circ \rho(S_k, S_i) \end{aligned}$$

Concrete Relational Summaries: Let p be a procedure, $S, \rho, \mathcal{I}, \mathcal{E}$, respectively, its set of states, its transition relation, its sets of initial states (global precondition) and exit states. We assume that S is partitioned, and that $\mathcal{I}, \mathcal{E}$ belong to the partition. The *concrete relational summary* of p is $\sigma_p = \rho^*(\mathcal{I}, \mathcal{E})$. So, for the forward computation of the summary, we are concerned with the computation of $\rho^*(\mathcal{I}, S_j),\ j = 1..n$, according to the equations

$$\rho^*(\mathcal{I}, S_j) = \left(\bigcup_{k=1}^{n} \rho^*(\mathcal{I}, S_k) \circ \rho(S_k, S_j)\right) \cup \left\{ \begin{array}{c} Id_{\mathcal{I}} \text{ if } S_j = \mathcal{I} \\ \emptyset \quad \text{otherwise} \end{array} \right\}$$

Concrete Semantics of Procedure Calls: Let S be the set of states of a procedure p, T be the set of states of a program calling p. For a given call to p, let us write π the mapping $\in 2^{S\times S} \mapsto 2^{T\times T}$ representing the parameter passing mechanism (generally, a renaming of formal parameters into actual ones). Then, if T_i (resp. T_j) represents the sets of states just before (resp., just after) the call, the elementary relation corresponding to the call is $\rho(T_i, T_j) = \pi(\sigma_p)$.

3.2 Numerical Programs and Procedures

Procedures: For simplicity, and without loss of generality, the following assumptions are taken:

- All procedure parameters are supposed to be passed by reference. However, we are not concerned with pointer manipulation, and we entrust existing analyses to detect aliasing problems.
- Global variables are dealt with as additional parameters.
- For clarity, we will consider that all variables are parameters, since local variables don't raise any problem, but complicate the presentation.

In LRA, only numerical variables — taking their values in a numerical domain $\mathcal{N}$ ($= \mathbb{Z}$ or $\mathbb{Q}$) — are considered. A state of a numerical procedure with n variables is thus a pair (c, V), where $c \in C$ is a control point (a line, a statement, a block in a control-flow graph, ...), and $V = (v_1, ..., v_n) \in \mathcal{N}^n$ is a vector of numerical values. Control points provide a natural partitioning of such a set of states: $S_c = \{(c, V) \mid V \in \mathcal{N}^n\}$. The set of initial states $\mathcal{I}$ of a procedure with entry point $c_\mathcal{I}$ is such an $S_{c_\mathcal{I}}$, possibly restricted by a precondition $A_\mathcal{I} \subseteq \mathcal{N}^n$ on parameter values: $\mathcal{I} = \{(c_\mathcal{I}, V) \mid V \in A_\mathcal{I}\}$.

From State to Relational Collecting Semantics: Given such a partition $\{S_c \mid c \in C\}$, the usual collecting semantics defines the set A_c of reachable variable valuations in each S_c, such that $A_c = \{V \mid (c, V) \text{ is a reachable state from } \mathcal{I}\}$, as the least solution of a system of fixpoint equations:

$$A_c = F_c\left(\{A_{c'} \mid c' \in C\}\right) \cup \begin{cases} A_\mathcal{I} & \text{if } c = c_\mathcal{I} \\ \emptyset & \text{otherwise} \end{cases}$$

where the semantic function F_c expresses how the states in S_c depends on the states at other control points. This state semantics can be straightforwardly extended to relational semantics as follows: for each variable v_i, a new variable v_i^0 is introduced to record the initial value of v_i. The new set of states is thus $C \times \mathcal{N}^{2n}$, and the new initial state is

$$\mathcal{I} = \{(c_\mathcal{I}, (v_1^0, \ldots v_n^0, v_1, \ldots, v_n)) \mid (v_1^0, \ldots, v_n^0) \in A_\mathcal{I} \wedge v_i = v_i^0, i = 1..n\}$$

The relational semantics is equivalent to the state semantics of the same procedure, initialized with the assignments $v_i^0 = v_i$ for each $i = 1..n$.

Concrete Relational Summary: Let $\mathcal{E} \subset C$ be the set of exit points of the procedure. Then, $\bigcup_{c\in\mathcal{E}} A_c$ is the concrete summary of the procedure. In presence of local variables, they should be eliminated from this expression by existential quantification.

3.3 A Very Simple Example

Our example program is the classical Euclidean division, shown below with its relational semantic equations:

```
   void div (int a, b, *q, *r){
      assume (a ≥ 0 && b ≥ 1);
1     *q=0; *r=a;
2     while
3        (*r ≥ b) {
4        *r = *r-b; *q = *q+1;
5     }
6  }
```

$$A_1 = \{(a^0, b^0, q^0, r^0, a, b, q, r) \mid a^0 \geq 0 \wedge b^0 \geq 1 \wedge a = a^0 \wedge b = b^0 \wedge q = q^0 \wedge r = r^0\}$$
$$A_2 = A_1[q \leftarrow 0][r \leftarrow a]$$
$$A_3 = A_2 \cup A_5$$
$$A_4 = A_3 \cap (r \geq b)$$
$$A_5 = A_4[r \leftarrow r - b][q \leftarrow q + 1]$$
$$A_6 = A_3 \cap (r \leq b - 1)$$

The least solution for A_6, the unique exit point, is the concrete summary of the procedure: $a = a^0 \wedge b = b^0 \wedge a = bq + r \wedge q \geq 0 \wedge b - 1 \geq r \geq 0$. Notice that it contains a non linear relation, so it cannot be precisely obtained by LRA. For simplicity, we pretended to duplicate all parameters. Of course, in practice, pure input parameters ("value" parameters, whose value is not changed in the procedure) as well as pure output parameters ("result" parameters, whose initial value is not used in the procedure) don't need to be duplicated.

4 Relational Abstract Interpretation

4.1 General Framework

Relational Abstract Domains: A *relational abstract domain* is a complete lattice $(\mathcal{R}^\sharp, \sqsubseteq, \bot, \top, \sqcap, \sqcup)$ related to $\mathcal{R}$ by a Galois connection, i.e., a pair of increasing functions: $\alpha_\mathcal{R} : \mathcal{R} \mapsto \mathcal{R}^\sharp$ (abstraction), $\gamma_\mathcal{R} : \mathcal{R}^\sharp \mapsto \mathcal{R}$ (concretization), such that $\forall r \in \mathcal{R}, r^\sharp \in \mathcal{R}^\sharp$, $\alpha(r) \sqsubseteq r^\sharp \Leftrightarrow r \subseteq \gamma(r^\sharp)$.

