ROBUST AND GENERIC ABSTRACT DOMAIN FOR STATIC PROGRAM ANALYSES:

THE POLYHEDRAL CASE

Duong NGUYEN QUE

July 13, 2006

Contents

Chapter 1

Introduction

Abstraction lies at the heart of computer science. To write a loop is to abstract a sequence of operations. To write a subroutine is to abstract what is common in sequences of operations in different places in a program. To define a type is to abstract what is common among all instances of the type, while the type's member data define what differentiates individual instances. Classes and inheritance, templates and type classes, functional and logic programming, are also just more sophisticated abstractions. As users expect increasing sophistication in program behavior and as programs accordingly become more complex, the need for improved abstractions will continue to grow.

Thus the history of computer programming has tended toward more sophisticated forms of abstraction, and promises to continue to do so. These abstractions, however, often come at a price: slower execution and greater memory consumption. Unfortunately, this conflicts with the broad principle in programming language design that *you should not have to pay for a feature you do not use*. Smart compiler optimizations, however, can often avoid these disadvantages, if they can determine how individual uses of an abstraction will behave. This is where static analysis comes in. It is the function of a static analyzer (henceforth just called an analyzer) to determine things about the way a program will behave that are not directly specified by the programmer.

The abstract interpretation introduced by the Cousots [CC77, Cou97, CC00] provides a solid theoretical foundation for static analysis. It is a theory of sound approximation of the semantics of computer programs, based on monotonic functions over ordered sets, especially lattices, called abstract domain. Its main concrete application is formal static analysis, the automatic extraction of information about the possible executions of computer programs. Such analyses have two main usages in analyzers: to analyze programs in order to decide whether certain optimizations or transformations are applicable; for debugging or even the certification of programs against classes of bugs.

Abstract domain libraries used in current analyzers are dealing with problems limiting the effectiveness of checking statically safety and security properties of programs written in different languages, and identifying and locating origins of failures. Approximate analyses must be used to overcome computing limit, especially when dealing with large scale application.

One of the two main goals of this dissertation is to help designing a common interface for abstract domain libraries used in five static analyzers PIPS [PIPb, IJT91], NBAC [NBA, JEA00], ASTREE [AST, BCC+03], the OMEGA framework [Ome, PUG91] and CHINA [CHI, PAR, BRZaH02].

The other goal is to provide a case study with the polyhedra-based analyzers: benchmarking polyhedral implementations used in these analyzers.

For the first goal, our starting point is to analyze problems existing in a particular analyzer named PIPS [PIPb, IJT91] when using abstract domain libraries, and then extend to other analyzers such as ASTREE [AST, BCC+03] and NBAC [NBA, JEA00].

Each static analyzer has its own library dealing with its own abstract domain(s). In practice, these libraries seem to have problems when dealing with large scale applications. However, recent developments from one team such as new abstract domains, e.g. the Octagon library [MIN, MIN01b], or algorithmic improvements, e.g. Cartesian factorization [HMPV03], cannot be readily exploited by other teams. Meanwhile existing static analyzers are mostly modular, and some important modules are based on similar technologies. Each analyzer develops different techniques that are integrated in different modules.

How to profit all these abstract domain implementations, when they come with advantages and disadvantages at the same time? And did we have the best use of each and every abstract domain? While it is still hard to compare the effectiveness among different abstract domains, we can make comparisons possible for each abstract domain, to answer the second question.

For the second goal, the most used polyhedral domain counts several implementations with several algorithmic discoveries that make it very robust. These implementations are varying and complicated to the point that without a benchmarking system, we cannot determine which one is the most efficient. Benchmarking helps deciding when and where to use which (appropriate) implementations. It also helps regression testing, bug detection, performance and stability evaluations, etc.

Our work is divided into two parts. The first part deals with an adaptive abstract domain, which in fact leads to the construction of a common interface. This interface tries to combine and use efficiently existing abstract domains implementations. The second part describes a framework permitting evaluations of equivalent implementations for the polyhedral domain. This framework later can be used with other abstract domains. We notice here that these two goals are new and original although they are both quite important.

A proposition of a common interface for already functional implementations, which are used in different projects, must prove itself worthy. Seeing that abstract domain community with strongly related groups is ignoring the possibility of sharing important progressions, it is of our strong feeling that someone must start this work. Even if we could not solve all the problems, we would be able to help understanding these problems.

As described in Chapter 4, a French project named APRON was launched in 2004 where such a common interface is of interest. Although Omega [Ome, PUG91] and Parma [CHI, PAR, BRZaH02] teams are not members of the project, the common interface must be compatible with their interfaces.

In order to design an interface which can replace all interfaces already used in current implementations, first of all we need to study these interfaces and to find out common points and incompatibilities among them. Incompatibilities may happen at the interface level, that is to say the signatures of operators, the data structures, etc., as well as at the implementation level, with exception management, thread-safety features, underlying arithmetics, etc.

Our benchmark system is quite different to existing benchmarks. Besides the performance, the stability (ability to cope with computational problems) analyses, we integrate some polyhedral characteristics into our evaluations. Since our set of tests is generated by a static analyzer (PIPS [PIPb, IJT91]) using standard benchmarks with large size applications (PerfectClub and SPEC95 benchmarks), the results provide richer information than previous experimental work. Moreover, new comparisons of implementations that were not considered before are presented.

Indeed, pre-existing evaluations are not satisfying for several reasons: our bibliographic study has revealed the fact that, it doesn't exist yet a mechanism to evaluate effectively these works, especially in the domain of program analysis and transformation with real-life examples. Conducted evaluations are based on at most one hundred problems, mostly theoretical, without analyses on quantity, on criteria, on exceptions (cases where algorithms fail because of resource limits), etc...

Apart from clarifying whether CPU or memory efficiency or both are the intended measures of interest, we offer problem-related analyses (such as polyhedral size parameters, their origin), stability comparisons, incoherent results checking, precision of computing comparisons, etc... Those features are not available elsewhere.

Since our work is in the context of static program analysis, Chapter 2 summarizes important concepts in static program analysis, introduces some examples of program analysis, and discusses our motivation in this point of view. It begins with some basic mathematical concepts and some main approaches in static program analysis, among which the uses of abstract interpretation with abstract domains. Some examples of program analyses are discussed in order to give a view of what a program analysis can consist of, and to discuss problems that may occur in analysis.

In Chapter 3, we introduce some important numerical domains used in static program analysis, based on abstract interpretation. We present in this chapter some related projects such as PIPS [PIPb, IJT91], NBAC [NBA, JEA00] and ASTRÉE [AST, BCC+03], with their underlying domains, and discuss in great detail three of these projects covering most of important abstract domains. We are interested in differences and problems concerning the utilization of these abstract domains. Some important definitions of convex polyhedra, octagons and BDDs are introduced. Also in this chapter, existing implementations for these abstract domains are presented, in order to draw the picture of abstract domains used in actual analyzers.

Chapter 4 discusses the problems with the abstract domains presented in the previous chapter and with their libraries. The design of a common interface for those different domains and libraries, called *HQ* interface, is introduced. Also in this chapter, we briefly present a related project, APRON, to which we contributed our HQ proposition. The main decisions made for APRON are discussed. We conclude by comparing HQ and APRON while taking into account their different goals.

Chapter 3 and Chapter 4 deal with several abstract domains. Meanwhile Chapter 4 and Chapter 5 concern only the polyhedral domain. Chapter 4 focus on polyhedral algorithms, whose implementations are briefly covered in Chapter 3, and important practical issues. Our contribution to improve some of these operators is also presented in this chapter, including the decomposition of polyhedra by the inclusion test.

Chapter 5 presents our framework to analyze the performances of the mentioned implementations. Then it discusses our experimental results which are obtained for the most important operators: satisfiability test, dual conversion, projection, minimization and convex hull. Also in this chapter, some examples illustrate the impact of exceptions and the occurrence of magnitude overflows during PIPS execution in Section 1.2.

After the conclusion in which the perspectives are discussed, we add the Appendix page 189 that details some algorithms which are briefly described in Chapter 4.

Chapter 2

Static program analysis

This chapter summarizes concepts in static program analysis, and introduces some examples of analysis. It begins with some basic mathematical concepts and some main approaches in static program analysis. Some examples of analyses are discussed in order to give a view of what a program analysis can consist of, and to discuss problems that may occur in analysis. The conclusion part summarizes the chapter and discusses our motivation.

1 Context

Any compiler checks the syntax of a program written in a programming language, and then translates it into machine code, in order to execute the program. Thanks to program analysis, more advanced compilers can help programmers to see whether the written program will do correctly what they want it to do, or produce an equivalent but better program, by using the semantics of the program.

Dynamic program analysis proposes methods that analyze programs in their execution phase, whereas static program analysis studies the behaviors of programs without actually running it. Thus, an interpreter can perform static analyses as well as dynamic analyses. A compiler can perform static analyses only.

Static program analysis, based on properties of programs that are sometimes incomputable, offers techniques for predicting safe approximations of behavioral properties arising dynamically at runtime of programs. Information obtained from this analysis can be used to support compiler optimization, verification, program comprehension, debugging, and documentation. We should be able, for example, to determine every error that might occur in the execution of programs. Several approaches to static program analysis exist such as program flow analysis, denotation - or logical formulations - based approaches, abstract interpretation, but important similarities and connections among them have been found.

This chapter firstly summarizes well-known concepts in static program analysis that are used in the dissertation and secondly introduces some examples of analyses with encountered problems. It begins with some basic mathematical concepts, namely, lattices, functions and fixed points (Section 2). Some main approaches in static program analysis such as program control and data flow analyses as well as interprocedural analysis are briefly described in the second section (Section 3). The semantic foundations of programs with the abstract interpretation methodology are also presented in this section.

Some examples of analyses are discussed in the next section (Section 4), in order to give a first view of what a program analysis can consist of, and to discuss problems that may occur in analysis. The conclusion part summarizes the chapter and discusses our motivation.

2 Basic Concepts (POS, lattices, chains)

This section recalls some definitions and theorems on lattice theory, based on Birkhoff [Bir67], and the approximation of fixed points, based on Nielson and al. [NNH99].

2.1 Partially Ordered Set, Lattices and Chains

DEFINITION 2.1 A *partially ordered set (poset)* is a set in which a binary relation $x \leqq y$ is defined, which satisfies for all x, y, z the following conditions:

1. *Reflexive*: for all x, $x \leqq x$
2. *Anti-symmetry*: if $x \leqq y$ and $y \leqq x$, then $x = y$
3. *Transitivity*: if $x \leqq y$ and $y \leqq z$, then $x \leqq z$

An *upper bound* of a subset X of a poset P is an element $a \in P$ such that $x \leqq a$ for all $x \in X$. a is called the *least upper bound* of X if a is an upper bound of X and if for any upper bound b of X, $a \leqq b$. The notions of *lower bounds* and the *greatest lower bound* of X are defined dually.

DEFINITION 2.2 A *lattice* is a partially ordered set P any two of whose elements have a greatest lower bound denoted by $x \sqcap y$ (*meet* operator), and a least upper bound denoted by $x \sqcup y$ (*join* operator).

A lattice L is *complete* when each of its subsets X has a least upper bound and a greatest lower bound in L. A non-void complete lattice L has a greatest lower bound denoted by $\perp$ (*least element*), and a least upper bound denoted by $\top$ (*greatest element*).

DEFINITION 2.3 A subset X of a partially ordered set $L(\leqq)$ is a *chain* if $\forall x_1, x_2 \in X : (x_1 \leqq x_2) \vee (x_2 \leqq x_1)$. A chain is a *finite chain* if it is a finite subset of L.

A sequence $x_0, x_1, \ldots, x_n, \ldots$ is an *ascending chain* if $n \leq m \Longrightarrow x_n \leqq x_m$. Similarly, a sequence $x_0, x_1, \ldots, x_n, \ldots$ is a *descending chain* if $n \leq m \Longrightarrow x_m \leqq x_n$.

We shall say that a sequence $x_0, x_1, \ldots, x_n, \ldots$ *eventually stabilizes* if and only if $\exists n_0 \in \mathbb{N} : \forall n \in \mathbb{N} : n \geq n_0 \Longrightarrow x_n = x_{n_0}$.

The partially ordered set L satisfies the *ascending chain condition* if and only if all ascending chains eventually stabilize. Similarly, it satisfies the *descending chain condition* if and only if all descending chains eventually stabilize.

2.2 Fixed Points Theorems

DEFINITION 2.4 Let f be a function from a partially ordered set $L_1(\leqq_1)$ to a partially ordered set $L_2(\leqq_2)$.

- f is *surjective* if $\forall x_2 \in L_2 : \exists x_1 \in L_1 : f(x_1) = x_2$
- f is *injective* if $\forall x_1, x_1' \in L_1 : f(x_1) = f(x_1') \Longrightarrow x_1 = x_1'$
- f is *monotone* if $\forall x_1, x_1' \in L_1 : x_1 \leqq_1 x_1' \Longrightarrow f(x_1) \leqq_2 f(x_1')$
- f is an *isomorphism* if f is monotone and there exists an unique monotone function $f^{-1} : L_2 \longrightarrow L_1$ such that $f \circ f^{-1} = id_2$ and $f^{-1} \circ f = id_1$ (where id_i is the identity function over $L_i, i = 1, 2$)

THEOREM 2.1 (Tarski) Let L be a complete lattice and f a monotone function on L into L: $f : L \longrightarrow L$. Then $f(a) = a$ for some $a \in L$.

An element $a \in L$ is called a *fixed point* of f if $f(a) = a$ and the set of fixed points of f is denoted by:

$$Fix(f) = \{a \in L : f(a) = a\}$$

Tarski's Fixed Point Theorem states that the set of fixed points of a monotone function f on a complete lattice $L(\leqq, \sqcup, \sqcap, \bot, \top)$ is a nonempty complete lattice with the partial ordering $\leqq$.

DEFINITION 2.5 The function $f : L \longrightarrow L$ is *reductive* at $x \in L$ if and only if $f(x) \leqq x$ and the set of elements upon which f is reductive is written:

$$Red(f) = \{x \in L : f(x) \leqq x\}$$

Similarly, the function f is *extensive* at $x \in L$ if and only if $x \leqq f(x)$ and the set of elements upon which f is extensive is written:

$$Ext(f) = \{x \in L : x \leqq f(x)\}$$

Tarski's Theorem also ensures that the *least fixed point* of f, symbolized as $lfp(f)$, is the greatest lower bound of the set $Red(f)$. Dually the *greatest fixed point* of f, symbolized as $gfp(f)$, is the least upper bound of the set $Ext(f)$.

$$lfp(f) = \sqcap Fix(f) = \sqcap Red(f)$$

$$gfp(f) = \sqcup Fix(f) = \sqcup Ext(f)$$

These propositions correspond to the two following principles:

- *Recursion Induction Principle*:

$$\forall x \in L, (f(x) \leqq x) \Longrightarrow (lfp(f) \leqq x))$$

- *Dual Recursion Induction Principle*:

$$\forall x \in L, (x \leqq f(x)) \Longrightarrow (x \leqq gfp(f)))$$

3 Static Program Analysis Approaches

3.1 Program Flow Analysis

Program flow analysis, in a few words, is a method for describing what a program does to its data. For each possible control point in the program, flow analysis establishes a finite description of the set of data states that the program could have when actual execution goes through that point. The resulting description may be either exact or approximate, in the sense that it does not provide all information, but sound, in the sense that it must be correct.

In this section, we describe the control flow and data flow analyses that apply to a procedure, as well as an interprocedural analysis that deals with whole programs. These analyses are vital for doing correct optimization and verification of programs. The material in the next subsections is based on the book *Advanced Compiler Design and Implementation* of Muchnick [Muc97].

3.1.1 Control-Flow Analysis

As the name tells, the purpose of this analysis is to reveal the structure of control flow within each procedure. For each routine of the program, its control structure is represented by a rooted and directed graph $G = (N, E)$, where N is the set of nodes including the root and $E \subseteq N \times N$ is the set of edges. G is called the *control flow graph* of the subroutine.

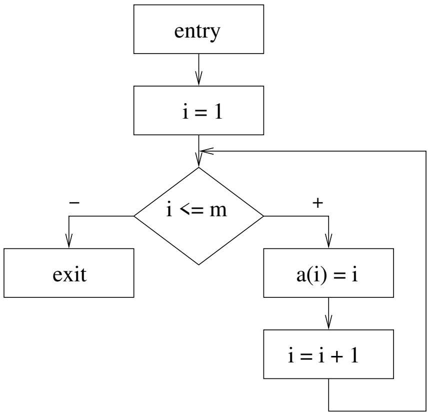
Basic components of the graph are defined as follows. A *basic block* is a straight-line sequence of statements that can be entered only at the beginning and exited only at the end. Set N contains a unique *entry* node, a unique *exit* node and the other basic blocks. A *branch node* is a node that has more than one successor and a *join node* is a node that has more than one predecessor. We say that node d *dominates* node n if every possible execution path from the entry node to n includes d. Dually, node p *post-dominates* node n if every possible execution path from n to the exit node includes p. A *back edge* in the graph is an edge $(m \rightarrow n)$ whose head n dominates its tail m.

Given a back edge $(m \rightarrow n)$, the *natural loop* of $(m \rightarrow n)$ is the subgraph consisting of the set of nodes containing n and all the nodes from which m can be reached in the graph without passing through n and the edge set connecting all the nodes in its node set. Node n is called the *loop header*. Unless two natural loops have the same header, they are either disjoint or one is nested within the other.

A *strongly connected component* of the graph is a subgraph $G_s = (N_s, E_s)$ such that every node in N_s is reachable from every other node by a path that includes only edges in E_s.

The graph $G = (N, E)$ is *reducible* (or *well-structured*) if and only if E can be partitioned into disjoint sets E_F, the *forward edge* set, and E_B, the back edge set, such that (N, E_F) forms a directed acyclic graph (DAG) in which each node can be reached from the entry node, and the edges in E_B are all back edges. *Irreducible* flow graphs (graphs that are not reducible) occur rarely in practice, but we must make sure that the control flow and data flow analyses can deal with them.

Figure 2.1 shows an example of a well-structured control flow graph.




```
I = 1
DO WHILE (I.LE.M)
    A(I) = I
ENDDO
```

Figure 2.1: Example of a control flow graph

There are two main approaches to control flow analysis of single routines, both of which start by determining the basic blocks that make up the routine and then constructing its flow graph. The first approach uses dominators to discover loops and simply notes the loops it finds for use in optimization. This approach is sufficient for iterative data flow analyzers and is the least time-consuming to implement.

The second approach, called *interval analysis*, includes a series of methods that analyze the overall structure of the routine and that decompose it into nested regions called intervals. They have widely been employed by optimizing compilers because they are fast, especially for structural analysis and programs that use only simple types of structures, and make it easier to update data flow information which has already been computed.

3.1.2 Interprocedural Analysis

Intraprocedural analyses are applied within a single procedure, without regard to the calling context in which that procedure is used or the procedures it calls. *Interprocedural analysis* refers to gathering information about the entire program instead of a single procedure. It studies the interactions among procedures in the whole program, including control-flow and data-flow.

Interprocedural Control-flow Analysis The problem addressed by interprocedural control-flow analysis is the construction of a program's *call graph*, which is a multi-graph with multi-directed edges from one node to another, defined as follows. Figure 2.2 is an example of a call graph.

Given a program P with procedures $p_1, p_2, \ldots, p_n$, the call graph of P is the graph $G = (N, S, E, p_1)$ with the node set $N = \{p_1, p_2, \ldots, p_n\}$, the set S of call site labels for different calls, the set $E \subseteq N \times S \times N$ of labeled edges, and the distinguished entry node for the main program $p_1 \in N$. For each e in E with $e = (p_i, s_k, p_j)$, s_k denotes a call site in p_i to p_j.

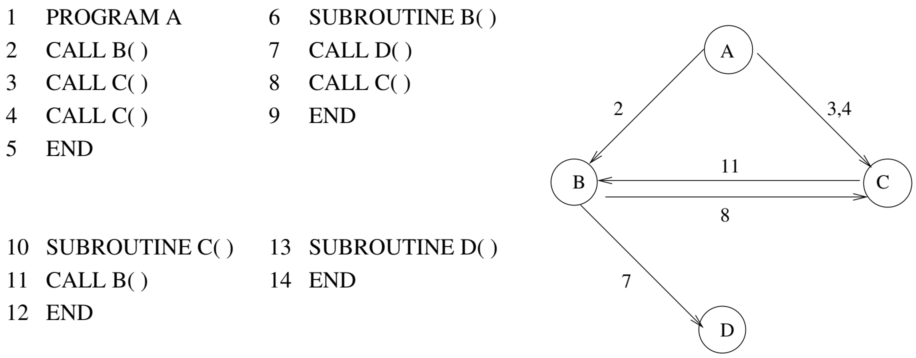


Figure 2.2: Example of a call graph

3.2 Abstract Interpretation

The *semantics* of a program is a computation model describing the behavior of a computer system executing this program. The *concrete semantics* of a syntactically correct program is a standard semantics that specifies exactly the effect of the program and the values it manipulates. In general, questions related to the concrete semantics of a non-trivial program are undecidable, in the sense that no computer can always answer them in finite time.

The Cousots in [CC77, Cou97, CC00] propose using an *abstract semantics*, which is a non-standard semantics that gives a conservative approximation of the concrete semantics. The abstraction does not allow to answer all questions about the program semantics but all answers given by the abstract semantics are always correct with respect to the concrete semantics. *Abstract interpretation* is a method of formalizing the approximation relation between the concrete semantics and the abstract semantics.

Each semantics must be based on a *semantic domain* which is a partially ordered set, or a stronger set, a complete lattice. To represent the relation between the *concrete semantic domains* and *abstract semantic domains*, we need the concept of Galois connection that is defined below.

3.2.1 Galois Connection

Let $L_1(\leqq_1, \sqcup_1, \sqcap_1, \perp_1, \top_1)$ and $L_2(\leqq_2, \sqcup_2, \sqcap_2, \perp_2, \top_2)$ be two complete lattices corresponding to the concrete and abstract semantic domains. To express the relationship between these two domains, we have an *abstraction function*

$$\alpha : L_1 \longrightarrow L_2$$

giving an abstract representation of the elements belonging to the concrete domain in the abstract domain, and a *concretization function*

$$\gamma : L_2 \longrightarrow L_1$$

that expresses the meaning of some abstract information in concrete terms.

DEFINITION 3.1 $(L_1, \alpha, \gamma, L_2)$ is a *Galois connection* between the complete lattices L_1 and L_2 if and only if $\alpha : L_1 \longrightarrow L_2$ and $\gamma : L_2 \longrightarrow L_1$ are monotone functions that satisfy:

$$\lambda l_1.l_1 \leqq_1 \gamma \circ \alpha \text{ and } \alpha \circ \gamma \leqq_2 \lambda l_2.l_2$$

The partial orderings $\leqq_1$ and $\leqq_2$ formalize the loss of information. $l_1 \leqq_1 \gamma(\alpha(l_1))$ means that the concretization of the abstract representation of l_1 is a safe approximation of l_1. The above conditions express that we do not lose safety by going back and forth between the two domains although we may lose precision.

A program analysis can be developed in several steps of refinement, by using different combinations of Galois connections such as sequential composition, parallel combinations, etc. Galois connections are indeed useful for transforming computations into more approximate computations that have better time, space or termination behavior and whose correctness properties are still preserved.

We can see that program flow analysis and abstract interpretation have many common points. For instance, in order to remain computable, both sometimes give approximate answers. These answers do not provide precise information but may give useful information. In general, the program analysis produces a possibly larger set of possibilities than what will ever happen during execution of the program. In addition, all program analyses should be semantics based, which means that the information obtained from the analysis can be proven to be correct with respect to the semantics of the programming language.

3.2.2 Widening/Narrowing Approach

Another method [CC76, CC77, CC79] for enforcing termination of the abstract interpretation consists in using a *widening* operator.

DEFINITION 3.2 An operator $\nabla \in L \times L \longmapsto L$ is a *widening* if:

$$\forall x, y \in L : x \sqsubseteq (x \nabla y)$$
$$\forall x, y \in L : y \sqsubseteq (x \nabla y)$$

for all increasing chains $x^0 \sqsubseteq x^1 \sqsubseteq \ldots$, the increasing chain defined by $y^0 = x^0, \ldots, y^{i+1} = (y^i \nabla x^{i+1})$ is not strictly increasing.

Approximation using widening operator permits the termination of the abstract interpretation, whereas using *narrowing* operator permits improvements of this approximation.

DEFINITION 3.3 An operator $\Delta \in L \times L \longmapsto L$ is a *narrowing* if:

$$\forall x, y \in L : (y \sqsubseteq x) \implies (y \sqsubseteq (x \Delta y) \sqsubseteq x)$$

for all decreasing chains $x^0 \sqsupseteq x^1 \sqsupseteq \ldots$, the decreasing chain defined by $y^0 = x^0, \ldots, y^{i+1} = (y^i \Delta x^{i+1})$ is not strictly decreasing.

In [CC91], it is shown that, in general, using infinite abstract domains with widenings and narrowings is more powerful and more precise than using finite lattices to ensure the termination of abstract interpretation of programs on infinite lattices.

4 Examples of Analyses

In this section we present three important analyses implemented in our tool PIPS ([PIPb]). To illustrate the analyses, we extract a small fragment of code (Figure 2.3) from the program ocean.f in Perfect Club benchmark [BCK+89], then analyze it with PIPS. In the following subsections we briefly define the analyses and expose problems related to them.

4.1 A Simplified Language

Since the syntax of real-life programming languages is complex, we need to introduce an abstract language which is a simplified version of them, based on which the analyses are

```
SUBROUTINE SHUF (A,N2P,N1,WORK)
DIMENSION A(N2P,1),WORK(1)
DO J = 1, N1, 2
   DO I = 1, N2P
      II = I+I
      WORK(II-1) = A(I,J)
      WORK(II) = A(I,J+1)
   ENDDO
   DO I = 1, N2P
      A(I,J) = WORK(I)
   ENDDO
   DO I = 1, N2P
      A(I,J+1) = WORK(I+N2P)
   ENDDO
ENDDO
END
```

Figure 2.3: Example of code from ocean.f

defined. Figure 2.4 shows the syntax of FORTRAN toy language that we use in order to present our analyses.

The sign := denotes a definition, the sign [ ] an optional item, the sign | the choice production and the sign $*$ the list production. For example, an unary operator can be a negation ($-$) or a logical negation (.NOT.); $< statement >^*$ is a sequence of zero, one or more statements.

An executable program includes the main program, function and subroutine subprograms, characterized by their header. There is no recursion in the language. Specification statements are parameter, type, common, equivalence, dimension and data declarations. Executable statements consist of continue, stop, read, write, assignment, conditional statement, loop, call to external subroutine and sequence of statements. An expression is either a constant, a reference to a variable, an unary or a binary expression. A variable reference can be a scalar variable (We do not consider arrays for the shake of simplicity). The syntactic domains of this language include:

Variable	: set of variables
Expression	: set of expressions
Statement	: set of statements

The semantic definition of a programming language characterizes the effect of programs on the values they manipulate. To define the semantics of our language, we have the following semantic domains:

$$
\begin{array}{lll}
Procedure & := & head : Header\ decls : Declaration^{*}\ body : Statement\ \text{END} \\
Header & := & \text{PROGRAM}\ name : Name \\
 & | & \text{SUBROUTINE}\ name : Name\ [(formals : Variable^{*})] \\
Declaration & := & \text{PARAMETER}\ (var : Variable = cons : Constant) \\
 & | & \text{DIMENSION}\ var : Variable\ (dims : Dimension^{*}) \\
 & | & \text{EQUIVALENCE}\ (var_1 : Variable, var_2 : Variable) \\
 & | & \text{COMMON}\ [/com : Name/]\ vars : Variable^{*} \\
 & | & type : Type\ var : Variable\ [(dims : Dimension^{*})] \\
 & | & \text{DATA}\ var : Variable\ /cons : Constant/ \\
Type & := & \text{INTEGER} \mid \text{REAL} \mid \text{DOUBLE PRECISION} \mid \\
 & & \text{COMPLEX} \mid \text{LOGICAL} \mid \text{CHARACTER} \\
Dimension & := & lower : Expression : upper : Expression \\
Statement & := & \text{CONTINUE} \\
 & | & \text{STOP} \\
 & | & \text{READ}\ ref : Reference \\
 & | & \text{WRITE}\ exp : Expression \\
 & | & ref : Reference = exp : Expression \\
 & | & \text{IF}\ (cond : Expression)\ \text{THEN}\ true : Statement \\
 & & \text{ELSE}\ false : Statement\ \text{ENDIF} \\
 & | & \text{DO}\ (cond : Expression)\ body : Statement\ \text{ENDDO} \\
 & | & \text{CALL}\ proc : Name\ [(actuals : Expression^{*})] \\
 & | & sequence : Statement^{*} \\
Expression & := & cons : Constant \\
 & | & ref : Reference \\
 & | & (exp : Expression) \\
 & | & unop : Unary\_op\ exp : Expression \\
 & | & exp_1 : Expression\ binop : Binary\_op\ exp_2 : Expression \\
Unary\_op & := & \text{-} \mid \text{.NOT.} \\
Binary\_op & := & + \mid - \mid * \mid / \mid ** \mid \text{.LT.} \mid \text{.LE.} \mid \text{.EQ.} \mid \text{.NE.} \mid \text{.GT.} \mid \text{.GE.} \mid \\
 & & \text{.AND.} \mid \text{.OR.} \mid \text{.EQV.} \mid \text{.NEQV.} \\
Reference & := & var : Variable\ [(exp : Expression^{*})]
\end{array}
$$

Figure 2.4: FORTRAN toy language syntax

$\mathbb{B}$	: set of boolean values
$\mathbb{N}$	: set of positive integers
$\mathbb{Z}$	: set of integers (positive, zero or negative)
$\mathbb{Q}$	: set of rational numbers
$\mathbb{R}$	: set of reals
Constant	: set of constants
Value	: set of values
State	: set of states

The set of possible states of a program is defined as the set of functions from the set of variables to the set of values. In other words, the state of a program at any instant of its execution is a function that associates each variable to its value at that instant.

Each semantic domain is provided with a partial ordering $\leqq$, a least element $\perp$ and a greatest element $\top$ when needed. The partial ordering is obvious for above semantic domains, except the set of states. For the set of states *State*, we define a partial ordering $\leqq_{State}$, for all $\sigma_1, \sigma_2 \in State$:

$$\sigma_1 \leqq_{State} \sigma_2 \Longleftrightarrow dom(\sigma_1) \subseteq dom(\sigma_2) \wedge \forall v \in dom(\sigma_1) : \sigma_1(v) \leqq_{Value} \sigma_2(v)$$

and the least and greatest elements:

$$\begin{array}{llrll}
\perp_{State} \in State, & \perp_{State}: & Variable & \longrightarrow & Value \\
 & & dom(\perp_{State}) & \longmapsto & \perp_{Value} \\
\top_{State} \in State, & \top_{State}: & Variable & \longrightarrow & Value \\
 & & var & \longmapsto & \top_{Value} \\
 & & dom(\top_{State}) & = & Variable
\end{array}$$

where $\leqq_{Value}$ is the partial ordering and $\top_{Value}$ the greatest element of the *Value* domain.