If $U \subseteq S$, we denote by $Id^\sharp_U$ the abstract relation $\alpha_\mathcal{R}(Id_U)$. If $r^\sharp_1, r^\sharp_2 \in \mathcal{R}^\sharp$, we define $r^\sharp_1 \circ r^\sharp_2$ their composition as $\alpha_\mathcal{R}(\gamma_\mathcal{R}(r^\sharp_1) \circ \gamma_\mathcal{R}(r^\sharp_2))$. A relational abstract domain induces two abstract domains, $S^\sharp_\rightarrow$ and $S^\sharp_\leftarrow$ on 2^S:

$$\forall U \subseteq S, \alpha_{S_\rightarrow}(U) = \alpha_\mathcal{R}(U \times S) \quad , \quad \alpha_{S_\leftarrow}(U) = \alpha_\mathcal{R}(S \times U)$$

Notice that both $S^\sharp_\rightarrow$ and $S^\sharp_\leftarrow$ are included in $\mathcal{R}^\sharp$. We can define the abstract projections $src^\sharp : \mathcal{R}^\sharp \mapsto S^\sharp_\rightarrow$ and $tgt^\sharp : \mathcal{R}^\sharp \mapsto S^\sharp_\leftarrow$ by:

$$src^\sharp(r^\sharp) = \alpha_{S_\rightarrow}(src(\gamma_\mathcal{R}(r^\sharp))) \quad , \quad tgt^\sharp(r^\sharp) = \alpha_{S_\leftarrow}(tgt(\gamma_\mathcal{R}(r^\sharp)))$$

Relational Abstract Analysis: Let ρ be a transition relation, and $\rho^\sharp$ be an upper bound of its abstraction. We assume the availability of both a widening and a narrowing operation $\nabla, \Delta : \mathcal{R}^\sharp \times \mathcal{R}^\sharp \mapsto \mathcal{R}^\sharp$. Classically [12], an upper approximation of $\rho^{\sharp *}$ can be obtained by computing the limit $r^{\sharp \nabla}$ of an increasing approximation sequence:

$$r^\sharp_0 = \bot, \; r^\sharp_{n+1} = r^\sharp_n \nabla (r^\sharp_n \circ \rho^\sharp)$$

then the limit $r^{\sharp\nabla\Delta}$ of a decreasing sequence:

$$r'^{\sharp}_0 = r^{\sharp\nabla}, r'^{\sharp}_{n+1} = r'^{\sharp}_n \Delta(r'^{\sharp}_n \circ \rho^\sharp)$$

The result $r^{\sharp\nabla\Delta}$ is an abstract approximation of ρ^*, i.e., $\rho^* \subseteq \gamma(r^{\sharp\nabla\Delta})$.

Abstract Partition: For $\leftrightarrow \in \{\leftarrow, \rightarrow\}$, we define an abstract partition of $S^\sharp_\leftrightarrow$ as a finite set $\{S^\sharp_0, ..., S^\sharp_n\} \subseteq S^\sharp_\leftrightarrow$, such that $\{S_i = \gamma_{S_\leftrightarrow}(S^\sharp_i) \,|\, i = 1..n\}$ is a partition of S. More generally, if $U \subseteq S$, an abstract partition of $U^\sharp_\leftrightarrow = \alpha_{S_\leftrightarrow}(U)$ is a finite set $\{U^\sharp_0, ..., U^\sharp_n\} \subseteq U^\sharp_\leftrightarrow$, such that $\{U_i = \gamma_{S_\leftrightarrow}(U^\sharp_i) \,|\, i = 1..n\}$ is a partition of U.

Partitioned Relational Abstract Analysis: Let $\{S_i = \gamma(S^\sharp_i) \,|\, i = 1..n\}$ be a partition of S, let ρ be a transition relation, $\rho(S_i, S_j)$ be defined as before for $i, j = 1..n$, and $\rho^\sharp(S^\sharp_i, S^\sharp_j)$ be (an upper bound of) the abstraction of $\rho(S_i, S_j)$. An upper approximation of the vector $\{\rho^{\sharp*}(S^\sharp_i, S^\sharp_j) \mid i, j = 1..n\}$ can be obtained as the limit of (vectorial) increasing-decreasing sequences corresponding to the system of fixpoint equations:

$$\forall i = 1..n, \forall j \neq i, \; \rho^{\sharp*}(S^\sharp_i, S^\sharp_j) = \bigsqcup_{k=1}^{n} \rho^{\sharp*}(S^\sharp_i, S^\sharp_k) \circ \rho^\sharp(S^\sharp_k, S^\sharp_j)$$

$$\rho^{\sharp*}(S^\sharp_i, S^\sharp_i) = Id^\sharp_{S^\sharp_i} \sqcup \bigsqcup_{k=1}^{n} \rho^{\sharp*}(S^\sharp_i, S^\sharp_k) \circ \rho^\sharp(S^\sharp_k, S^\sharp_i)$$

Abstract Summary and Abstract Effect of a Procedure Call: Let p be a procedure, $\mathcal{I}, \mathcal{E}$ its set of initial and exit states. The abstract summary of p is $\sigma^\sharp_p = \rho^{\sharp*}(\mathcal{I}^\sharp, \mathcal{E}^\sharp)$. The abstract effect of a call to p, with parameter passing π, situated between $T^\sharp_i$ and $T^\sharp_j$ is $\rho^\sharp(T^\sharp_i, T^\sharp_j) = \pi(\sigma^\sharp_p)$.

4.2 Building Summaries using LRA

LRA makes use of the lattice of convex polyhedra [17,5]. It abstracts a set of numerical vectors by its convex hull (i.e., its least convex superset). Notice that the convex hull of an infinite set of vectors is not necessarily a polyhedron, but the finiteness of the analysis — thanks to the use of a widening operation — ensures that all the computed approximations are polyhedra, i.e., sets of solutions of a finite system of affine inequalities.

Intersection ($P_1 \sqcap P_2$), convex hull ($P_1 \sqcup P_2$), projection ($\exists X.P$), effect of variable assignment ($P[x \leftarrow exp]$, widening ($P_1 \nabla P_2$), test for inclusion ($P_1 \sqsubseteq P_2$) and emptiness ($P = \emptyset$) are available. Instead of using a narrowing operator to ensure the finiteness of the decreasing sequence, a limited number of iterations of the abstract function is generally applied.

Polyhedra can be used for representing input-output relations, as an abstraction of the relational semantics described in Section 3.2. We write $P(X^0, X)$ a polyhedron involving initial values X^0 and current values X. Notice that the

source and the target of the relation r expressed by $P(X^0, X)$ can be obtained by polyhedron projections:

$$src^\sharp(r) = \exists X.P(X^0, X)\ ,\ tgt^\sharp(r) = \exists X^0.P(X^0, X)$$

4.3 Example

Let us apply LRA to our Euclidean division example. The abstract equations are as follows:

$$\begin{array}{l|l} \multicolumn{2}{l}{P_1 = (a^0 \geq 0, b^0 \geq 1, a = a^0, b = b^0, q = q^0, r = r^0)} \\ P_2 = P_1[q \leftarrow 0][r \leftarrow a] & P_4 = P_3 \sqcap (r \geq b) \\ P_3 = P_2 \sqcup P_5 & P_5 = P_4[r \leftarrow r - b][q \leftarrow q + 1] \\ P_6 = P_3 \sqcap (r \leq b - 1) & \end{array}$$

P_6 corresponds to the unique exit point of the procedure, so it is the summary. The standard analysis — where the widening is applied on P_3 during the increasing sequence, and the decreasing sequence is limited to 2 steps — provides:

$$P_6 = (a = a^0\ ,\ b = b^0\ ,\ r \geq 0\ ,\ q \geq 0\ ,\ b \geq r + 1)$$

It is a rather weak summary, all the more as the precondition $a^0 \geq 0$ has been lost. This suggests that preconditions should be considered more carefully.

5 Preconditions

For closed programs, the initial state is generally not relevant, since, normally, the variables are explicitly assigned an initial value before being used. When considering procedures, the initial state is implicitly defined by the initial values of parameters. Therefore, it is essential to take it into account. In particular, the correct behavior of a procedure often depends on (user-defined) preconditions on parameter values. We will call *global precondition* the abstraction of the set of legal initial states of a procedure: $\mathcal{I}_p^\sharp = \alpha_{S_\rightarrow}(\mathcal{I}_p)$. Notice that we already took into account the global precondition $a \geq 0, b \geq 1$ in our example. Such global precondition may be given by the user, or deduced from another analysis of the calling contexts, or simply $\top$.