4.2 Transformers

A statement is a state-to-state *transformer*, which means when a statement is executed in a certain state, it creates a new state upon termination. The semantic meaning of transformers is to model two directions of data-flow analysis: transformer and inverse transformer. The transformer represents the transformation of an input state to an output state, which is the result of the execution of the concerned statement. The *inverse transformer* corresponds to the transformation from the output to the input state.

Transformers as well as other analyses are based on the semantic function $\mathcal{E}$ that represents the evaluation of expressions. This evaluation yields a value that depends on the current state. The denotation of an expression is semantically a function from the set of states to the set of values, defined as follows:

$$\mathcal{E} : \textit{Expression} \longrightarrow \textit{State} \longrightarrow \textit{Value}$$

The transformer, denoted by $\mathcal{T}$, is a function from the set of states to itself:

$$\mathcal{T} : \textit{Statement} \longrightarrow \textit{State} \longrightarrow \textit{State}$$

Let $e \in \textit{Expression}, \sigma \in \textit{State}$, we have:

$$\begin{array}{ll}
\mathcal{E}(e) := \lambda\sigma.\ \text{Case } e \text{ of:} & \\
Constant & \longrightarrow e \\
Reference & \longrightarrow \sigma(e) \\
(exp) & \longrightarrow \mathcal{E}(exp)(\sigma) \\
unary_op\ (exp) & \longrightarrow \Theta[unary_op](\mathcal{E}(exp)(\sigma)) \\
(exp_1)\ binary_op\ (exp_2) & \longrightarrow \Delta[binary_op](\mathcal{E}(exp_1)(\sigma), \mathcal{E}(exp_2)(\sigma))
\end{array}$$

where Θ and Δ are some predefined functions.

Let $s \in \textit{Statement}$ and $\sigma, \top \in \textit{State}$. $\uplus$ is the *overriding union* operator [Mey90] where $\sigma \uplus \{(var, val)\}$ returns a state identical to σ, except that the value of var becomes val. Definition of the transformer is simplifed as follow:

$$\begin{array}{ll}
\mathcal{T}(s) := \lambda\sigma.\ \text{If } \sigma = \top \text{ then } \top & \text{else case } s \text{ of:} \\
\text{CONTINUE} & \longrightarrow \sigma \\
\text{STOP} & \longrightarrow \top \\
\text{READ } var & \longrightarrow \sigma \uplus \{(var, stdin)\} \\
\text{WRITE } exp & \longrightarrow \sigma \\
var = exp & \longrightarrow \sigma \uplus \{(var, \mathcal{E}(exp)(\sigma))\} \\
\text{IF } e \text{ THEN } s_1 \text{ ELSE } s_2 \text{ ENDIF} & \longrightarrow \text{if } \mathcal{E}(e)(\sigma) \text{ then } \mathcal{T}(s_1)(\sigma) \text{ else } \mathcal{T}(s_2)(\sigma) \\
\text{DO } e\ s_1 \text{ ENDDO} & \longrightarrow \text{if } \mathcal{E}(e)(\sigma) \text{ then } \mathcal{T}(s)(\mathcal{T}(s_1)(\sigma)) \text{ else } \sigma \\
\text{CALL } p(e_1, \ldots, e_n) & \longrightarrow \mathcal{T}(p.body)(\sigma \uplus \{(p.f_1, \mathcal{E}(e_1)(\sigma)), \ldots, (p.f_n, \mathcal{E}(e_n)(\sigma))\}) \\
s_1; s_2; \ldots; s_n; & \longrightarrow \mathcal{T}(s_2; \ldots; s_n;)(\mathcal{T}(s_1)(\sigma)))
\end{array}$$

Transformers are computed from elementary statements such as continue, stop, read, write and assignments to compound statements such as conditional statements, loops and sequences of statements. They are also analyzed inter-procedurally through procedure calls with the no side effects assumption.

The $CONTINUE$ and $WRITE$ statements do not change the program state. The *STOP* statement causes the termination of the whole program, and results in the $\top$ state. The *READ var* statement gives *var* the value from the standard input. The $CALL$ statement yields a program state which is the output state of the transformer of the called procedure body. The input state of this transformer is the initial state σ, except that the value of each formal parameter is associated to the value of the corresponding actual parameter.

In fact, the set of state *State* : *Exp* is abstracted by the set of polyhedra. In Figure 2.5, calculation of Transformer T is given at each step of the code. Note here that constraints are used to represent semantic meaning of transformers, where comes the term *polyhedral representation* because any set of constraints can be understood as a polyhedron ([CH78, HAL79]).

```
C  T() {}
      SUBROUTINE SHUF (A,N2P,N1,WORK)
      DIMENSION A(N2P,1),WORK(1)
C  T(I,II,J) {}
      DO J = 1, N1, 2                                                  0001
C  T(I,II) {}
         DO I = 1, N2P                                                 0002
C  T(II) {2I==II}
            II = I+I                                                   0003
C  T() {}
            WORK(II-1) = A(I,J)                                        0004
C  T() {}
            WORK(II) = A(I,J+1)                                        0005
         ENDDO
C  T(I) {}
         DO I = 1, N2P                                                 0006
C  T() {}
            A(I,J) = WORK(I)                                           0007
         ENDDO
C  T(I) {}
         DO I = 1, N2P                                                 0008
C  T() {}
            A(I,J+1) = WORK(I+N2P)                                     0009
         ENDDO
      ENDDO
      END
```

Figure 2.5: Transformer analysis example

Transformers to be handled at every step of program analysis can grow up in size quite fast. The above semantic functions for transformers are usually not computable in finite time, or not machine representable, except for very simple programs. Non trivial programs containing loops and having an infinite set of states could lead to the undecidable termination problem. Furthermore, the value of variables is not always known at compile-time. In addition, describing the exact behavior of any program written in any language may be impossible. Therefore in order to analyze complex programs, calculation of transformers is sometimes approximated.

Approximations of Transformers Approximated transformers are computed for scalar integer variables and represented as systems of equalities and inequalities. They are propagated from the module entry point down to the abstract syntax tree leaves.

Since the exact transformer contains only one program state, its under-approximation gives either the same result as the exact one or the least element $\perp$ of *State*, which is always safe but not interesting.

$$\begin{aligned} \underline{\mathcal{T}} : \mathit{Statement} \longrightarrow \quad & \mathit{State} \longrightarrow \quad \mathit{State} \\ & \underline{\mathcal{T}}(s)(\sigma) \longmapsto \quad \mathcal{T}(s)(\sigma) \text{ or } \perp \end{aligned}$$

So we are only interested in the over-approximation of transformers that computes a set of states containing the exact state:

$$\begin{aligned} \overline{\mathcal{T}} : \mathit{Statement} \longrightarrow \quad & \mathit{State} \longrightarrow \quad \wp(\mathit{State}) \\ & \mathcal{T}(s)(\sigma) \in \quad \overline{\mathcal{T}}(s)(\sigma) \end{aligned}$$

Theoretically, this over-approximation can be built from the definition of the exact transformer by using approximate operators. Then an over-approximation of the inverse transformer can be defined as:

$$\begin{aligned} \overline{\mathcal{T}}^{-1} : \quad \mathit{Statement} \quad & \longrightarrow \mathit{State} \longrightarrow \wp(\mathit{State}) \\ \overline{\mathcal{T}}^{-1}(s)(\sigma) & \longmapsto \{\sigma_1 : \sigma \in \overline{\mathcal{T}}(s)(\sigma_1)\} \end{aligned}$$

In fact, a constructive definition of the over-approximated transformers depends on the representation choice. It is often impossible to represent every set of states $S \in \wp(\mathit{State})$, but only sets that share common properties.

Transformers are approximated by polyhedra and manipulated by libraries implementing operators such as the test of emptiness of a polyhedron, the projection along one dimension, or the convex hull of two polyhedra. When analysis becomes complex, polyhedra of very large size appear so that existing polyhedral libraries cannot handle them. Examples of these polyhedra are given in the experimental part, Chapter 5. Since we cannot handle this kind of problem, techniques for approximation are used; accuracy of analysis is then reduced.

Approximations reduce the physical size of transformers. However, transformers are forwarded and combined, thus new transformers of larger size appear. The following paragraph discusses possible reasons of transformer's size accumulation, and related operations.

Size expansion A simple example in which the transformer becomes larger is that when we model an assignment to a scalar variable with an affine right hand size expression that contains the same variable. Then we need to keep the initial input value and the final output value of the variable. A new variable has to be introduced. For example, to model $I = I + 1$ assignment, we have:

$$T[I = I + 1] = (I == I\#init + 1)\ \sigma[I\#init/I]$$

In another case, the transformer of a test statement *if* is in fact the approximate union of two polyhedra: the first represents the test condition, and the second represents the transformer of *then* part, or *else* part, depending on the test condition. In practice, the convex hull of two polyhedra is computed instead of union, and the generated polyhedron sometimes can be physically much larger.

The transformer for a sequence of statements is the combination of transformers of all the elementary transformers. In this combination, variables are renamed and then merged into the resulting transformer, whereas intermediate values are eliminated by projection. The size of the transformer becomes important, especially when interprocedural analysis is performed: transformer translation is needed, which adds equations that model links and bindings among parts of programs.

4.3 Preconditions

Precondition analyses try to discover the constraints holding among variables of a program at a control point and at entry point. For any statement, the precondition and the postcondition provide a condition that holds just before the statement execution for the former and just after for the latter.

Preconditions and Postconditions Preconditions and postconditions describe the set of program states reached just before or after the execution of statements and can be represented as functions from the set of statements to the power-set of *State*:

$$\mathcal{P}re, \mathcal{P}ost : \mathit{Statement} \longrightarrow \wp(\mathit{State})$$

Let $s \in \mathit{Statement}$ and $\sigma \in \mathit{State}$:

$$\begin{aligned} \mathcal{P}re(s) &= \{\sigma : \sigma \in dom(\mathcal{T}(s))\} \\ \mathcal{P}ost(s) &= \{\sigma' : \sigma' = \mathcal{T}(s)(\sigma), \sigma \in \mathcal{P}re(s)\} \end{aligned}$$

According to these definitions, we say that the transformer of a statement is *applied* to the precondition of the statement to obtain the postcondition. This postcondition becomes the precondition of the following statement in a sequence. So the computation of the precondition (and the postcondition) is based on the transformer computation whose semantic function has already been described in Section 4.2.

Approximations of Preconditions Like transformers, we are interested in the over-approximation of preconditions:

$$\begin{aligned} \overline{\mathcal{P}}: \quad & \textit{Statement} \longrightarrow \wp(\textit{State}) \\ & \mathcal{P}(s) \subseteq \overline{\mathcal{P}}(s) \end{aligned}$$

Over-approximated preconditions are computed for scalar integer variables and represented as systems of equalities and inequalities. They are propagated from the module entry point down to the abstract syntax tree leaves.

Size expansion In order to obtain the postcondition by combining the precondition and the transformer, variables of precondition and initial values of transformer are renamed to intermediate variables which are then eliminated. Just like the transformers, precondition and postcondition in test statements can be obtained by calculation of convex hull of two polyhedra.

In a sequence of statements, we repeat applying transformers to preconditions to obtain postconditions, then postconditions become preconditions of the next transformers. Thus the precondition at the end can be a very complicated combination. The complexity in computation for precondition is multiplied in interprocedural analysis.

Since preconditions and postconditions are combined with transformers, their calculation can be even more expensive. In some cases, an operation can take up to hours before running out of memory space. Figure 2.6 shows the precondition P of the example.

4.4 Array Regions

Array region analyses collect information about the way array elements that are defined and used by programs. Many studies concerning these analyses have been realized. Batrice CREUSILLET, in her dissertation, continued and developed more techniques that are implemented in our tool PIPS. These techniques are based on *convex array regions* [TFI86, CI95, Cre95, CI96].

4.4.1 Array Region Analyses

The region of an array A of n dimensions at a statement s is a function from the set of states to the power set of $\mathbb{Z}^n$:

$$\begin{aligned} \mathcal{R}: \textit{Statement} \longrightarrow \quad & \textit{State} \longrightarrow \wp(\mathbb{Z}^n) \\ \mathcal{R}(s) \longmapsto \quad & \lambda\sigma.(\{\phi = (\phi_1, \ldots, \phi_n) \in \mathbb{Z}^n : r(\phi, \sigma)\}) \end{aligned}$$

The semantic function $\mathcal{R}$ associates to each statement and to each program state a set of elements indexing the associated array, described by the *region parameters* $\phi = (\phi_1, \ldots, \phi_n)$. The variable ϕ_i represents the $i-th$ dimension parameter. r is the relationship existing between ϕ and the current program state σ.

```
C  P() {}
      SUBROUTINE SHUF (A,N2P,N1,WORK)
      DIMENSION A(N2P,1),WORK(1)
C  P() {}
      DO J = 1, N1, 2                                           0001
C  P(I,II,J) {1<=J, J<=N1}
         DO I = 1, N2P                                          0002
C  P(I,II,J) {1<=I, I<=N2P, 1<=J, J<=N1}
            II = I+I                                            0003
C  P(I,II,J) {2I==II, 1<=I, I<=N2P, 1<=J, J<=N1}
            WORK(II-1) = A(I,J)                                 0004
C  P(I,II,J) {2I==II, 1<=I, I<=N2P, 1<=J, J<=N1}
            WORK(II) = A(I,J+1)                                 0005
         ENDDO
C  P(I,II,J) {1<=I, N2P+1<=I, 1<=J, J<=N1}
         DO I = 1, N2P                                          0006
C  P(I,II,J) {1<=I, I<=N2P, 1<=J, J<=N1}
            A(I,J) = WORK(I)                                    0007
         ENDDO
C  P(I,II,J) {1<=I, N2P+1<=I, 1<=J, J<=N1}
         DO I = 1, N2P                                          0008
C  P(I,II,J) {1<=I, I<=N2P, 1<=J, J<=N1}
            A(I,J+1) = WORK(I+N2P)                              0009
         ENDDO
      ENDDO
      END
```

Figure 2.6: Precondition analysis example

Since array region analysis was introduced to support dependence analyses on array structures, two kinds of effects on array elements are used: `READ`, if they are used and `WRITE`, if they are defined. The difference between `READ` and `WRITE` regions is that `WRITE` regions are only defined on the domain of statements. Expressions also have read effects, so we have to define the semantic function of `READ` regions on the domain of expressions and that of statements:

$$\mathcal{R}_w : \mathit{Statement} \longrightarrow \mathit{State} \longrightarrow \wp(\mathbb{Z}^n)$$
$$\mathcal{R}_r : \mathit{Statement} \oplus \mathit{Expression} \longrightarrow \mathit{State} \longrightarrow \wp(\mathbb{Z}^n)$$

We compute regions for each array independently to simplify the representation. Array regions for all arrays in the program are the product of array regions for each array.