Moreover, preconditions can be used to differentiate cases of input values (calling contexts) that should be considered separately. These preconditions will be obtained by refining the global precondition. This is the way we intend to build disjunctive summaries.

Widening under a Precondition: In relational analysis, a precondition provides an obvious invariant: a procedure may not change its initial state, so any concrete relation $\rho^*(\mathcal{I}_p, S_i)$ has its source within $\mathcal{I}_p$. However, it is not as obvious with abstract analysis: because of the use of widening, it may happen that the result $r^{\sharp\nabla\Delta}$ does not satisfy this invariant, i.e., $\gamma_\mathcal{R}(r^{\sharp\nabla\Delta})$ is not included in $\mathcal{I}_p \times S$. This is what happened in our example (§4.3). As a consequence, it is sound and interesting to make use of a "limited widening" when computing $r^{\sharp\nabla}$: we define this more precise widening by $r\nabla_{\mathcal{I}_p^\sharp} r' = (r\nabla r') \sqcap \mathcal{I}_p^\sharp$.

Example: Coming back to our example in §4.3, the widening is performed on P_3. Instead of applying the widening classically, i.e., computing $P_3 = P_3 \nabla (P_2 \sqcup P_5)$, we limit it with the precondition, i.e., compute $P_3 = (P_3 \nabla (P_2 \sqcup P_5)) \sqcap (a^0 \geq 0, b^0 \geq 1)$. The summary we obtain

$$P_6 = (a = a^0\,,\ b = b^0\,,\ r \geq 0\,,\ q \geq 0\,,\ b \geq r+1\,,\ a \geq q+r)$$

recovers more than just the precondition. Instead of gaining just $a \geq 0$, we get the stronger $a \geq q + r$.

6 Disjunctive Summaries

Up to now, we described the classical analysis by abstract interpretation, with an emphasis on relational analysis, use of trace partitioning, and taking care of preconditions. In this section, we propose to refine the partitioning by distinguishing the calling contexts of a procedure, defined as preconditions.

Abstract domains are generally not closed under disjunction (in some sense, it is the essence of abstraction). In order to build more precise procedure summaries, it is natural to consider disjunctions of abstract relations. However, some restrictions must be applied to be able to compute on such disjunctions. Moreover, in order to be able to exploit such a disjunctive procedure summary when using it on a procedure call, the values of the actual parameters should determine which disjunct must apply. Thus, different disjuncts should have disjoint sources.

6.1 Disjunctive Refinements of an Abstract Relation

If p is a procedure with global precondition $\mathcal{I}$, a disjunctive refinement of the abstract relation $\rho^{\sharp*}(\mathcal{I}^\sharp, S_i^\sharp)$ will be a finite set $r_1^{\sharp*}, ..., r_m^{\sharp*}$ of abstract relations, such that

$$\begin{array}{ll} (1) & \forall k = 1..m, r_k^{\sharp*} \sqsubseteq \rho^{\sharp*}(\mathcal{I}^\sharp, S_i^\sharp) \\ (2) & \forall k_1, k_2 = 1..m, k_1 \neq k_2 \Rightarrow \gamma(src^\sharp(r_{k_1}^{\sharp*})) \cap \gamma(src^\sharp(r_{k_2}^{\sharp*})) = \emptyset \\ (3) & \displaystyle\bigcup_{k=1}^{m} \gamma(src^\sharp(r_k^{\sharp*})) = \gamma(\mathcal{I}^\sharp) \end{array}$$

In other words, $\{src^\sharp(r_k^{\sharp*})\}_{k=1..m}$ forms an abstract partition of $\mathcal{I}^\sharp$. Notice that, with this definition, the disjunctive summary of a procedure can also be seen as a conjunction of implications:

$$\bigvee_{k=1}^{m} r_k^{\sharp*} \iff \bigwedge_{k=1}^{m} \left(src^\sharp(r_k^{\sharp*}) \Rightarrow r_k^{\sharp*}\right)$$

emphasizing the fact that the partitioning is directed by properties of input parameters. Conversely, given an abstract partition $\{\mathcal{I}_k^\sharp\}_{k=1..m}$ of $\mathcal{I}^\sharp$, one can compute a disjunctive refinement of the abstract relation $\rho^{\sharp*}(\mathcal{I}^\sharp, S_i^\sharp)$ simply by computing $r_k^{\sharp*} = \rho^{\sharp*}(\mathcal{I}_k^\sharp, S_i^\sharp)$ for each $k = 1..m$.

6.2 Disjunctive Abstract Summary and Abstract Effect of a Call

Given a disjunctive refinement $\{r_k^{\sharp *}\}_{k=1..m}$ of an abstract relation, the corresponding abstract summary of a procedure is a set of disjuncts:

$$\{\sigma_k^\sharp = r_k^{\sharp *}(\mathcal{I}_p^\sharp, \mathcal{E}_p^\sharp)\}_{k=1..m}$$

Given such a disjunctive summary, the abstract effect of a call to p, with parameter passing π, situated between $T_i^\sharp$ and $T_j^\sharp$ is

$$\rho^\sharp(T_i^\sharp, T_j^\sharp) = \bigsqcup_{k=1}^{m} \pi(\sigma_k^\sharp)$$

6.3 Application to LRA

Disjunctive Polyhedral Summaries: Let p be a procedure, X be its vector of variables, and $\mathcal{I}^\sharp$ be its polyhedral global precondition. A disjunctive polyhedral summary of p is a disjunction of input-output relations expressed by a set of polyhedra $\{R_1, ..., R_m\}$, and such that, if we define $\mathcal{I}_k^\sharp = src^\sharp(R_k) = \exists X.R_k$ $(k = 1..m)$, the set $\{\mathcal{I}_k^\sharp\}_{k=1..m}$ forms an abstract partition of $\mathcal{I}^\sharp$.

Polyhedron Transformer of a Procedure Call: With the same notations concerning the procedure p and its disjunctive polyhedral summary, assume that $X = (x_1, \ldots, x_n)$ is the list of formal parameters. Let q be a caller to p, $A = (a_1, \ldots, a_n)$ be the actual parameters of a call to p situated between control points c and c' in q. Let Q_c be the polyhedron associated with c in q. Then the polyhedron associated with the return point c' is

$$Q_{c'} = \bigsqcup_{k=1}^{m} \left(\exists A^1.Q_c[A/A^1] \sqcap R_k[X^0/A^1][X/A]\right)$$

where

- $Q_c[A/A^1]$ is the result of renaming, in Q_c, each variable a_i as a_i^1
- $R_k[X^0/A^1][X/A]$ is the result of renaming, in R_k, each variable x_i^0 as a_i^1, and each variable x_i as a_i (this term is what we wrote $\pi(\sigma_k^\sharp)$ in the general framework §6.2).

In other words, the auxiliary variables $A^1 = (a_1^1, \ldots, a_n^1)$ represent the values of actual parameters before the call, so they are substituted for A in the calling context Q_c and to X^0 in the summary; the values A of the actual parameters after the call, are substituted for X in the summary.

7 Partition Refinement

7.1 General Framework

Given an abstract partition of the global precondition of a procedure, we know how to compute and use a disjunctive summary based on this partition. In this section, we propose a heuristic method to choose the abstract partition.