Suppose that A is the current array with declarations $A(l_1 : u_1, \ldots, l_n : u_n)$. Let $e \in \mathit{Expression}, s \in \mathit{Statement}$ and $\sigma \in \mathit{State}$. Then the semantic functions for `READ` and `WRITE` regions, $\mathcal{R}_r$ and $\mathcal{R}_w$ respectively, are defined as follows:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathcal{R}_r(e) := \lambda\sigma(\text{ Case } e \text{ of:}} \\
\mathit{Constant} & \longrightarrow \emptyset \\
\mathit{var} & \longrightarrow \emptyset \\
\mathit{var}(e_1, \ldots, e_n) & \longrightarrow \cup_{i=1}^{n} \mathcal{R}_r(e_i)(\sigma) \cup (\text{if } (\mathit{var} = A) \text{ then } \{(\phi_1, \ldots, \phi_n) : \phi_i = \mathcal{E}(e_i)(\sigma)\} \text{ else } \emptyset) \\
(\mathit{exp}) & \longrightarrow \mathcal{R}_r(\mathit{exp})(\sigma) \\
\mathit{unary\_op\ exp} & \longrightarrow \mathcal{R}_r(\mathit{exp})(\sigma) \\
\mathit{exp}_1\ \mathit{binary\_op}\ \mathit{exp}_2 & \longrightarrow \mathcal{R}_r(\mathit{exp}_1)(\sigma) \cup \mathcal{R}_r(\mathit{exp}_2)(\sigma))
\end{array}
$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathcal{R}_r(s) := \lambda\sigma(\text{ Case } s \text{ of:}} \\
\text{CONTINUE} & \longrightarrow \emptyset \\
\text{STOP} & \longrightarrow \emptyset \\
\text{READ } \mathit{var} & \longrightarrow \emptyset \\
\text{READ } \mathit{var}(e_1, \ldots, e_n) & \longrightarrow \cup_{i=1}^{n} \mathcal{R}_r(e_i)(\sigma) \\
\text{WRITE } \mathit{exp} & \longrightarrow \mathcal{R}_r(\mathit{exp})(\sigma) \\
\mathit{var} = \mathit{exp} & \longrightarrow \mathcal{R}_r(\mathit{exp})(\sigma) \\
\mathit{var}(e_1, \ldots, e_n) = \mathit{exp} & \longrightarrow \mathcal{R}_r(\mathit{exp})(\sigma) \cup (\cup_{i=1}^{n} \mathcal{R}_r(e_i)(\sigma)) \\
\text{IF } e \text{ THEN } s_1 \text{ ELSE } s_2 \text{ ENDIF} & \longrightarrow \mathcal{R}_r(e)(\sigma) \cup (\text{if } \mathcal{E}(e)(\sigma) \text{ then } \mathcal{R}_r(s_1)(\sigma) \text{ else } \mathcal{R}_r(s_2)(\sigma)) \\
\text{DO } e\ s_1 \text{ ENDDO} & \longrightarrow \mathcal{R}_r(e)(\sigma) \cup (\text{if } \mathcal{E}(e)(\sigma) \text{ then } \mathcal{R}_r(s_1)(\sigma) \cup \mathcal{R}_r(s)(\mathcal{T}(s_1)(\sigma)) \text{ else } \emptyset) \\
\text{CALL } p(e_1, \ldots, e_n) & \longrightarrow \mathcal{R}_r(p.\mathit{body})(\mathcal{T}(p.\mathit{body})(\sigma \uplus \{(p.f_1, \mathcal{E}(e_1)(\sigma)), \ldots, (p.f_n, \mathcal{E}(e_n)(\sigma))\})) \\
s_1; s_2; \ldots; s_n; & \longrightarrow \mathcal{R}_r(s_1)(\sigma) \cup \mathcal{R}_r(s_2; \ldots; s_n;)(\mathcal{T}(s_1)(\sigma)))
\end{array}
$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathcal{R}_w(s) := \lambda\sigma(\text{ Case } s \text{ of:}} \\
\text{CONTINUE} & \longrightarrow \emptyset \\
\text{STOP} & \longrightarrow \emptyset \\
\text{READ } A & \longrightarrow \{(\phi_1, \ldots, \phi_n) : \mathcal{E}(l_i)(\sigma) \le \phi_i \le \mathcal{E}(u_i)(\sigma), i = 1, \ldots, n\} \\
\text{READ } A(e_1, \ldots, e_n) & \longrightarrow \{(\phi_1, \ldots, \phi_n) : \phi_i = \mathcal{E}(e_i)(\sigma), i = 1, \ldots, n\} \\
\text{WRITE } \mathit{exp} & \longrightarrow \emptyset \\
A(e_1, \ldots, e_n) = \mathit{exp} & \longrightarrow \{(\phi_1, \ldots, \phi_n) : \phi_i = \mathcal{E}(e_i)(\sigma), i = 1, \ldots, n\} \\
\text{IF } e \text{ THEN } s_1 \text{ ELSE } s_2 \text{ ENDIF} & \longrightarrow \text{if } \mathcal{E}(e)(\sigma) \text{ then } \mathcal{R}_w(s_1)(\sigma) \text{ else } \mathcal{R}_w(s_2)(\sigma) \\
\text{DO } e\ s_1 \text{ ENDDO} & \longrightarrow \text{if } \mathcal{E}(e)(\sigma) \text{ then } \mathcal{R}_w(s_1)(\sigma) \cup \mathcal{R}_w(s)(\mathcal{T}(s_1)(\sigma)) \text{ else } \emptyset \\
\text{CALL } p(e_1, \ldots, e_n) & \longrightarrow \mathcal{R}_w(p.\mathit{body})(\mathcal{T}(p.\mathit{body})(\sigma \uplus \{(p.f_1, \mathcal{E}(e_1)(\sigma)), \ldots, (p.f_n, \mathcal{E}(e_n)(\sigma))\})) \\
s_1; s_2; \ldots; s_n; & \longrightarrow \mathcal{R}_w(s_1)(\sigma) \cup \mathcal{R}_w(s_2; \ldots; s_n;)(\mathcal{T}(s_1)(\sigma)))
\end{array}
$$

Note that unlike the evaluation of expression $\mathcal{E}$ and transformer $\mathcal{T}$ analyses, references to a scalar variable or an array element are distinguished by var and $var(e_1, \ldots, e_n)$. The `READ` regions of an array A in the assignment $var(e_1, \ldots, e_n) = exp$ consist of regions of the right hand side expression (exp) and that of the subscript expressions on the left side $(e_1, \ldots, e_n)$. These regions are defined recursively by the semantic function of `READ` regions for an expression. The `WRITE` region of the input statement $READ\ A$ is the whole array declaration.

Approximations of Array Regions Unlike transformers and preconditions, both under- and over-approximations are computed for array regions. Let's consider the example in Figure 2.3 (Page 13).

Memory management is an important point in program optimization. The principle of array privatization, for example, is to discover the regions that are used as temporary variables in loops, then replace them by local copies for each processor in parallelization. In order to detect such regions, precise analyses are needed. In order to parallelize the outermost loop, we must check that all the read references are covered by written references at the same iteration. This requires a backward analysis, which collects read references in the form of arrays regions, and removes written references from these arrays element sets when they are encountered.

In this case, we state that the last loop uses all the elements of WORK from $N2P+1$ to $2*N2P$, denoted $WORK(N2P+1:2*N2P)$. Elements used by the middle loop are from 1 to $N2P$, denoted $WORK(1:N2P)$. We thus have the set of elements in $WORK$ to be read is from 1 to $2*N2P$, denoted $WORK(1:2*N2P)$. We must then remove (subtract) the elements initialized by the first inner loop ($WORK(1:2*N2P)$); this gives the empty set $\emptyset$.

As we have mentioned above, exact analyses are not decidable. Here, the sets of read array elements are over-approximations of the actual sets (in the example, we have $WORK(N2P+1:2*N2P)+WORK(1:N2P) = WORK(1:2*N2P)$, then we need to remove $WORK(1:2*N2P)$). Since subtracting an over-approximation from another one would not make much sense, the subtracted set of written array elements is necessarily an exact or under-approximation of the actual set.

Note that in PIPS ([Cre96]), exact solutions are computed whenever possible, and switching to conservative approximations is only performed when exactness cannot be preserved anymore.

Size expansion Intraprocedural region analysis has to deal with statements such as assignments, basic blocks, tests, loops and control flow graphs. Regions are built bottom-up from the deepest nodes to the largest compound statement nodes in the control flow graph. It means that at each meet point of a control flow graph, the region information from different control branches are merged with a union operator of two abstract sets. The approximation of regions must be conservative.

In reality, polyhedral computation for regions is even more expensive than for transformers and preconditions (postconditions) because of large size polyhedra, especially when interprocedural regions analysis is applied. Figure 2.7 Page 25 shows polyhedra representing regions analysis for the example.

5 Conclusion

Static program analyses aim at discovering the run-time behavioral properties of a program without actually running it. There are numerous approaches to static program analyses.

Program flow analysis considers a program as a graph whose nodes are basic blocks and whose edges describe how control might pass from one block to another (Section 3.1). In interprocedural analysis, the node set is the set of procedures and the edge set represents procedure calls. Each program analysis is formalized by a set of data-flow equations.

Abstract interpretation is a general theory for calculating and combining analyses rather than specifying them (Section 3.2). The abstraction of a semantics provides an approximation which is less precise but computable. In general, this abstraction is the fixed point solution of an equation system that can be solved iteratively by using the widening and narrowing operators (Section 3.2.2) to speed up convergence.

As examples, we have presented three key analyses of *PIPS* with their brief definitions (Section 4). These examples are simplified in order to show that polyhedra, in form of sets of constraints, are a natural way to approximate the semantic meaning of transformers, preconditions, postconditions and array regions. In these examples, we have seen the application in static program analysis of some polyhedral operators that manipulate polyhedral structures such as the test of emptiness of a polyhedron, the projection along one dimension, or the convex hull of two polyhedra.

Problems that we encounter in the process of those analyses, such as magnitude problems, timeout exceptions or accuracy issues, must be considered at a lower level, by means of sub-libraries. Examples from PIPS's execution are given in the experimental part (Chapter 5).

In the next chapter (Chapter 3), important static analyzers are introduced along with their abstract domains, since these analyzers are based on abstract interpretation (Section 3.2). A brief presentation of available abstract domains, such as polyhedra, octagon and interval domains, as well as their related libraries are introduced.

```
C  <A(PHI1,PHI2)-R-MAY-{1<=PHI1, PHI1<=N2P, 1<=PHI2, PHI2<=1+N1, 1<=N1}>
C  <A(PHI1,PHI2)-W-MAY-{1<=PHI1, PHI1<=N2P, 1<=PHI2, PHI2<=1+N1, 1<=N1}>
C  <WORK(PHI1)-R-EXACT-{1<=PHI1, PHI1<=2N2P, 1<=N1}>
C  <WORK(PHI1)-W-EXACT-{1<=PHI1, PHI1<=2N2P, 1<=N1}>
      SUBROUTINE SHUF (A,N2P,N1,WORK)
      DIMENSION A(N2P,1),WORK(1)
C  <A(PHI1,PHI2)-R-MAY-{1<=PHI1, PHI1<=N2P, 1<=PHI2, PHI2<=1+N1, 1<=N1}>
C  <A(PHI1,PHI2)-W-MAY-{1<=PHI1, PHI1<=N2P, 1<=PHI2, PHI2<=1+N1, 1<=N1}>
C  <WORK(PHI1)-R-EXACT-{1<=PHI1, PHI1<=2N2P, 1<=N1}>
C  <WORK(PHI1)-W-EXACT-{1<=PHI1, PHI1<=2N2P, 1<=N1}>
      DO J = 1, N1, 2                                                 0001
C  <A(PHI1,PHI2)-R-EXACT-{1<=PHI1, PHI1<=N2P, J<=PHI2, PHI2<=1+J, 1<=J, J<=N1}>
C  <WORK(PHI1)-W-EXACT-{1<=PHI1, PHI1<=2N2P, 1<=J, J<=N1}>
         DO I = 1, N2P                                                0002
            II = I+I                                                  0003
C  <A(PHI1,PHI2)-R-EXACT-{PHI1==I, PHI2==J, II==2I, 1<=I, I<=N2P, 1<=J, J<=N1}>
C  <WORK(PHI1)-W-EXACT-{PHI1==II-1, II==2I, 1<=I, I<=N2P, 1<=J, J<=N1}>
            WORK(II-1) = A(I,J)                                       0004
C  <A(PHI1,PHI2)-R-EXACT-{PHI1==I, PHI2==J+1, II==2I, 1<=I, I<=N2P, 1<=J, J<=N1}>
C  <WORK(PHI1)-W-EXACT-{PHI1==II, II==2I, 1<=I, I<=N2P, 1<=J, J<=N1}>
            WORK(II) = A(I,J+1)                                       0005
         ENDDO
C  <A(PHI1,PHI2)-W-EXACT-{PHI2==J, 1<=PHI1, PHI1<=N2P, 1<=J, J<=N1}>
C  <WORK(PHI1)-R-EXACT-{1<=PHI1, PHI1<=N2P, 1<=J, J<=N1}>
         DO I = 1, N2P                                                0006
C  <A(PHI1,PHI2)-W-EXACT-{PHI1==I, PHI2==J, 1<=I, I<=N2P, 1<=J, J<=N1}>
C  <WORK(PHI1)-R-EXACT-{PHI1==I, 1<=I, I<=N2P, 1<=J, J<=N1}>
            A(I,J) = WORK(I)                                          0007
         ENDDO
C  <A(PHI1,PHI2)-W-EXACT-{PHI2==J+1, 1<=PHI1, PHI1<=N2P, 1<=J, J<=N1}>
C  <WORK(PHI1)-R-EXACT-{1+N2P<=PHI1, PHI1<=2N2P, 1<=J, J<=N1}>
         DO I = 1, N2P                                                0008
C  <A(PHI1,PHI2)-W-EXACT-{PHI1==I, PHI2==J+1, 1<=I, I<=N2P, 1<=J, J<=N1}>
C  <WORK(PHI1)-R-EXACT-{I+N2P==PHI1, 1<=I, I<=N2P, 1<=J, J<=N1}>
            A(I,J+1) = WORK(I+N2P)                                    0009
         ENDDO
      ENDDO
      END
```

Figure 2.7: Regions analysis example

Chapter 3

Abstract domains and their signatures

In this chapter, we introduce some important numerical domains used in static program analysis, based on abstract interpretation. We present some related projects with their underlying domains, and discuss in greater detail three of these projects covering most of important abstract domains. We are interested in differences and problems concerning the utilization of these abstract domains. Some important definitions of convex polyhedra, octagons and BDDs are introduced.

Also in this chapter, existing implementations for these abstract domains are presented, in order to show the picture of abstract domains used in actual static analyzers.

1 Numeric Domains

Verifying the correctness of complex software systems requires reasoning about numeric quantities. In particular, an analysis technique may have to discover some relationships among values of *numeric objects*, such as numeric variables, numeric arrays elements, or numeric-valued fields of heap-allocated structures [CH78].

Data-flow analysis is performed by operating on a lattice. Elements of the lattice represent abstract properties of variables, expressions, or other programming constructs for all possible executions of a procedure, independently of the values of the input data and, usually, independently of the control-flow paths through the procedure.

Numeric analyses have been a research topic for several decades, and a number of numeric domains to approximate the numeric states of a system have been designed over the years. These domains exhibit varying precision/cost tradeoffs, and target different types of numeric properties. The list of existing numeric domains includes non-relational domains: intervals [CC76], congruences [MAS93]; weakly relational domains: difference constraints, octagons [MIN04a]; relational domains: polyhedra [CH78], trapezoidal congruences [MAS92] and Presburger formulae [PUG91]. Also, the Binary Decision Diagrams domain is used in ASTRÉE [AST, BCC+03] and NBAC [NBA, JEA00] in addition with numeric domains.

In the following subsections, we describe four main domains which are widely used in static program analysis: interval, polyhedral, octagonal and BBDs domains. Each domain actually forms a lattice over R^n, Q^n or Z^n. In this chapter, we use D^n when we do not specify which one.

1.1 Interval Domain

The simplest numerical domain is the well-known interval domain which was discovered early and has been exploited up until now. Despite its low accuracy, it can be widely applied by static analysis, for constant propagation, elimination of dead code, arithmetic optimization, static range checking, etc. [VCH96, CC76]. In practice, good precision can sometimes be achieved and computation on this domain is normally quite fast (linear in space and time complexity), thus nowadays projects such as ASTRÉE [AST, BCC+03] and NBAC [NBA, JEA00] still use this domain for some fragments of their analyses.

In static analysis using interval domain, values of a variable at each point in the program is estimated by the minimum and maximum values this variable can have. The interval domain is then: closed (scalar) intervals, partially ordered by inclusion with both $\perp$ - the smallest element ($\perp = \emptyset$, the empty interval) and $\top$ - the largest element ($\top = (-\infty, ..\infty)$, where by ∞ we mean the largest machine representable scalar). Figure 3.1 presents an example of a two dimensional interval: $1/2 \leq x \leq 3$ and $1/2 \leq y \leq 3$.

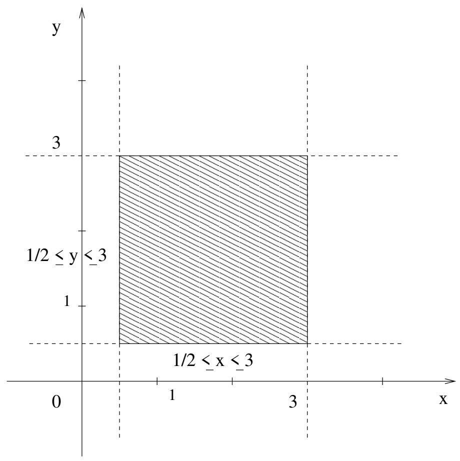


Figure 3.1: Example of an interval

Almost every data type fits into this paradigm; chars, shorts, int, longs, floats and doubles, as discrete approximations to real numbers, structs as aggregate scalars, and even pointers as unsigned integers. Arrays are approximated as the merge of their elements. If $[a, b]$ and $[c, d]$ are (inclusive, closed) intervals, one can consider $[a, b] \leqq [c, d]$ if $a \geq c$ and $b \leq d$. The meet of two intervals $[a, b]$ and $[c, d]$ is then the interval which includes them both: $[min(a,c), max(b,d]$. For the above types of data, the intervals are discrete, even for real numbers; every element is finitely represented, and there exist least upper bounds for arbitrary sets of elements. The existence of fixed points for monotonic functions, and closure under cross-product are then guaranteed.

Using the interval domain is quite fast, but in some cases, large memory space is required if one does not apply approximation. For example, a simple semantic function can result in monotonic chains of very long length, e.g., $[0..0] \leqq [0..1] \leqq \ldots \leqq [-\infty..\infty]$, where ∞ is typically 2^{31}. In the worst case, the monotonicity only ensures convergence after 2^{32} steps per variable. Thus, for any domain, approximation is fundamental.

1.2 Polyhedral Domain

1.2.1 Introduction

The domain of convex polyhedra is extensively used by almost all important projects concerning static analysis, e.g. PIPS [PIPb, IJT91] and NBAC [NBA, JEA00]. Mathematical background on convex polyhedra can be found in "Theory of linear and integer Programming", by Alexander SCHRIJVER [Sch86]. Application of convex polyhedra is precise but has a huge memory cost: exponential in the number of variables. Examples of techniques developed using this domain can be found in Nicolas HALBWACHS [HAL79]'s thesis.

The fundamental theorem on polyhedra is due to Farkas, Minkowski and Weyl. This theorem states that every polyhedron is finitely generated and every finitely generated set is a polyhedron (see [Sch86] page 85).

THEOREM 1.1 (Farkas-Minkowski-Weyl's theorem) Let $a_1, \ldots a_m, b$ be vectors in n-dimensional space. Then:

- either b is a nonnegative linear combination of linearly independent vectors from $a_1, \ldots a_m$;
- or there exists a hyper-plane $\{x|cx = 0\}$, containing $t-1$ linearly independent vectors from $a_1, \ldots a_m$ such that $cb < 0$ and $ca_1, \ldots, ca_m \geq 0$, where $t := rank\{a_1, \ldots a_m, b\}$

DEFINITION 1.1 The *convex hull* of a nonempty set S of n points x_i in *Euclidean* space is the set:

$$convex.hull(S) = \{\sum_{i=n}^{i=1} \lambda_i x_i | x_i \in S, \forall i \lambda_i \geq 0, \sum_{i=n}^{i=1} \lambda_i = 1\}$$

It is *finitely generated* by the vectors $x_1, \ldots, x_n$, thus also denoted as $convex.hull\{x_1, \ldots, x_n\}$.

A nonempty set C of points in *Euclidean* space is called a *convex cone* if $\lambda x + \mu y \in C$ whenever $x, y \in C$ and $\lambda, \mu \geq 0$. A cone is *polyhedral* if $C = \{x|Ax \leq 0\}$ for some matrix A, i.e. if C is the intersection of finitely many linear half-spaces. Here a *linear half-space* is a set of the form $\{x|ax \leq 0\}$ for some nonzero row vector a. The cone C *finitely generated* by the vectors $x_1, \ldots, x_m$ is the set:

$$cone\{x_1, \ldots, x_m\} := \{\lambda_1 x_1 + \ldots + \lambda_m x_m | \lambda_1, \ldots, \lambda_m \geq 0\}$$

As a result of the above theorem, a convex cone is polyhedral if and only if it is finitely generated (see [Sch86] page 87). The *characteristic cone* or *recession cone* of P is the polyhedral cone:

$$char.cone\ P := \{y|x + y \in P\ \forall x \in P\} = \{y|Ay \leq 0\}$$

Thus, the *linearity space* of P is the linear space:

$$linear.space\ P := char.cone\ P \cap -char.cone\ P = \{y|Ay = 0\}$$

THEOREM 1.2 (Decomposition of polyhedra) Any polyhedron P has an unique minimal representation as:

$$P = convex.hull\{v_1, \ldots, v_\alpha\} + cone\{r_1, \ldots, r_\beta\} + linear.space\ P$$

where the set of vertices $v_1, \ldots v_\alpha$ and rays $r_1, \ldots, r_\beta$ are orthogonal to the linearity space of P, and where the $r_i, \forall i = [1, \beta]$, are unique up to multiplication by a positive scalar (see [Sch86] page 106).

More information related to this theorem and the decomposition of polyhedra can be found in [Sch86] from page 85 to page 111. A convex polyhedron P has two representations, namely *H-representation* (H for half-space, also known as constraint system) and *V-representation* (V for vertex, also known as generating system).

DEFINITION 1.2 A *convex polyhedron* consists of points in D^n defined as:

$$P = \{x \in D^n : Ax \leq b\}$$

where A is a matrix of $D^m \times D^n$ and b a vector in D^m.

Then the corresponding *system of constraints* or *constraint system* is:

$$Ax \leq b \tag{3.1}$$

where A is a matrix of $m \times n$ dimension containing the *coefficients*, b is a vector of m components containing m *constants* and x is a vector of n unknown *variables*: $x = (x_1, \ldots, x_n)$. Each line of the system represents a *constraint*. The *base* of P is the basis $\{e_1, \ldots, e_n\}$ of D^n.

The *system of generators* or *generating system* of P consists of three sets: vertices $V = \{v_1, \ldots, v_\alpha\}$, rays $R = \{r_1, \ldots, r_\beta\}$, and lines $L = \{l_1, \ldots, l_\gamma\}$, where:

1. A point of P is a *vertex* (or *extremal point*) of P if it is not a linear combination of other points of P.

   A point u is a *linear combination* of p points $u_i, i = [1,p]$, if $\exists \lambda_i \geq 0, i = [1,p]$, such that $\sum_{i=1}^{p} \lambda_i = 1$ and $u = \sum_{i=1}^{p} \lambda_i u_i$.

2. A *ray* of a polyhedron P is a vector $r \in D^n$ that belongs to P:

$$\forall x \in P,\ \forall \mu \in D^+,\ x + \mu r \in P$$

   An *extremal ray* of P is a ray of P that is not a linear combination of other rays of P.

3. A *line* of P is a vector $l \in D^n$ such that l and $-l$ are rays of P:

$$\forall x \in P,\ \forall \nu \in D,\ x + \nu l \in P$$

   An *extremal line* of P is a line of P that is not a linear combination of other lines of P.

The polyhedron P with vertices v_i, rays r_i and lines l_i is then defined as (see [Sch86], page 106):

$$\left\{ \begin{array}{l} x : \left| \begin{array}{l} \exists \lambda_1, \ldots, \lambda_\alpha \in [0,1] : \sum_{i=1}^{\alpha} \lambda_i = 1 \\ \exists \mu_1, \ldots, \mu_\beta \in D^+ \\ \exists \nu_1, \ldots, \nu_\gamma \in D \end{array} \right. \\ x = \sum_{i=1}^{\alpha} \lambda_i v_i + \sum_{i=1}^{\beta} \mu_i r_i + \sum_{i=1}^{\gamma} \nu_i l_i \end{array} \right\} \tag{3.2}$$

These two representations are dual. We can apply algorithms to change from one to another. In fact, a polyhedron P can also be considered as the intersection of hyper-planes (representing the equalities) and half-spaces (representing the inequalities), defined by constraints.

DEFINITION 1.3 For each vector $a \in D^n$ and scalar $b \in \mathbb{R}$, where $a \neq 0$, and for each relation symbol $\bowtie \in \{=, \geq, >\}$, the linear constraint, denoted $ax \bowtie b$, defines:

- an *affine hyper-plane* if it is an equality constraint, i.e., if $\bowtie \in \{=\}$;
- a *topologically closed affine half-space* if it is a non-strict inequality constraint, i.e., if $\bowtie \in \{\geq\}$;
- a *topologically open affine half-space* if it is a strict inequality constraint, i.e., if $\bowtie \in \{>\}$.

Note that each hyper-plane $ax = b$ can be defined as the intersection of the two closed affine half-spaces $ax \geq b$ and $-ax \geq -b$. Also note that, when $a = 0$, the constraint $0x \bowtie b$ is either a *tautology* (i.e., always true) or *inconsistent* (i.e., always false), so that it defines either the whole vector space D^n called *universe* polyhedron or the *empty* polyhedron $\emptyset$.

Figure 3.2 presents an example of a two dimensional polyhedron, represented by the following constraint system:

$$\begin{cases} x \leq 3 \\ y \leq 3 \\ -x + y \leq 1 \\ -3x - y \leq -3 \\ -x - 3y \leq -3 \\ -x + y \leq -1 \end{cases}$$

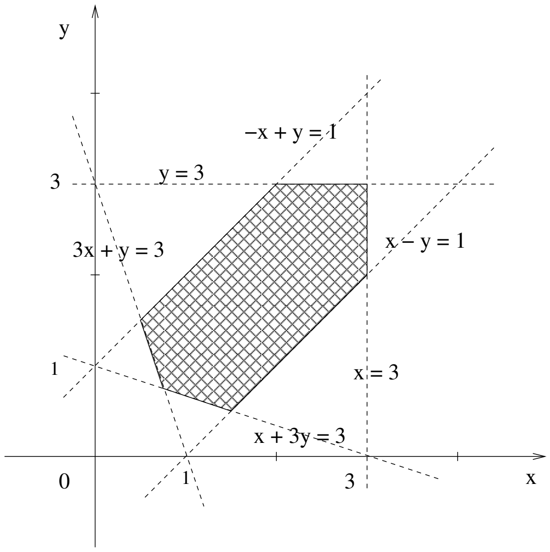


Figure 3.2: Example of a polyhedron

DEFINITION 1.4 We present here some other definitions needed in the following chapters:

- A bounded n-dimensional polyhedron is called a *n-polytope*.

- A polyhedron P is a *closed convex polyhedron* if and only if either P can be expressed as the intersection of a finite number of closed affine half-spaces of D^n or $n = 0$ and $P = \emptyset$.

- The *topological closure* of a polyhedron P is the smallest closed set that contains P. A vector $c \in D^n$ is called a *closure point* of P if it is a point of the topological closure of P.

- A hyper-plane h of D^n is *supporting* the polyhedron P if one of the closed half-spaces of h contains P.

- A subset of the polyhedron P is called a *face* of P if it is either $\emptyset$, P itself or the intersection of P with a supporting hyper-plane.

- The faces of dimension $0, 1, n-2$, and $n-1$ are called the *vertices*, *edges*, *ridges* and *facets* respectively.

- A polyhedron P is *degenerate* in n dimensions if at least one of its vertices lies on $n+1$ or more facets.

From now on, we might omit the words *convex* and *affine*.

1.2.2 Convex Polyhedron Operators

In a polyhedral library, usually both representations of polyhedra are used. Meanwhile, the C^3 library emphasizes constraint systems. Its operators deal only with constraint systems with integer coefficients. As is explained in Chapter 4, which is dedicated to the polyhedral domain, there are some name incompatibilities between the existing projects (Page 88). Furthermore, in some libraries a few operators are not implemented.

We present here an example of some important operations on convex polyhedra that are implemented in the polyhedral library called LINEAR C^3 [PIPa, CAI00] (C^3 for short), which is extensively used in the analyzer and transformer of scientific programs named PIPS [PIPb, IJT91]. This library implements several important algorithms in linear programming such as Simplex, Fourier-Motzkin, etc. IRISA contributed an implementation of Chernikova algorithm [WIL97b] and PRISM a C implementation of PIP (Parametric Integer Programming) [Fea88]. For more details on polyhedral operators, readers are referred to Chapter 4.

The Feasibility of Constraints - A constraint system over D^n is said to be *feasible* when it contains at least one solution in D^n. On the contrary, if a contradiction between its constraints is detected, it is said to be *infeasible*. Otherwise, when we cannot prove that the system is feasible or infeasible, we say that the system is *non practically computable*.

The Elimination of a Variable - The *projection* performs the elimination of a variable. Using constraint systems, it computes the convex hull of the projections of all the rational points that belong to the initial constraint system. The variable is eliminated, even if the projection is not exact, i.e., it introduces new integer points that do not belong to the projections of the initial integer points. It is then a rational/integer algorithm.

The Intersection - The intersection of two constraint systems is the constraint system containing the union of the constraints. It is in fact the intersection of all their hyper-planes (representing the equalities) and half-spaces (representing the inequalities), because a polyhedron is a finite intersection of hyper-planes and half-spaces.

The Union and Convex Hull - The union of two constraint systems is defined by the union of the two corresponding polyhedra. The union of two polyhedra is not necessarily a convex polyhedron. Therefore, the convex hull operator is used instead. It may contain points that do not belong to the original polyhedra. In the library C^3, only this operator calls the *POLYLIB*'s function (Section 3, page 54) to calculate the convex hull, using the generating system.

1.3 Octagonal Domain

1.3.1 Introduction

The octagonal domain proposed by Antoine Mine is a new numerical abstract domain for static analysis by abstract interpretation [MIN04a, MIN01b, MIN01a, MIN04b, MIN02]. It extends a former numerical abstract domain based on Difference-Bound Matrices (DBMs) and allows representation of invariants of the form $(\pm x \pm y \leq c)$, where x and y are program variables and c is a real constant.

This abstract domain proposes a representation based on DBMs of $O(n^2)$ memory cost, where n is the number of variables, and graph-based algorithms manipulating its objects, of $O(n^3)$ time cost in most of cases. The material in this section is taken from the paper [MIN01b].

DEFINITION 1.5 Let $V = \{v_0, \dots v_{N-1}\}$ be a finite set of variables with value in a numerical set I (which can be Z, Q or R). We extend I by adding the $+\infty$ element and standard $\leq, =, +$, min and max to $\overline{I}$.