Complementable Abstract Values: An abstract value $r^\sharp$ is said to be complementable, if there exists an abstract value $\overline{r^\sharp}$ (its complement) such that $r^\sharp \sqcap \overline{r^\sharp} = \bot$ and $\gamma(r^\sharp) \cup \gamma(\overline{r^\sharp}) = \mathcal{R}$. For instance, complementable convex polyhedra are half-spaces, i.e., polyhedra defined by a single inequality.

Refinement According to Local Reachability: Let $\{r^{\sharp\nabla\Delta}(\mathcal{I}^\sharp, S_i^\sharp)\}_{i=1..n}$ be the result of a classic analysis from a precondition $\mathcal{I}^\sharp$. For a given $i \in \{1..n\}$, $\mathcal{I}_i^\sharp = src^\sharp(r^{\sharp\nabla\Delta}(\mathcal{I}^\sharp, S_i^\sharp))$ is a necessary condition for $S_i^\sharp$ to be reachable. As a consequence, if $s^\sharp$ is a complementable abstract value such that

- $\mathcal{I}_i^\sharp \sqsubseteq s^\sharp$
- $\mathcal{I}'^\sharp = \mathcal{I}^\sharp \sqcap s^\sharp \neq \bot$ and $\mathcal{I}''^\sharp = \mathcal{I}^\sharp \sqcap \overline{s^\sharp} \neq \bot$

then $(\mathcal{I}'^\sharp, \mathcal{I}''^\sharp)$ is a good candidate for refining the precondition $\mathcal{I}^\sharp$. As a matter of fact, $\mathcal{I}''^\sharp$ is a sufficient precondition for $S_i^\sharp$ to be unreachable.

Refinement According to the Summary of a Called Procedure: The effect of a call to a procedure with a partitioned summary $\{\sigma_k^\sharp\}_{k=1..m}$ (as defined in §6.2) involves a least upper bound $\bigsqcup_{k=1}^m \pi(\sigma_k^\sharp)$, which is likely to lose precision. So it is interesting to refine the partition in the caller in order to split this least upper bound. Let us denote by $\mathcal{J}_k^\sharp = \pi(src^\sharp(\sigma_k^\sharp))$, i.e., the condition on actual parameters for $\sigma_k^\sharp$ to be applicable. Then, in the caller, $\mathcal{I}_k^\sharp = src^\sharp(r^{\sharp\nabla\Delta}(\mathcal{I}^\sharp, \mathcal{J}_k^\sharp))$, is a necessary precondition for $\mathcal{J}_k^\sharp$ to be satisfiable. As a consequence, if $s^\sharp$ is a complementable abstract value such that

- $\mathcal{I}_k^\sharp \sqsubseteq s^\sharp$
- $\mathcal{I}'^\sharp = \mathcal{I}^\sharp \sqcap s^\sharp \neq \bot$ and $\mathcal{I}''^\sharp = \mathcal{I}^\sharp \sqcap \overline{s^\sharp} \neq \bot$

then $(\mathcal{I}'^\sharp, \mathcal{I}''^\sharp)$ is a good candidate for refining the precondition $\mathcal{I}^\sharp$. As a matter of fact, $\mathcal{I}''^\sharp$ is a sufficient precondition for $\mathcal{J}_k^\sharp$ to be unsatisfiable at the call.

Iterative Refinements: Our proposal is to build the summary of a procedure as the result of a sequence of analyses, working on more and more refined partitions. We define $\mathcal{P}^{(\ell)} = \{\mathcal{I}_k^{\sharp(\ell)}\}_{k=1..m_\ell}$ the partition of abstract preconditions considered at ℓ-th analysis. Starting with $\mathcal{P}^{(0)} = \{\mathcal{I}^\sharp\}$ (the singleton made of the global precondition of the procedure), for each ℓ, we compute from $\mathcal{P}^{(\ell)}$ the corresponding disjunctive abstract relation $\{r_k^{\sharp(\ell)}\}_{k=1..m_\ell}$, which is used to refine $\mathcal{P}^{(\ell)}$ into $\mathcal{P}^{(\ell+1)}$, using one of the refinement techniques presented above. This process is not guaranteed to terminate, but may be stopped at any step. In practice, the size of the partition will be limited by a constant parameter of the analysis.

Ensuring the Monotonicity of the Refinement: Refining a precondition is intended to provide a more precise summary. However, this is not guaranteed because of the non-monotonicity of the widening operator. So at step ℓ, when precondition $\mathcal{I}_k^{\sharp(\ell)}$ has been split into a pair $(\mathcal{I}_{k'}^{\sharp(\ell+1)}, \mathcal{I}_{k''}^{\sharp(\ell+1)})$ of new preconditions, the analyses performed at step $\ell+1$ from these new preconditions should use a widening limited by $r_k^{\sharp(\ell)}$. The monotonicity of the refinement is especially

important when dealing with recursive procedures, and avoids the difficulties tackled by [4].

7.2 Application to LRA

Complementable Polyhedra: As said before, complementable polyhedra are those defined by a single inequality. So any polyhedron is the intersection of a finite number of complementable polyhedra. The complement of "$aX \leq b$" is obtained either with the converse *strict* inequality "$aX > b$" (strict inequalities are handled in the PPL [5,8] and in Apron [29]), or, in case of integer variables, by the inequality "$aX \geq b + 1$".

Precondition Refinement: From a precondition $\mathcal{I}^\sharp$, a standard analysis by LRA provides, at each control point c of the program, a polyhedron $P_c(\mathcal{I}^\sharp)$. From these solutions, we can try to refine the precondition:

- For each control point c, let $Q_c = \exists X.P_c(\mathcal{I}^\sharp)$ be the projection of $P_c(\mathcal{I}^\sharp)$ on initial variables. Then, if $Q_c \neq \mathcal{I}^\sharp$, any constraint χ of Q_c not satisfied by $\mathcal{I}^\sharp$ can be used to separate $\mathcal{I}^\sharp$ into $\mathcal{I}_1^\sharp = \mathcal{I}^\sharp \cap \chi$ and $\mathcal{I}_2^\sharp = \mathcal{I}^\sharp \cap \overline{\chi}$, since the point c is unreachable by any execution starting in $\mathcal{I}_2^\sharp$. Obviously, this should be tried on control points following a test, and especially those corresponding to loop conditions.
- For each control point c corresponding to a call to a procedure, say $p(A)$, let $\{R_1, ..., R_m\}$ be the polyhedral summary of p, and for each $k = 1..m$, $\mathcal{J}_k^\sharp(p) = src^\sharp(R_k)[X^0/A]$ (i.e., $\mathcal{J}_k^\sharp(p)$ is the precondition of R_k, expressed on actual parameters). Then, let $Q_{c,k} = \exists X.P_c(\mathcal{I}^\sharp) \sqcap \mathcal{J}_k^\sharp(p)$ be the projection of $P_c(\mathcal{I}^\sharp) \sqcap \mathcal{J}_k^\sharp(p)$ on the initial variables of the caller. Then, as before, if $Q_{c,k} \neq \mathcal{I}^\sharp$, any constraint χ of $Q_{c,k}$ not satisfied by $\mathcal{I}^\sharp$ can be used to separate $\mathcal{I}^\sharp$ into $\mathcal{I}_1^\sharp = \mathcal{I}^\sharp \cap \chi$ and $\mathcal{I}_2^\sharp = \mathcal{I}^\sharp \cap \overline{\chi}$, and it is interesting since starting the caller in $\mathcal{I}_2^\sharp$ makes empty the precondition $\mathcal{J}_k^\sharp(p)$.

Notice that, in both cases, the choice of the constraint χ is arbitrary, and that several such constraints can be used in turn. So the fact that the refinement is done according to one single constraint is not a limitation.