A *potential constraint* over V is a constraint of the form $(v_i - v_j \leq c)$, where $v_i, v_j \in V$ and $c \in I$. Let C be a set of potential constraints over V. Without lost of generality, C can be represented uniquely by a $N \times N$ matrix m, called a *Difference-Bound Matrix* (DBM) with elements in $\overline{I}$:

$$m_{ij} := \begin{cases} c & \text{if } (v_j - v_i \leq c) \in C \\ +\infty & \text{otherwise} \end{cases} \tag{3.3}$$

A *potential graph* of a DBM m is the weighted graph $G(m) = \{V, A, w\}$, defined by:

$$\begin{array}{ll} A \subseteq V \times V, & w \in A \mapsto I, \\ A := \{(v_i, v_j) | m_{ij} < +\infty\} & w(v_i, v_j) := m_{ij}. \end{array}$$

The $\leq$ order on $\overline{I}$ induces a point-wise partial order $\trianglelefteq$ on the set of DBMs:

$$m \trianglelefteq n \leftrightarrow \forall i, j \; m_{ij} \leq n_{ij}$$

Given a DBM m, the subset of $V \mapsto I$ verifying the constraints $\forall i, j, v_j - v_i \leq m_{ij}$ is denoted $D(m)$ and call the $V - domain$ of m.

constraint over V^+	constraint over V
$v_i - v_j \leq c (i \neq j)$	$v_i^+ - v_j^+ \leq c, v_j^- - v_i^- \leq c$
$v_i + v_j \leq c (i \neq j)$	$v_i^+ - v_j^- \leq c, v_j^+ - v_i^- \leq c$
$-v_i - v_j \leq c (i \neq j)$	$v_j^- - v_i^+ \leq c, v_i^- - v_j^+ \leq c$
$v_i \leq c$	$v_i^+ - v_i^- \leq 2c$
$-v_i \leq -c$	$v_i^- - v_i^+ \leq -2c$

Table 3.1: Translation between extended constraints over V^+ and potential constraints over V

$$D(m) := \{(s_0, \ldots, s_{N-1}) \in I^N | \forall i, j, s_j - s_i \leq m_{ij}\}$$

DEFINITION 1.6 We suppose that $V^+ = \{v_0, \ldots, v_{N-1}\}$ is a finite set of variables, and consider $V = \{v_0^+, v_0^-, \ldots, v_{N-1}^+, v_{N-1}^-\}$ and DBMs over V. Given a potential constraint, a positive variable v_i^+ will be interpreted as $+v_i$, a negative variable v_i^- as $-v_i$. Thus, any set of constraints of the form $(\pm v_i \pm v_j \leq c)$ can be represented by a DBM over V, following the translation defined by Table 3.1.

We define the operator on indices of variables in V by: $\bar{i} := i \oplus 1$, where $\oplus$ is the *bit-wise exclusive or* operator. So that, if i corresponds to v_j^+, then $\bar{i}$ corresponds to v_j^- and if i corresponds to v_j^-, then $\bar{i}$ corresponds to v_j^+.

A DBM m^+ over V representing a set of extended constraints over V^+ is called *coherent* if and only if $\forall i, j, m^+_{ij} = m^+_{\bar{j}\bar{i}}$.

The $V^+ - domain$ of the DBM m^+ is denoted $D^+(m^+)$, defined by:

$$D^+(m^+) := \left\{(s_0, \ldots, s_{N-1}) \in I^N | (s_0, -s_0, \ldots, s_{N-1}, -s_{N-1}) \in D(m^+)\right\} \quad (3.4)$$

The $V - domain$ of a DBM m is not *empty* if its graph corresponding $G(m)$ has no strictly negative cycle. Given a DBM m of which the $V - domain$ is not empty, its *shortest path closure* m^* is defined by:

$$\begin{cases} m^*_{ii} := 0 \\ m^*_{ij} := \min_{1 \leq M}^{(i=i_1, i_2, \ldots, i_M = j)} \sum_{k=1}^{M-1} m_{i_k} i_{k+1} & \text{if } i \neq j \end{cases} \quad (3.5)$$

A DBM m^+ is *strongly closed* if and only if:

- m^+ is coherent: $\forall i, j, m^+_{ij} = m^+_{\bar{i}\bar{j}}$
- m^+ is closed: $\forall i, m^+_{ii} = 0$ and $\forall i, j, k, m^+_{ij} \leq m^+_{ik} + m^+_{kj}$
- $\forall i, j, m^+_{ij} \leq (m^+_{i\bar{i}} + m^+_{\bar{j}j})/2$

The notation of strong closure is needed for the operations of the octagonal domain. From a DBM m^+, we can find its closed form $(m^+)^\bullet$ in $O(n^3)$ in time, by the *strong closure* operator, denoted $^\bullet$.

Figure 3.3 presents an example of a two dimensional octagon, represented by the constraint system:

$$\begin{cases} x \leq 3 \\ y \leq 3 \\ -x + y \leq 1 \\ -x \leq -1/2 \\ -y \leq -1/2 \\ -x + y \leq -1 \end{cases}$$

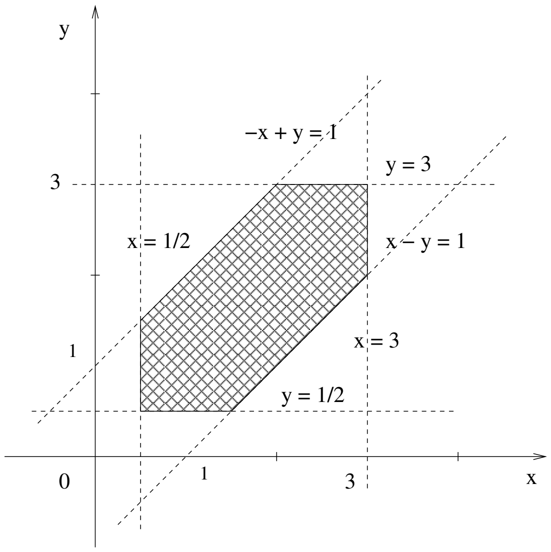


Figure 3.3: Example of an octagon

1.3.2 Octagonal Operators

In this section, we briefly present some important operators. Greater details on these operators and others can be found in Octagon library's documentation, available on the web [MIN, MIN01b].

The Emptiness Test - The $V^+ - domain$ $D^+(m^+)$ can be *empty*, based on the fact that the graph $G(m)$ corresponding has a cycle with a strictly negative weight. The *Bellman-Ford* algorithm ($O(N^3)$ in time) is used in this case.

The Normalization - This algorithm modifies the *Floyd-Warshall* shortest path algorithm of $O(N^3)$ time to find the shortest path closure of a DBM, which is proved to be a normal form.

The Equality and Inclusion Test - Given two DBM m^+ and n^+, we need to compare their $V^+ - domains$. If none of their $V^+ - domains$ are empty, otherwise it is obvious, we have:

- $D^+(m^+) \subseteq D^+(n^+) \leftrightarrow (m^+)^\bullet \trianglelefteq (n^+)^\bullet$

- $D^+(m^+) = D^+(n^+) \leftrightarrow (m^+)^\bullet = (n^+)^\bullet$

The Projection - The interval in which a variable v_i ranges is removed: $\{t | \exists (s_0, \ldots s_{N-1}) \in D^+(m^+)$ such that $s_i = t\} = [-(m^+)^\bullet_{2i,2i+1}/2, (m^+)^\bullet_{2i+1,2i}/2]$

The Union and Intersection - The *max* and *min* operators on $\overline{I}$ lead to the point-wise least upper bound $\vee$ and greatest lower bound $\wedge$ (with respect to the $\trianglelefteq$ order) operators on DBMs:

$$\begin{aligned} [m^+ \vee n^+]_{ij} &:= \max(m^+_{ij}, n^+_{ij}) \\ [m^+ \wedge n^+]_{ij} &:= \min(m^+_{ij}, n^+_{ij}) \end{aligned}$$

In order to keep the octagonal representation, the union of octagons has an upper approximation by using the strong closure algorithm. We have:

- $D^+(m^+ \vee n^+) \supseteq D^+(m^+) \cup D^+(n^+)$
- $D^+(m^+ \wedge n^+) = D^+(m^+) \cap D^+(n^+)$

The Widening - The following definition is a widening operator:

$$[m^+ \bigtriangledown n^+]_{ij} := \begin{cases} {m^+}_{ij} & \text{if } ({n^+}_{ij} \leq {m^+}_{ij}) \\ +\infty & \text{otherwise} \end{cases} \tag{3.6}$$

THEOREM 1.3 Given the definition (3.6) above, we have:

- $D^+(m^+ \bigtriangledown n^+) \supseteq D^+(m^+) \cup D^+(n^+)$
- For all chains $(n^+_i)_{i \in N}$, the chain defined by induction:

$$m^+_i := \begin{cases} {n^+}_0{}^\bullet & \text{if } (i = 0) \\ m^+_{i-1} \bigtriangledown {n^+}_i{}^\bullet & \text{otherwise} \end{cases}$$

  is increasing, ultimately stationary and with a limit greater than the least fixed point of ${n^+}_i{}^\bullet{}_{i \in N}$.

This theorem shows that Definition 3.6 satisfies the two conditions of a widening operator (see Chapter 2, Section 3.2.2 for widening operators).

The Guard and Assignment - They are designed to model tests and assignments in program analysis. Given a DBM m^+ that represents a set of a possible value of the variables V^+ at a program point, an arithmetic comparison g, a variable $v_i \in V^+$, and an arithmetic expression e, the set of possible values of V^+ if the test g succeeds and after the assignment $v_i \leftarrow e(v_0, \ldots, v_{N-1})$ are denoted by $m^+_{(g)}$ and $m^+_{(v_i \leftarrow e)}$.

PROPERTY 1.1 Since the exact representation of the resulting set is in general impossible, an upper approximation is computed:

- $D^+(m^+_{(g)}) \supseteq \{s \in D^+(m^+) | s \text{ satisfies } g\}$
- $D^+(m^+_{(v_i \leftarrow e)}) \supseteq \{s[s_i \leftarrow e(s)] | s \in D^+(m^+)\}$
  where $s[s_i \leftarrow x]$ means s with its i^{th} component changed into x.

1.4 Binary Decision Diagrams Domain

1.4.1 Introduction

Historically, circuits or combinational networks and also sequential networks have been considered suitable computation models for Boolean functions. Binary Decision Diagrams, or BDDs, can be regarded as a computation model as well as a representation or data structure for Boolean functions, because they are representable in polynomial size. The class of Boolean functions $f : \{0,1\}^n \longrightarrow \{0,1\}^m$ is denoted by $B^{n,m}$ or by B^n if $m = 1$.

DEFINITION 1.7 *Syntax:* a BDD on the variable set $X_n = \{x_1, \ldots, x_n\}$ consists of a directed acyclic graph $G = (V, E)$ whose inner nodes (*non-sink* nodes) have out-degree 2 and a labelling of the nodes and edges. The inner nodes get labels from X_n and the *sinks* get labels from $\{0,1\}$. For each inner node, one of the outgoing edges gets the label 0 and the other one gets the label 1.

In a BDD on X_n, each node v represents a Boolean function $f_v \in B_n$ defined in the three equivalent ways (see [WEG00] page 2):

Semantics 1: The computation of $f_v(a), a \in \{0,1\}^n$, starts at node v. At nodes labeled by x_i, the outgoing edge labeled by a_i is chosen. Then $f_v(a)$ is equal to the label of the sink which is reached on the considered path.

Semantics 2: Each input $a \in \{0,1\}^n$ activates all a_i-edges leaving x_i-nodes, $1 \leq i \leq n$. Then $f_v(a)$ is equal to the label of the final node on the unique *activated path* starting at v.

Semantics 3: A sink s with label b ($b \in \{0,1\}$) represents the constant function $f_s(a) \equiv b$. Let v_0 and v_1 be the direct successors of an inner node v reached via the 0-edge and 1-edge, respectively, and let f_0 and f_1 be the functions represented at v_0 and v_1, respectively. If v is labeled by x_i, then f_v is defined by *Shannon's decomposition rule* [C.E49]: $f_v(a) := a_i f_1(a) + \overline{a}_i f_0(a)$ (the product is the Boolean AND, + the Boolean OR, and $\overline{\cdot}$ the Boolean NOT). This can also be written as $f_v(a) := a_i f_1(a) \oplus \overline{a}_i f_0(a)$ ($\oplus$ is the Boolean XOR).

The first semantics reflects the viewpoint that a BDD is a computation model and the main issue is to evaluate the function represented at some specific node efficiently. The second one reflects the more global view that all nodes represent Boolean functions. Both approaches use a top-down view with respect to the underlying directed acycle graph. The third definition suggests a bottom-up approach for the iterative evaluation of the represented Boolean functions. It is proved that each Boolean function can be represented by a BDD.

Figure 3.4 represents the BDD of a function called *hidden weighted bit* HWB_n where $n = 4$. The function is defined as follows: the hidden weighted bit function $HWB_n \in B_n$ computes the bit a_{sum} on the input $a = (a_1, \ldots, a_n)$, where $sum := \ \|a\| \ := a_1 + \ldots + a_n$ and $a_0 = 0$.

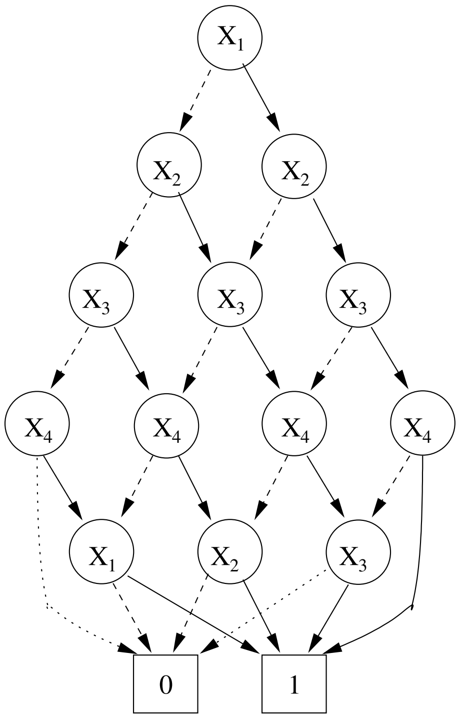


Figure 3.4: Example of a BDD: the function HWB_4

The following definitions can also be found in [WEG00].

DEFINITION 1.8 The *size* of a BDD is equal to the number of its nodes. The *depth* of a BDD is equal to the length of its longest path.

An *inconsistent path* or *null chain* of a BDD G is a directed path containing for some variable x_i a 0-edge leaving an x_i-node and a 1-edge leaving an x_i-node.

An *ite straight line program* is a sequence $I_1, \ldots, I_s$ of instructions of type "if then else" $I_j = ite(C_j, T_j, E_j)$ (ite comes from if-then-else). The condition C_j, the then-branch T_j

and the else-branch E_j are elements from $\{0, 1, x_1, \ldots x_n, I_1, \ldots I_{j-1}\}$. Each instruction I_j represents a Boolean function $f_j \in B_n$ using recursively the semantics $ite(x, y, z) := xy + \overline{x}z$ (if $x = 1$ then y else z).

A BDD is an ite straight line program where the conditions C_j are variables from $\{x_1, \ldots x_n\}$ and $T_j, E_j \in \{0, 1, I_1, \ldots I_{j-1}\}$.

BDDs can represent Boolean functions in a disjunctive normal form (DNF). They support the testing and synthesis of the circuits as well as other purposes. Depending on these purposes, a number of operations are supported. A tradeoff between the physical size of the representation and the performance of the operations is needed. Bryant presents a list of operations on Boolean functions which are useful for their symbolic manipulation [R.E85, R.E86]. Then he presents a model of ordered BDDs (OBDDs) which allows efficient algorithms for all these and many more operations. Other variants of BDDs, such as Free BDDs (FBDDs), Zero-Suppressed BDDs (ZBDDs), Ordered Functional DDs (OFDDs), Randomized BDDs, etc. are then studied.

An example of applications using the BDDs is *integer programming*, which is an $NP-$ *equivalent* optimization problem. Many problems have a direct representation as integer programming problems. The aim is to minimize (or maximize) a linear function under conditions given by linear inequalities and the condition that the variables only take integer values. Without loss of generality, we can assume that the problem is given in the following form:

$$c_1 x_1 + \ldots + c_n x_n \rightarrow \min$$

where $a_{i1}x_1 + \ldots + a_{in}x_n + b_i \leq 0, 1 \leq m$, and $x_1, \ldots x_n \in Z$.

In most practical problems, we have bounds like $d_i \leq x_i \leq e_i$. Then we replace x_i with $x_i - d_i$ and obtain $0 \leq x_i \leq e_i - d_i$. If $e_i - d_i \leq 2^k - 1$ for some k, we replace x_i with k Boolean variables $x_{i,0}, \ldots, x_{i,(k-1)}$ with the interpretation that $x_i = x_{i,0} + 2x_{i,1} + \ldots + 2^{k-1}x_{i,(k-1)}$. We have to add the inequality $x_i - (e_i - d_i) \leq 0$. This leads to the special case of *binary programming* where the variables have to take Boolean values, thus we can use the BDDs.

Now we present the proposed list of key operations on BDDs; using graph representations of Boolean functions, we denote the representation of f by G_f. The operators described briefly in the next section are based on the book [WEG00].

1.4.2 BDD operators

The Evaluation - Given a BDD G_f on n variables, $a \in \{0, 1\}^n$, we have the value of function $f(a)$ as output.

The Binary Synthesis - Given two BDDs G_f, G_g, and a Boolean operation $\otimes$: $\{0, 1\}^2 \longrightarrow \{0, 1\}$. We have output G_h where $h = f \otimes g$.

The Satisfiability Test - Given a BDD G_f. We test if there exists some a such that $f(a) = 1$.

The Satisfiability Count - Given a BDD G_f. We return the number of inputs a such that $f(a) = 1$.

The Equivalent Test - Given two BDDs G_f, G_g, we test if $f \equiv g$, i.e., $f(a) = g(a)$ for all $a \in \{0,1\}^n$.

The Replacement by Constants - Given a BDD G_f on n variables $x_i \in X_n$, and a constant $c \in \{0,1\}$. The operation returns G_g where $g := f_{|x_i=c}$ is the *sub-function* or *cofactor* of f for $x_i = c$, or: $g(a_1, \ldots a_n) := f(a_1, \ldots, a_{i-1}, c, a_{i+1}, \ldots a_n)$.

The Replacement by Functions - Given two BDDs G_f, G_g on n variables $x_i \in X_n$. The output is G_h where $h := f_{|x_i=g}$, or: $h(a_1, \ldots a_n) := f(a_1, \ldots, a_{i-1}, g(a_1, \ldots a_n), a_{i+1}, \ldots a_n)$.

The Redundancy Test - Given a BDD G_f on n variables $x_i \in X_n$, we test if f does not depend on x_i, which means, if $f(a_1, \ldots, a_{i-1}, 0, a_{i+1}, \ldots, a_n) = f(a_1, \ldots, a_{i-1}, 1, a_{i+1}, \ldots, a_n)$ for all $a_j \in \{0,1\}$.

The Quantification - Given a BDD G_f on n variables $x_i \in X_n$, and a quantifier $Q \in \{\exists, \forall\}$. We return G_g such that: if $Q = \exists$, then $g := (\exists x_i), f := f_{|x_i=0} + f_{|x_i=1}$; if $Q = \forall$, then $g := (\forall x_i), f := f_{|x_i=0} \wedge f_{|x_i=1}$ ($\wedge$ is the Boolean AND).

The Minimization - Given a BDD G_f, the output is G_f^*, a representation of f of the same representation type as G_f which has minimal size for f.

The above list is not exhaustive, and there exist several libraries implementing different operators for the BDDs domain, e.g. CUDD [Som], that can be extended for specific uses, e.g. CUDDAUX [JEAa].

1.5 Presburger formulae - The OMEGA project

1.5.1 Introduction

The OMEGA project has two major components. One component is the OMEGA test, a system for manipulating sets of affine constraints over integer variables, e.g. decision test for the existence of integer solutions to affine constraints. The other component is a framework for analyzing and transforming programs using the OMEGA test (PETIT and UNIFORM tools) [Ome, PUG91].

The OMEGA test forms an abstract domain by implementing a system for simplifying and verifying Presburger formulae. Presburger formulae use affine constraints, the usual logical connectives $=, <, \leq, >, \geq, \neg, \wedge, \vee, \implies$, and existential $\exists$ and universal $\forall$ quantifiers. Thus in comparison with the polyhedral domain, it is more expressive.

In order to describe some important operators later on, instead of complete definitions (which can be found in OMEGA's documentation [Ome, PUG91]), we present here two examples of OMEGA's structures, a set S and a relation R:

$$S := \{[x] \mid (0 \leq x \leq 100 \ \wedge \ \exists y \text{ such that } (2n \leq y \leq x \ \wedge \ y \text{ is odd })) \ \vee \ x = 17\}$$

$$R := \{[i, j] \longrightarrow [i', j'] \mid 1 \leq i \leq i' \leq n \ \wedge \ \neg(F(i) = F(i')) \ \wedge \ 1 \leq j, j' \leq m\}$$

Figure 3.5 presents another example of a Presburger formula that represents the logical OR (denoted $\vee$) of three polyhedra in two dimensions.

$$\begin{cases} y \leq 3 \\ -3x - y \leq -3 \\ x - y \leq -1 \end{cases} \vee \begin{cases} -3x - y \leq -3 \\ x + 3y \leq 3 \\ x - y \leq 1 \end{cases} \vee \begin{cases} -x \leq -3 \\ x - y \leq 1 \\ y \leq 3 \end{cases}$$

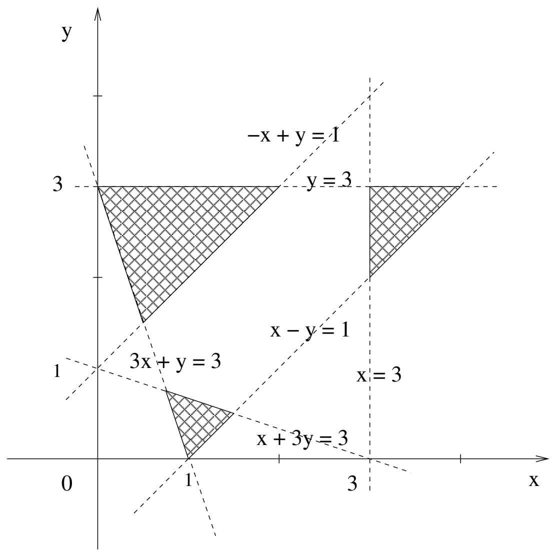


Figure 3.5: Example of a Presburger

The OMEGA test cannot simplify all Presburger formulae efficiently: there is a 2^{2^n} non-deterministic lower bound and a $2^{2^{2^n}}$ deterministic upper bound on the time required to verify Presburger formulae.

The domain of Presburger formulae can provide more useful information for static analysis than polyhedra, since the OMEGA test returns symbolic answers, instead of boolean answers. For example, dependence analysis using polyhedra is often structured as a decision problem: tests simply answer yes or no. Meanwhile, using Presburger formulae (i.e. the OMEGA test), one can construct a set of constraints that describes all possible dependency distance vectors which can be used directly in deciding the validity of program transformation (see [Ome, PUG91]). However, because of its complexity, large scale applications should use this abstract domain only in complement with other domains.

1.5.2 Presburger formulae operators

The Satisfiability Test checks if the Presburger formula (a set S or a relation R) is satisfiable or not.

The Simplification operator simplifies the Presburger formula to its *simplified* form (see OMEGA's document for more details [Ome, PUG91]).

The Projection operator of a set (or a relation) $A := \{x|f(x)\}$ along a variable v is a Presburger formula $A' := \{x|\exists z \text{ such that } f'(x)\}$, where f' is the same as f except that all occurrences of variable v are replaced by z.

The Intersection operator of two sets (or two relations) of the same dimensions $A := \{x \mid P(x)\}$ and $B := \{y \mid Q(y)\}$ gives a Presburger formula $A \cap B := \{z \mid P(z) \wedge Q(z)\}$.

The Difference operator of two sets (or two relations) of the same dimensions $A := \{x \mid P(x)\}$ and $B := \{y \mid Q(y)\}$ gives a Presburger formula $A \setminus B := \{z \mid P(z) \wedge \neg Q(z)\}$.

The Union operator of two sets (or two relations) of the same dimensions $A := \{x \mid P(x)\}$ and $B := \{y \mid Q(y)\}$ gives a Presburger formula $A \cup B := \{z \mid P(z) \vee Q(z)\}$.

The Convex Hull operator of a set A is a Presburger formula representing the smallest set of inequalities whose intersection contains A.

In fact the computation of the convex hull operator in OMEGA is approximated since it is an expensive operation. For more information, readers are referred to OMEGA's documentation, page 31.

1.6 List of Polyhedra - Arnauld Leservot's work

Arnauld LESERVOT, in his dissertation, has shown the utilization of the domain of lists of polyhedra [LES96]. Starting from the fact that all Presburger formulae can be represented in Disjunctive Normal Form (DNF) or Conjunctive Normal Form (CNF), so an element of the domain can have two representations based on convex polyhedra.

Given k convex polyhedra P^i, $i = [1, k]$, in DNF, we have a disjunction of these polyhedra:

$$D(x) = \{x \in D^n : \vee_{i=1}^{k} P^i(x)\}$$

In CNF, we then have a conjunction of a convex polyhedron and the negation of others:

$$C(x) = \{x \in D^n : P^0(x) \wedge_{i=1}^{k} \neg P^i(x)\}$$

where P^0 is the *positive polyhedron* of the element C. These representations can be referred as a list of convex polyhedra. Figure 3.6 presents an example of the two representations.

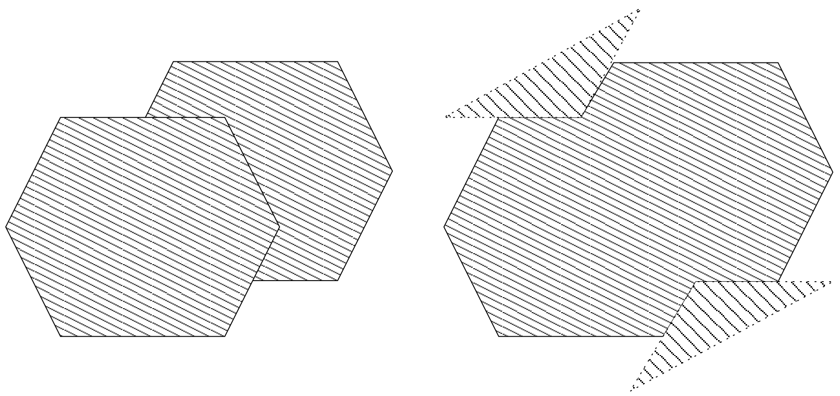


Figure 3.6: Example: DNF (left) and its CNF(right)

The set of lists of polyhedra, with its operations like union, difference, inclusion and equality between two elements, the emptiness test as well as the conversion from one form to another, form an abstract domain, which has been implemented in PIPS for the calculation of Array Data Flow Graph.

1.7 Other approaches

General abstract domains like intervals, octagons or polyhedra cannot offer enough precision in all cases, in particularly in non-linear cases. However, general purpose non-linear abstract domains would probably be extremely costly. For example, most algorithms from semi-algebraic geometry have costs in the form of towers of exponentials.

Another approach which has been explored recently is more domain-aware and restricts its preciseness ambition to specific constructs such as digital filtering in [FER05b, FER04, FER05a]. Consider this following C code:

where a and b are constants, i, j, t are expressions, C is a boolean expression, X, Y, X' are program variables. If we use octagonal or interval domain, we have an imprecise result that X and Y may take any value.

If t is bounded: $|t| \leq t_M$, and $0 < b, a^2 - 4b < 0$ and $k \geq \left(\frac{t_M}{1-\sqrt{b}}\right)^2$, then the constraint $X^2 - aXY + bY^2 \leq k$ is preserved by the affine transformation $X' := aX - bY + t$. Thus

```
if (C) {
  Y := i;
  X := j;
} else {
  X' := aX - bY + t;
  Y := X;
  X := X';
}
```

Figure 3.7: Ellipsoid domain: Example of C code

this type of formulae forms a new abstract domain (called *ellipsoid domain*) that gives a more precise result. This approach has proved to be precise and efficient [BCC+03].

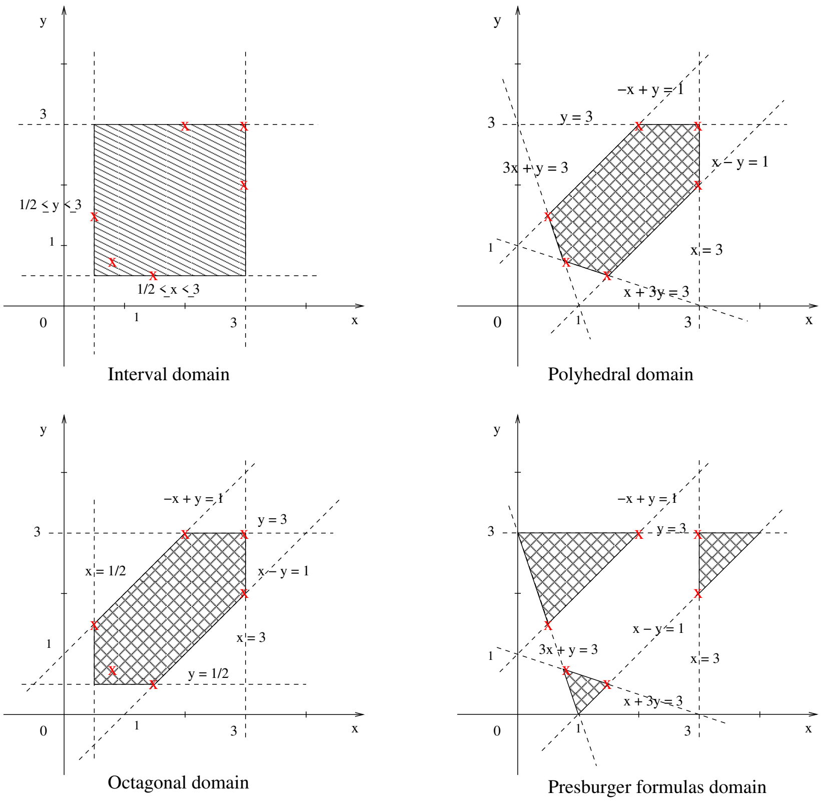


Figure 3.8: Representations of a finite set of points

Figure 3.8 gives an example which compares the four domains, where it shows that the Presburger formulae domain is even more expressive than polyhedral domain: disjunctions of polyhedra are representable.

2 Projects and Their Underlying Domains

In this section, we present three static analyzers using all abstract domains available nowadays. These projects share many features, among which the capacity to deal with real-life applications is key. Large scale applications lead to problems of robustness and effectiveness in currently used static analyses. They participate to the APRON project [APRb]. Then, some related projects are briefly presented.