7.3 Example

The analysis of the example in §4.3, from the precondition $\mathcal{I}^{\sharp(0)} = (a^0 \geq 0, b^0 \geq 1)$, as done in §5, provides, on the branches of the loop condition $(r \geq b)$, the solutions:

$$P_4(\mathcal{I}^{\sharp(0)}) = (a^0 = a, b^0 = b, r \geq b, q \geq 0, b \geq 1, a \geq q + r)$$
$$P_6(\mathcal{I}^{\sharp(0)}) = (a^0 = a, b^0 = b, r \geq 0, q \geq 0, b \geq r + 1, a \geq q + r)$$

The projections of these solutions on the initial values are:

$$src^\sharp(P_4(\mathcal{I}^{\sharp(0)})) = (a^0 \geq b^0 \geq 1) \quad src^\sharp(P_6(\mathcal{I}^{\sharp(0)})) = (a^0 \geq 0, b^0 \geq 1)$$

$src^\sharp(P_6(\mathcal{I}^{\sharp(0)})) = \mathcal{I}^{(0)}$, so it does not induce any refinement. However, $src^\sharp(P_4(\mathcal{I}^{\sharp(0)})) \neq \mathcal{I}^{\sharp(0)}$, since $\mathcal{I}^{\sharp(0)}$ does not imply $a^0 \geq b^0$. We can refine $\mathcal{I}^{\sharp(0)}$ into

$$\mathcal{I}_1^{\sharp(1)} = (a^0 \geq b^0 \geq 1) \text{ and } \mathcal{I}_2^{\sharp(1)} = (b^0 - 1 \geq a^0 \geq 0)$$

i.e., separate the cases where the loop is entered at least once or not. New analyses from these refined preconditions provide:

$$\begin{array}{l} P_4(\mathcal{I}_1^{\sharp(1)}) = (a^0 = a, b^0 = b, r \geq b, q \geq 0, b \geq 1, a \geq q + r) \\ P_6(\mathcal{I}_1^{\sharp(1)}) = (a^0 = a, b^0 = b, r \geq 0, q \geq 0, q + r >= 1, b \geq r + 1, \\ \qquad\qquad\quad a + 1 \geq b + q, a \geq b) \\ P_4(\mathcal{I}_2^{\sharp(1)}) = \bot \\ P_6(\mathcal{I}_2^{\sharp(1)}) = (a^0 = a, b^0 = b, b - 1 \geq a \geq 0, q = 0, r = a) \end{array}$$

The projections of these solutions on the initial values are:

$$\begin{array}{l} src^\sharp(P_4(\mathcal{I}_1^{\sharp(1)})) = (a^0 \geq b^0 \geq 1) = \mathcal{I}_1^{\sharp(1)}) \\ src^\sharp(P_6(\mathcal{I}_1^{\sharp(1)})) = (a^0 \geq b^0 \geq 1) = \mathcal{I}_1^{\sharp(1)}) \\ src^\sharp(P_4(\mathcal{I}_2^{\sharp(1)})) = \bot \\ src^\sharp(P_6(\mathcal{I}_2^{\sharp(1)})) = (b^0 - 1 \geq a^0 \geq 0) = \mathcal{I}_2^{\sharp(1)} \end{array}$$

so, according to our criteria, the preconditions cannot be further refined, and we get the summary

$$\begin{array}{l} R_1 = \big(a^0 = a, b^0 = b, a^0 \geq b^0 \geq 1, \\ \qquad\quad r \geq 0, q \geq 0, q + r >= 1, b \geq r + 1, a + 1 \geq b + q) \\ R_2 = \big(a^0 = a, b^0 = b, b^0 - 1 \geq a^0 \geq 0, q = 0, r = a\big) \end{array}$$

directed by input conditions $R_1^0 = (a^0 \geq b^0 \geq 1)$ and $R_2^0 = (b^0 - 1 \geq a^0 \geq 0)$.

7.4 A Last Improvement: Postponing Loop Feedback

The previous example shows a weakness of the analysis: the summary has been partitioned according to whether the loop is entered at least once (R_1) or not (R_2). However, in the former case, since the loop body is executed at least once, we should obtain $q \geq 1$, a fact which is missed by the analysis. We could recover this fact by systematically unrolling once each loop that gives raise to such a partitioning. We propose another, cheaper solution. The problem comes from the least upper bound computed at loop entry ($P_3 = P_2 \sqcup P_5$ in the abstract equations of §4.3), before the test on the loop condition ($P_6 = P_3 \sqcap (r \leq b-1)$). The solution consists in permuting the least upper bound and the test, computing instead $P_6 = (P_2 \sqcap (r \leq b-1)) \sqcup (P_5 \sqcap (r \leq b-1))$[1].

[1] This change in the abstract equations could also be obtained by transforming each loop "*while c do B*" into "*if c {do B while c}*", a transformation called "loop inversion" often applied by compilers.

Back to the Example: Computing $R_1 = P_6(\mathcal{I}_1^{\sharp(1)})$ with this new equation, since $P_2 \sqcap (r \le b-1) = \bot$, we get

$$R_1 = \big(a^0 = a, b^0 = b, a^0 \ge b^0 \ge 1, r \ge 0, q \ge 1, b \ge r+1, a+1 \ge b+q+r\big)$$

Once again, we recover more precision than expected, since, in addition to finding $q \ge 1$, $a+1 \ge b+q$ is strengthened into $a+1 \ge b+q+r$.

8 Recursive Procedures

The relational abstract interpretation of recursive procedures was proposed a long time ago [13,21,15]. It involves the use of widening, since the summary of a recursive procedure depends on itself. Moreover, a group of mutually recursive procedures must be analyzed jointly, with widening applied on a cutset of their call graph. In this section, we only show a simple example of how our technique can be applied to build a disjunctive summary of a recursive procedure. It will also illustrate the refinement according to the summary of a called procedure.

Example: McCarthy's 91 Function. The opposite procedure is the well-known "91 function" defined by John McCarthy. For simplicity, we don't duplicate parameters, knowing that x is a value parameter and y is a result parameter. The polyhedral summary of the procedure can be defined by the following equations:

```
void f91 (int x,*y) {
     int z, t ;
1    if (x > 100) *y = x -10 ;
2
3    else {   z = x + 11 ;
4             f91 (z, &t) ;
5             f91 (t, y) ;
6    }
}
```

$$\begin{aligned}
&R(x,y) = P_2 \sqcup P_7\\
&P_2 = (x \ge 101, y = x-10)\\
&P_7 = (x \le 100 \sqcap (\exists t. R(x+11, t) \sqcap R(t,y)))
\end{aligned}$$

A first, standard analysis, without partitioning, reaches the following fixpoint after one widening step:

$$\begin{aligned}
&P_2 = (x \ge 101, y = x-10) \quad , \quad P_6 = (x \le 100, y+9 \ge x, y \ge 91)\\
&R^{(0)} = (x \le y+10, y \ge 91)
\end{aligned}$$

Since $src^\sharp(P_2) = (x \ge 101)$ splits the global precondition $\mathcal{I}^\sharp = \top$, we refine the precondition into $\mathcal{I}_1^{\sharp(1)} = (x \ge 101)$ and $\mathcal{I}_2^{\sharp(1)} = (x \le 100)$. From this (obvious) partition, the results are not much better:

$$\begin{aligned}
&P_2(\mathcal{I}_1^{\sharp(1)}) = (x \ge 101, y = x-10) &&, P_2(\mathcal{I}_2^{\sharp(1)}) = \bot\\
&P_6(\mathcal{I}_1^{\sharp(1)}) = \bot &&, P_6(\mathcal{I}_2^{\sharp(1)}) = (x \le 100, y \ge 91)\\
&R^{(1)}(\mathcal{I}_1^{\sharp(1)}) = (x \ge 101, y = x-10) &&, R^{(1)}(\mathcal{I}_2^{\sharp(1)}) = (x \le 100, y \ge 91)
\end{aligned}$$

But now, the partitioned precondition involves a refinement of $\mathcal{I}_2^{\sharp(1)}$ at the first recursive call, according to the condition $x+11 \ge 101$. We get $\mathcal{I}_1^{\sharp(2)} = (90 \le$

$x \leq 100$) and $\mathcal{I}_2^{\sharp(2)} = (x \leq 89)$. The final result is

$$
\begin{aligned}
R^{(1)}(\mathcal{I}_1^{\sharp(1)}) &= (x \geq 101, y = x - 10) \\
R^{(2)}(\mathcal{I}_1^{\sharp(2)}) &= (90 \leq x \leq 100, y = 91) \\
R^{(2)}(\mathcal{I}_2^{\sharp(2)}) &= (x \leq 89, y = 91)
\end{aligned}
$$

which is the most precise summary.

9 Implementation

This approach has been implemented in a prototype static analyzer. Organized as a collection of tools, the analyzer computes numerical invariants on programs written in a significant subset of C. A front-end tool based on Clang [34] and LibTooling translates the abstract syntax tree of a C program into an intermediate representation. The analyzer tool then computes numerical invariants on the intermediate representation. Abstract domains, such as convex polyhedra, are provided by the Apron [29] library. The analyzer can either consider an inlined version of the program, or construct and use procedure summaries as described in the paper, with some restrictions: for the time being, recursive procedures are not yet taken into account, and postponing the loop feedback is not performed as described in §7.4, but makes use of "loop inversion".

Procedures are analyzed only once, regardless of the number of call sites, in a bottom-up fashion according to the inverse invocation order, with respect to the dependencies induced by the program call graph.

In order to limit the number of additional variables, the tool does not duplicate all procedure parameters, but applies a simple dataflow analysis before summary construction to identify procedure parameters which are either pure input parameters or pure output parameters, and thus which do not need to be duplicated.

Refinement according to local reachability is performed by the analyzer only at direct successors of test nodes and particularly at loop entry and loop exit. Candidate nodes for refinement are examined during each refinement step using a breadth-first traversal of the program graph. For practical reasons, in order to guarantee the termination of the refinement process and to limit procedure summaries to a reasonable size, an upper-bound θ on the refinement depth for a given procedure is set to $\theta = 2$. This limits procedure summaries to a maximum size of 4.

10 Experiments

Up to now, to illustrate our approach, we presented only tiny examples, for which the precondition partitioning is obvious. However, in presence of more complex control structures — nested and/or successive loops —

and when preconditions result from more involved invariants, the usefulness of our method for discovering relevant preconditions is more convincing. Several more complex ad-hoc examples can be found on the repository github.com/programexamples/programexamples.

More thorough experiments are necessary to validate our approach, and in particular to answer the following questions:

– Since we analyze a procedure several times to construct its summary, it is likely to be time-consuming. So it is interesting to measure the cost of summary construction with respect to the time hopefully saved by using the summary.
– Precondition partitioning is a heuristic process, so it is important to evaluate the precision lost or gained by using a disjunctive summary instead of analyzing again the procedure for each calling context.

So our experiments consists in comparing our bottom-up approach with an analysis of inlined programs, both with respect to the analysis time and the precision of results. Several difficulties must be addressed first:

– Most public benchmarks are not usable, since they contain very few numerical programs with procedures. For instance, in the SV-COMP benchmark[2], most numerical examples are inlined; the ALICe benchmark[3] also contains only monolithic programs. For our assessment, we used the benchmark of the Mälardalen[4] WCET research group, which contains various small and middle-sized programs, such as sorts, matrix computations, fft, etc. Moreover, some programs of this benchmark were sometimes extended with auxiliary variables counting the number of executions of each block to help the evaluation of the execution time [10]; these extensions — the name of which are prefixed with "`cnt_`" below — are interesting for us, since they contains more numeric variables.
– The comparison of polyhedral results is not straightforward:
  - On one hand, we must decide which polyhedra to compare. The correspondence of control points between the inlined program and the structured one is not easy to preserve. In our experiments, we only compared the results at the end of the main program. Of course, for the comparison to be meaningful, the results on the inlined program must be first projected on the variables of the main program.
  - On the other hand, while a qualitative comparison of two polyhedra is easy — by checking their inclusion in both directions —, a quantitative comparison is more difficult: it could be precisely achieved by comparing their volumes — algorithms are available for that [7,11] — but it is only possible for bounded polyhedra. In our assessment, besides a qualitative comparison, we only compared the number of constraints.

All our experiments are done using the convex polyhedra abstract domain. Widening is never delayed and decreasing sequences are limited to 7 terms. The

[2] sv-comp.sosy-lab.org/2018/benchmarks.php
[3] alice.cri.mines-paristech.fr/models.html
[4] www.mrtc.mdh.se/projects/wcet/benchmarks.html

	#	max.	Inlining		Interprocedural		cmp.	
Program	procs	# calls	t_{IL}	C_{IL}	t_{IP}	C_{IP}	res.	S
fabs	2	1	0.013	4	0.015	4	=	0.87
fdct	2	1	0.084	0	0.069	0	=	1.22
fft1	6	3	0.742	4	0.465	3	<>	1.59
fir	2	1	0.040	1	0.072	1	=	0.55
janne_complex	2	1	0.948	5	0.062	8	$\sqsupset$	15.34
minver	4	2	0.155	1	0.686	2	$\sqsubset$	0.23
my_sin	2	1	0.032	1	0.028	5	$\sqsupset$	1.14
jfdctint	2	1	0.082	3	0.060	3	=	1.38
ludcmp	3	1	0.074	3	0.102	3	=	0.73
ns	2	1	0.057	0	0.051	0	=	1.13
qurt	4	1	0.057	1	0.028	1	=	2.06
select	2	1	0.097	0	0.057	0	=	1.69
ud	2	1	0.093	3	0.118	3	=	0.79
cnt_fdct	2	1	0.098	1	0.075	1	=	1.31
cnt_fft1	6	3	33.417	5	2.646	3	<>	12.63
cnt_jfdctint	2	1	0.102	5	0.070	5	=	1.46
cnt_ns	2	1	0.085	0	0.067	0	=	1.25
cnt_qurt	4	1	0.601	2	0.063	2	=	9.54
cnt_minver	4	2	1.008	1	3.424	6	$\sqsupset$	0.29

Table 1: Experimental results.

analysis times are those obtained on an Intel Xeon E5-2630 v3 2.40Ghz machine with 32GB of RAM and 20MB of L3 cache running Linux.

Table 1 compares our method with a standard LRA on inlined programs, in terms of analysis time, qualitative precision and number of constraints of results found at the exit points of the main procedures. The "# procs" column gives the number of procedures in each program and the "max. # calls" column gives the maximum number of call sites per procedure in a program. We define:

- t_{IL} (resp. t_{IP}) the time (in seconds) for analyzing the inlined program (resp. the time for interprocedural analysis)
- P_{IL} (resp. P_{IP}) the polyhedron result of the inlined analysis (resp., of the interprocedural analysis)
- C_{IL} (resp. C_{IP}) the number of constraints of P_{IL} (resp. of P_{IP}).

The qualitative results comparison is shown by column "cmp. res." which indicates whether the result P_{IP} is better ($\sqsupset$), worse ($\sqsubset$), equal ($=$) or incomparable (<>) w.r.t. P_{IL}. The S column gives for each program the speedup of our method defined as $S = t_{\text{IL}}/t_{\text{IP}}$. Our method is significantly faster than standard LRA using inlining for 13 over 19 programs ($\approx$ 68% of programs), with an average speedup of 2.9. The loss of precision is very moderate since only 1 over 19 programs, namely **minver**, has a less precise convex polyhedra at the exit node of the main procedure.

Interestingly, our method also leads to precision improvements for some programs, such as **janne_complex**, **my_sin** and **cnt_minver**, due to the use of

Program	Function	Time (s)	τ_c
fabs	fabs	0.001	0.067
fdct	fdct	0.050	0.588
fft1	my_fabs	< 0.001	0.001
	my_sin	0.002	0.004
	my_cos	< 0.001	0.001
	my_log	< 0.001	< 0.001
	fft1	0.350	0.753
fir	fir	0.019	0.267
janne	janne	0.037	0.602
minver	mmul	0.047	0.069
	minver_fabs	< 0.001	< 0.001
	minver	0.616	0.897
my_sin	my_sin	0.003	0.098
jfdctint	jpeg_fdct_islow	0.031	0.528
ludcmp	fabs	< 0.001	0.002
	ludcmp	0.055	0.540
ns	foo	0.011	0.215
qurt	qurt_fabs	< 0.001	0.007
	qurt_sqrt	0.004	0.138
	qurt	0.002	0.066
select	select	0.042	0.730
ud	ludcmp	0.050	0.425
cnt_fdct	fdct	0.070	0.941
cnt_fft1	my_fabs	0.001	< 0.001
	my_sin	0.004	0.001
	my_cos	< 0.001	< 0.001
	my_log	< 0.001	< 0.001
	fft1	1.750	0.661
cnt_jfdctint	jpeg_fdct_islow	0.035	0.500
cnt_ns	foo	0.026	0.382
cnt_qurt	qurt_fabs	0.001	0.010
	qurt_sqrt	0.019	0.308
	qurt	0.003	0.047
cnt_minver	mmul	0.126	0.037
	minver_fabs	< 0.001	< 0.001
	minver	2.925	0.854

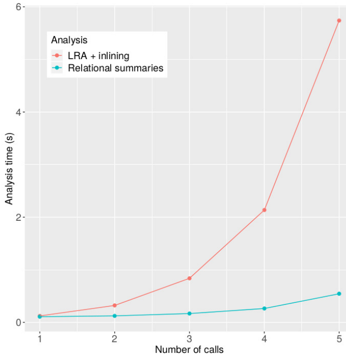
Table 2: Summaries computation times.

disjunction, enabling a more accurate analysis of procedure behaviors. Moreover, those precision improvements are not necessarily obtained at the expense of analysis time, since the **janne_complex** program has a more precise convex polyhedra at the exit of the main procedure, with a 60% increase in the number of constraints and has also the highest speedup with $S = 15.34$.

Table 2 reports the computation times of the summary of each procedure in each program. The τ_c column gives the fraction of the analysis time using our method spent during the computation of each procedure summary, defined as τ_c = Procedure summary comp. time/Program analysis time using rel. summ.

The summary construction time for small utility procedures, such as the **my_fabs**, **my_sin**, **my_cos** and **my_log** procedures, in the **fft1** and **cnt_fft1** programs, are very small (lower than 4ms) and often individually negligible with respect to the analysis time of the entire program (with τ_c often lower than 1%). This suggests that our method could be particularly beneficial, in terms of analysis performance, for programs built on top of a collection of utility procedures or a library of such procedures, each procedure summary being computed only once and possibly used in many call contexts.

Our last experiment concerns the speedup of our interprocedural analysis with respect to the number of calls. Notice that the Mälardalen benchmark is not very favorable in this respect, since most procedures are called only once. Our analysis on the **cnt_ns** program has a moderate speedup of 1.25. In order to observe the evolution of the speedup with the number of calls, we increase the number of calls to the *foo* procedure in the main procedure of the **cnt_ns** program. The opposite graphic shows the evolution of the analysis times of these successive versions, comparing our analysis with respect to standard LRA with inlining.



The analysis of the **cnt_ns** program using our disjunctive relational summaries analysis becomes significantly faster than standard LRA with inlining when there are more than 2 calls to the *foo* procedure in the main procedure.

11 Conclusion and Future Work

In this paper, we proposed a method for interprocedural analysis as a solution to the cost of using expressive relational abstract domains in program analysis. An analysis using a relational domain can be straightforwardly transformed into a relational analysis, computing an input-output relation. Such relations can be used as procedure summaries, computed once and for all, and used bottom-up to compute the effect of procedure calls. Applying this idea with linear relation analysis, we concluded that the obtained polyhedral summaries are not precise enough, and deserve to be refined disjunctively. The main ideas of the paper are as follows. First, we used precondition partitioning as a basis of disjunctive summaries. Then, we proposed a heuristic method for refining a summary according to reachability of control points or calling contexts of called procedures. We also identified some technical improvements, like widening limited by preconditions and previously computed relations, and more precise computation of results at loop exit points. Our experiments show that using summaries built in this way can significantly reduce the analysis time, especially for procedures used several times. On the other hand, the precision of the results is not dramatically damaged, and can even be improved, due to disjunctive analysis.

Future work should be devoted to applying the method with other relational domains. In particular, octagons [39] would be interesting since they permit a better quantitative comparison of results: apart from infinite bounds, two octagons on the same variables can be precisely compared by comparing their constant vectors. Another, longer-term, perspective is to use disjunctive relational summaries for procedures acting on remanent memories, like methods in object-oriented programming or reaction functions in reactive programming. Our precondition partitioning could result in partitioning memory states, and allow disjunctive memory invariants to be constructed modularly.

References

1. Allen, F.E.: Interprocedural analysis and the information derived by it. In: IBM Germany Scientific Symposium Series. pp. 291–321. Springer (1974)
2. Allen, F.E.: Interprocedural data flow analysis. In: IFIP Congress. pp. 398–402 (1974)
3. Ancourt, C., Coelho, F., Irigoin, F.: A modular static analysis approach to affine loop invariants detection. Electronic Notes in Theoretical Computer Science **267**(1), 3–16 (2010)
4. Apinis, K., Seidl, H., Vojdani, V.: How to combine widening and narrowing for non-monotonic systems of equations. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13. pp. 377–386. Seattle, WA (Jun 2013)
5. Bagnara, R., Ricci, E., Zaffanella, E., Hill, P.M.: Possibly not closed convex polyhedra and the Parma Polyhedra Library. In: Hermenegildo, M.V., Puebla, G. (eds.) 9th International Symposium on Static Analysis, SAS'02. LNCS 2477, Madrid, Spain (Sep 2002)
6. Barth, J.M.: An interprocedural data flow analysis algorithm. In: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. pp. 119–131. ACM (1977)
7. Barvinok, A.I.: A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. Math. Oper. Res. **19**(4), 769–779 (1994). https://doi.org/10.1287/moor.19.4.769, https://doi.org/10.1287/moor.19.4.769
8. Becchi, A., Zaffanella, E.: An efficient abstract domain for not necessarily closed polyhedra. In: Static Analysis, SAS 2018 (Aug 2018)
9. Bourdoncle, F.: Abstract interpretation by dynamic partitioning. Journal of Functional Programming **2**(4), 407–435 (1992)
10. Boutonnet, R., Asavoae, M.: The WCET analysis using counters - a preliminary assessment. In: Proceedings of 8th JRWRTC, in conjunction with RTNS14. Versailles, France (Oct 2014)
11. Clauss, P.: Counting solutions to linear and nonlinear constraints through ehrhart polynomials: applications to analyze and transform scientific programs. In: Proceedings of the 10th international conference on Supercomputing, ICS 1996, Philadelphia, PA, USA, May 25-28, 1996. pp. 278–285 (1996). https://doi.org/10.1145/237578.237617, http://doi.acm.org/10.1145/237578.237617
12. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: 4th ACM Symposium on Principles of Programming Languages, POPL'77. Los Angeles (Jan 1977)
13. Cousot, P., Cousot, R.: Static determination of dynamic properties of recursive procedures. In: IFIP Conference on Formal Description of Programming Concepts, St. Andrews, NB, Canada. North-Holland Publishing Company (1977)
14. Cousot, P., Cousot, R.: Relational abstract interpretation of higher order functional programs (extended abstract). In: Actes JTASPEFL'91 (Bordeaux), October 1991, Laboratoire Bordelais de Recherche en Informatique (LaBRI), Proceedings. pp. 33–36 (1991)
15. Cousot, P., Cousot, R.: Compositional separate modular static analysis of programs by abstract interpretation. In: Proc. SSGRR. pp. 6–10 (2001)

16. Cousot, P., Cousot, R.: Modular static program analysis. CC **2**, 159–178 (2002)
17. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. pp. 84–96. ACM (1978)
18. Flexeder, A., Müller-Olm, M., Petter, M., Seidl, H.: Fast interprocedural linear two-variable equalities. ACM Transactions on Programming Languages and Systems (TOPLAS) **33**(6), 21 (2011)
19. Giacobazzi, R., Scozzari, F.: A logical model for relational abstract domains. ACM Transactions on Programming Languages and Systems (TOPLAS) **20**(5), 1067–1109 (1998)
20. Gulwani, S., Tiwari, A.: Computing procedure summaries for interprocedural analysis. Programming Languages and Systems pp. 253–267 (2007)
21. Halbwachs, N.: Détermination automatique de relations linéaires vérifiées par les variables d'un programme. Ph.D. thesis, Université Scientifique et Médicale de Grenoble (1979)
22. Howe, J.M., King, A.: Polyhedral analysis using parametric objectives. In: International Static Analysis Symposium. pp. 41–57. Springer (2012)
23. Irigoin, F., Jouvelot, P., Triolet, R.: Semantical interprocedural parallelization: An overview of the pips project. In: ACM International Conference on Supercomputing 25th Anniversary Volume. pp. 143–150. ACM (2014)
24. Jeannet, B.: Dynamic partitioning in linear relation analysis: Application to the verification of reactive systems. Formal Methods in System Design **23**(1), 5–37 (2003)
25. Jeannet, B.: Interproc analyzer for recursive programs with numerical variables. INRIA, software and documentation are available at the following URL: http://pop-art. inrialpes. fr/interproc/interprocweb. cgi. Last accessed pp. 06–11 (2010)
26. Jeannet, B.: Relational interprocedural verification of concurrent programs. Software & Systems Modeling **12**(2), 285–306 (2013)
27. Jeannet, B., Gopan, D., Reps, T.: A relational abstraction for functions. In: SAS. vol. 3672, pp. 186–202. Springer (2005)
28. Jeannet, B., Halbwachs, N., Raymond, P.: Dynamic partitioning in analyses of numerical properties. In: International Static Analysis Symposium. pp. 39–50. Springer (1999)
29. Jeannet, B., Miné, A.: Apron: A library of numerical abstract domains for static analysis. In: Computer Aided Verification. pp. 661–667. Springer (2009)
30. Jeannet, B., Serwe, W.: Abstracting call-stacks for interprocedural verification of imperative programs. In: AMAST. pp. 258–273. Springer (2004)
31. Kelly, W., Maslov, V., Pugh, W., Rosser, E., Shpeisman, T., Wonnacott, D.: The Omega calculator and library, version 1.1. 0. College Park, MD **20742**, 18 (1996)
32. Khedker, U., Sanyal, A., Sathe, B.: Data flow analysis: theory and practice. CRC Press (2009)
33. Kranz, J., Simon, A.: Modular analysis of executables using on-demand heyting completion. In: International Conference on Verification, Model Checking, and Abstract Interpretation. pp. 291–312. Springer (2018)
34. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04). Palo Alto, California (Mar 2004)
35. Maisonneuve, V.: Convex invariant refinement by control node splitting: a heuristic approach. Electronic Notes in Theoretical Computer Science **288**, 49–59 (2012)

36. Maisonneuve, V., Hermant, O., Irigoin, F.: Computing invariants with transformers: experimental scalability and accuracy. Electronic Notes in Theoretical Computer Science **307**, 17–31 (2014)
37. Maréchal, A., Monniaux, D., Périn, M.: Scalable minimizing-operators on polyhedra via parametric linear programming. In: International Static Analysis Symposium. pp. 212–231. Springer (2017)
38. Mauborgne, L., Rival, X.: Trace partitioning in abstract interpretation based static analyzers. In: European Symposium on Programming. pp. 5–20. Springer (2005)
39. Miné, A.: The octagon abstract domain. In: AST 2001 in WCRE 2001. pp. 310–319. IEEE, IEEE CS Press (October 2001)
40. Müller-Olm, M., Rüthing, O., Seidl, H.: Checking Herbrand equalities and beyond. In: VMCAI. vol. 5, pp. 79–96. Springer (2005)
41. Müller-Olm, M., Seidl, H.: Computing interprocedurally valid relations in affine programs. Princ. of Prog. Lang (2004)
42. Müller-Olm, M., Seidl, H., Steffen, B.: Interprocedural analysis (almost) for free. Dekanat Informatik, Univ. (2004)
43. Popeea, C., Chin, W.N.: Inferring disjunctive postconditions. In: Annual Asian Computing Science Conference. pp. 331–345. Springer (2006)
44. Popeea, C., Chin, W.N.: Dual analysis for proving safety and finding bugs. In: Proceedings of the 2010 ACM Symposium on Applied Computing. pp. 2137–2143. ACM (2010)
45. Popeea, C., Chin, W.N.: Dual analysis for proving safety and finding bugs. Science of Computer Programming **78**(4), 390–411 (2013)
46. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 49–61. ACM (1995)
47. Rival, X., Mauborgne, L.: The trace partitioning abstract domain. ACM Transactions on Programming Languages and Systems (TOPLAS) **29**(5), 26 (2007)
48. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. New York University. Courant Institute of Mathematical Sciences. Computer Science Department (1978)
49. Singh, G., Püschel, M., Vechev, M.T.: Fast polyhedra abstract domain. In: POPL. pp. 46–59 (2017)
50. Sotin, P., Jeannet, B.: Precise interprocedural analysis in the presence of pointers to the stack. In: European Symposium on Programming. pp. 459–479. Springer (2011)
51. Spillman, T.C.: Exposing side-effects in a PL/I optimizing compiler. In: IFIP Congress (1). pp. 376–381 (1971)
52. Yorsh, G., Yahav, E., Chandra, S.: Generating precise and concise procedure summaries. In: ACM SIGPLAN Notices. vol. 43, pp. 221–234. ACM (2008)
53. Zhang, X., Mangal, R., Naik, M., Yang, H.: Hybrid top-down and bottom-up interprocedural analysis. In: ACM SIGPLAN Notices. vol. 49, pp. 249–258. ACM (2014)