2.1 ASTRÉE Project

ASTRÉE [AST, BCC+03] is a static program analyzer for a class of C programs. It aims at proving the absence of Run Time Errors (RTE). It does not deal with dynamic memory allocation and recursion.

The analyzer, based on abstract interpretation [CC79], proceeds by approximating the set of all possible inputs into a symbolic representation. The program to be analyzed is interpreted on this set of values, approximating each basic instruction when necessary to keep manageable the sets of representable values.

Inputs of ASTRÉE are C programs used in real-time critical industrial applications of 100000 to 400000 lines of code. Analyses can take several hours of execution time. A tradeoff between precision and efficiency in static program analyses is then unavoidable.

In abstract interpretation, this tradeoff can be tuned in two main categories: the widening strategy and the symbolic representation of the sets of values. The latter can be split into a number of abstract domains; each domain is specialized on certain shapes of sets of values, in order to obtain the most appropriate approximation/precision for the analyses.

The authors of ASTRÉE have proposed and experimented successfully several new abstract domains. Among those domains used by ASTRÉE, we are interested in the interval, octagonal and BDD domain. Considered costly, the polyhedral domain is not implemented in ASTRÉE, whereas the octagonal domain [MIN01b, MIN01a, MIN02] is newly introduced by Antoine MINÉ. This abstract domain presents a good compromise between complexity and precision, between the interval domain, which is not expressive enough for complex analyses, and the polyhedral domain, which shows problems in time and magnitude complexity. Moreover, like the polyhedral domain, it is a general purpose abstract domain, thus interesting, while other abstract domains such as digital filters (or ellipsoid domains, see [FER05b, FER04, FER05a]) are actually very specific.

The octagonal domain is more expressive than the interval domain; for example, it can represent the relation $x = y$ which is in fact a line that ranges $[-\infty, \infty]$. It is however not as expressive as the polyhedral domain, for example with the relation $x - 2y = 0$ (see also Figure 3.8, Page 45 for another example).

This domain has been developed and used in ASTRÉE in order to replace the widely used polyhedral domain. However, several incompatibilities between the two domains exist, even at the interface level. This problem is addressed in Chapter 4. Thus, it is not only difficult to use existing implementations of the polyhedral domain in ASTRÉE, but also much more difficult to exploit this octagonal domain in other analyzers that still use the polyhedral domain without major changes to the analyzers. In the goal of expanding the utilization of this new domain, some adaptation needs to take place.

The design of ASTRÉE is parametric. This means that the rate between the cost and the precision of its analyses can be adapted to the needs of users. The analyzer is modular, which means for each domain we have independent libraries. Different modules of ASTRÉE, based on different abstract domains, can be assembled and parameterized to build application-specific analyzers. This also leads to a complex utilization, i.e. users need to "know" the problem, requiring expertise.

The implementation of the project amounts to about 60000 lines of OCaml and 4000 lines C code (the Octagon library).

2.2 NBAC Project

This tool analyses synchronous and deterministic reactive systems containing combinations of Boolean and numerical variables and continuously interacting with an external environment [NBA, JEA00]. Its input format is directly inspired by the low-level semantics of the LUSTRE dataflow synchronous language [LUS].

The kind of analyses performed by NBAC are:

- Reachability analysis from a set of initial states, which computes invariants satisfied by the system.
- Co-reachability analysis from a set of final states, which computes the sets of states that may lead to a given final state.
- Combination of the above analyses, which allows to either verify invariance or more generally safety properties or to slice a system, i.e., to compute sets of states (or their approximations) belonging to executions from an initial state to a final state.

NBAC has been developed mainly with the Caml language [Cam], and uses the BDD library CUDD [Som], and the convex polyhedra library New POLKA [JEAb, JEA00]. The tool is founded on the theory of abstract interpretation (Chapter 2). Sets of states are represented in an approximate way by abstract values belonging to an abstract domain, and computations are performed on this abstract domain. This leads to conservative results: if a state is shown unreachable (or not co-reachable), then it is for sure.

The abstract domain used is the direct product of the Boolean lattice and the convex polyhedra lattice. A set of states is represented by the conjunction of a BDD for the Boolean variables of the program, and a convex polyhedron for its numerical variables.

Using a single abstract value would lead to very imprecise results. The tool works actually with an explicit control structure, defined by a partitioning of the state space.

This allows to represent a global set of states as a finite union of abstract values, instead of a single abstract value. The first feature of NBAC is to use a very general notion of control structure in order to precisely tune the tradeoff between precision and efficiency in the analyses.

The second feature of NBAC is the ability to dynamically refine the control structure according to the needs of the analysis. The initial control structure is either provided by the user or generated by the tool. Its refinements during the analysis allow to improve the accuracy of the results and to remove states which have already been shown unreachable and/or not co-reachable.

To handle the BDDs, an extension to the well-known Colorado University Decision Diagram (CUDD), called CUDDAUX, is developed. This library offers additional features and new functions are implemented (see [Som, JEAa] for the list of functions). The New POLKA library is mainly based on the IRISA library POLYLIB [LOE, Wil] and the old library used in the POLKA tool inside the SYNCHRONE team of the VERIMAG laboratory [LUS].

In fact, the approach using BDDs and polyhedra as abstract domains is fairly common. To overcome problems linked to the high complexity of polyhedral computations, an adaptation to the octagonal domain is attempted. A very simple interface between the polyhedral engine, New POLKA, and the octagonal one has been established. However, this interface is not sufficient in many areas: *genericity*, completeness, compatibility with other libraries, etc.

The implementation of NBAC is about 10000 lines of Objective Caml, as well as 750 lines of Objective Caml plus 2000 lines of C code for interfacing the libraries manipulating BDDs and polyhedra.

2.3 PIPS Project

The goal of the PIPS project [PIPb, IJT91] is to develop a free, open and extensible workbench for automatically analyzing and transforming scientific and signal processing applications.

The PIPS workbench includes program compilation, reverse-engineering, program verification, source-to-source program optimization and parallelization. Its interprocedural analyses help with program understanding and with checking legality and impact of automatic program transformations. These transformations are used to reduce the execution cost and latency.

PIPS takes as input Fortran 77 codes and emphasizes interprocedural techniques for static program analyses. It automatically computes affine preconditions for integer scalar variables and several kinds of polyhedral array regions (READ, WRITE, IN and OUT) which are used for array and partial array interprocedural privatization as well as for interprocedural parallelization (see [Cre96] for analyses of array regions). Program verification techniques such array bound checking, alias analysis and uninitialized variables checking are also implemented [NGU02]. For examples of analyses in PIPS, readers are referred to Chapter 2, Section 4, Page 12.

PIPS can be used as a reverse-engineering tool. Region analyses provide useful information about procedure effects, whereas partial evaluation and dead-code elimination based on preconditions reduce code size. Cloning has also been used successfully to split a routine implementing several functionalities into a set of routines, where each routine provides only one functionality. Automatic cleaning of declarations is useful when commons are over-declared thru include statements.

Static analyses compute call graphs, memory effects, use-def chains, dependence graphs, interprocedural checks, transformers, preconditions, continuation conditions, complexity estimation, reduction detection, array regions and aliases. Several parallelization algorithms are available, as well as automatic code distribution. The polyhedral method of Feautrier is also implemented [Fea96].

Program transformations include loop distribution, scalar and array privatization, atomization, loop unrolling, strip-mining, loop interchange, partial evaluation, dead-code elimination, use-def elimination, control restructuring, loop normalization, declaration cleaning, cloning, forward substitution and expression optimizations. Even though inputs for PIPS actually are only Fortran programs, a version for C program is under development, which can re-use developed techniques.

Having around 100000 lines of C code, PIPS is built on top of two other libraries. The first one is Newgen which provides basic manipulation functions for persistent supported data structures. All PIPS data-types are based on Newgen.

The second tool is the LINEAR C^3 library which handles vectors, matrices, linear constraints and structures based on these such as polyhedra. The library is extensively used since in PIPS, analyses such as dependence testing, precondition and region computation [IJT91, Cre96], and transformations such as loop interchange and tiling [Bou02] are based on linear algebra techniques.

The computation within the C^3 library has shown to be very expensive when it deals with large scale industrial applications. Our experiments (Chapter 5) have shown that it is not because the implementation in the library is not efficient enough with respect to other implementations, but because of the exponential complexity of the polyhedral domain and its variant, the domain of lists of polyhedra (Section 1.6, page 43).

Polyhedral domain-based analyses in PIPS require great capacity of computation. Limits in time and memory space have resulted in precision loss in analyses. In the worst case they can block analyses. In order to refine the tradeoff between precision lost and the efficiency of execution, other abstract domains have to be adopted.

The octagonal domain can be seen as a good compromise, which stands in between the interval domain and the polyhedral domain. Nonetheless, in many cases, the precision provided by this domain is not good enough. We are sometimes interested in a even higher precision with the Presburger formulae (Section 1.5, page 41), or the list of polyhedra

(Section 1.6, page 43).

The open problem is then: how can we benefit from all these abstract domains, and how can we provide a mechanism to switch among them? A generic interface has been developed in Chapter 4 in order to deal with this problem.

2.4 Other Projects

CHINA Data Flow Analyzer. In order to perform precise and practical analyses, most analyzers try to find a good way around the abstract domains. New abstract domains and their combinations are discovered and experimented throughout many projects, and each project has its own specific goals.

The abstract domain employed in CHINA [CHI] for detecting the property of definite *groundness* is based on the domain of positive Boolean functions, where many operations on Boolean formulae have exponential worst-case complexity, and also the polyhedral domain, using the Parma library [PAR, BRZaH02]. Reduced Ordered Binary Decision Diagrams (ROBDDs), also known as Bryant graphs, are a representation for Boolean functions supporting many efficient operations. An extension of this domain is described in [BS99].

Another specific abstract domain is used in this analyzer to capture *pair-sharing*. This domain is based on the *set-sharing* domain Sharing of Jacobs and Langen [HZB01, BHZ02]. The complexity of the Sharing domain is exponential in the number of program variables.

CTI. In program analysis, termination inference answers the termination question, with a compact formula called a termination condition. The CTI system analyses programs to infer termination of programs with Prolog style execution [CTI].

CTI provides compact explanations for the provable properties of termination of all predicates and improves a slicing-system in explaining reasons of non-termination. It is written in SICStus Prolog, based on abstract interpretation by using the polyhedral domain and relying mostly on the Parma library [PAR, BRZaH02].

Action Language Verifier. Action Language is a specification language for reactive software systems. It supports both synchronous and asynchronous compositions and hierarchical specifications. An Action Language specification consists of integer, boolean and enumerated variables, parameterized integer constants and a set of modules and actions which are composed using synchronous and asynchronous composition operators [BYK01].

Action Language Verifier is an infinite state symbolic *model checker* [BGP97]. It consists of a compiler that converts action language specifications to composite symbolic representations, and a model checker that verifies CTL properties of Action Language specifications [ALV]. It uses conservative approximation techniques, reachability and acceleration heuristics to achieve convergence.

Action Language Verifier uses the Composite Symbolic Library as its symbolic manipulation engine [YKTB01]. It supports polymorphic verification procedures which dynamically select symbolic representations based on the input specification. Composite Symbolic Library combines different symbolic representations, such as BDDs for representing boolean logic formulae and polyhedral representations for linear arithmetic formulae, with a single interface.

Based on this common interface, these data structures are combined using a composite representation. The idea is to use a disjunctive normal form where every disjunct consists of a conjunction of different symbolic representations. Enumerated and boolean variables are represented by BDDs, and integer variables are represented with polyhedral representations.

The composite symbolic library is designed and implemented in an object-oriented way, where CUDD [Som] and OMEGA Library [Ome, PUG91] are imported. Recently, an experimental version of Action Language Verifier using the Parma library [PAR, BRZaH02] is developed.

Overall, the composite symbolic library addresses the same problem as does our approach of a common interface (Chapter 4). Nevertheless, the library is designed for the model checking community [YKTB01], deals only with BDDs and Presburger domains (the version using PPL is not yet finished), and emphasizes the data structures rather than the operators. Actually, the prototype using Composite Symbolic Library deals only with a subset of analyses [YAV04]. Meanwhile, octagonal and polyhedral domains consist of many more operators defined in abstract interpretation. Our approach targets directly three analyzers (ASTRÉE, NBAC, PIPS) for incompatibilities, trying to solve existing problems in their underlying abstraction engines, that is to say the polyhedral libraries C^3 [PIPa, CAI00], New Polka [JEAb, JEA00], PPL [PAR, BRZaH02] and the Octagon library [MIN, MIN01b].

CSSV (C String Static Verifier) is a tool that statically uncovers string manipulation errors in C programs [Da]. A prototype has been implemented, based on the polyhedral domain, using the New POLKA library (see Section 3, page 54). The tool uses static analysis to reduce the problem of checking string manipulations to that of checking integer manipulations, a problem for which well-known solutions exist.

3 Available Polyhedral Libraries and Operators

Much work has been concentrated on the possible improvements of the underlying mathematical algorithms. Available implementations for manipulating polyhedra include some "complete" polyhedral libraries like *POLYLIB* [LOE, Wil], *New POLKA* [JEAb, JEA00], C^3 [PIPa, CAI00], or just libraries that focus on some particular operators, like *CDD* [FP95], *LRS* [AF92] or *JANUS* [SOG, SOG96]. *JANUS* deals with the satisfiability problem using Simplex method [SOG96, ST01], whereas the calculation of the convex hull of two or more polyhedra is maybe the most interesting operator [AB95, BDH96, FLL01, FQ88, BAY].

$\mathbf{C^3}$ also known as *LINEAR* library, is a library that handles vectors, matrices, linear constraints and structures based on them, such as polyhedra [PIPa, CAI00]. This library implements several important algorithms in linear programming such as Simplex, Fourier-Motzkin, etc. IRISA contributed an implementation of the Chernikova algorithm [WIL97b] (see Section 3, page 54 Page 54) and PRISM a C implementation of PIP - Parametric Integer Programming [Fea88]. These algorithms are designed for integer and/or rational coefficients.

Although it is a ten year old library which has been used and optimized several times by PIPS members, this library still encounters problems when dealing with large scale industrial applications. Here are some specific notes about the library:

- Overflow exception handlers by longjumps are systematically implemented;
- Memory management is dynamic;
- Arithmetic modes are integer $32-bit$, $64-bit$, $128-bit$ (GNU Multi Precision is not provided);
- Using a *sparse representation*: variable name and constant term handling;
- A rich set of debugging functions.

CDD stands for C implementation of Double Description method, developed by Komei FUKUDA [FUK]. It comes with two versions, C++ and C, that implement the method of Motzkin and al for generating all vertices (i.e. extremal points) and extremal rays of a general convex polyhedron P, given by a constraint system, and vice versa. Also, CDD can solve a linear programming problem, i.e. a problem of maximizing and minimizing a linear function over P.

The program CDD+ is a C++ version of the ANSI C program CDD, basically for the same purpose. The input, sharing the same format than LRS's (see Section 3, page 53), consists of either an $H-representation$ or $V-representation$ of a convex polyhedron, which needs not to be full dimensional. Linearities, respectively hyper-planes or lines, are allowed in either description [FUK]. Here are some specific notes about the library:

- There are no overflow handlers, thus strange behavior can sometimes be observed;
- A number MAX_NB_RAYS must be defined for memory management;
- Arithmetic modes are rational GNU Multi Precision and C built-in double floating-point;

- A mixed arithmetic mode option is implemented. CDD uses floating-point arithmetics first and then checks with rational GNU Multi Precision whether the output is correct. According to the author, when using this mixed mode, it runs much faster (from 5 to 10 times) for most of the cases than when using only GNU Multi Precision (see the CDD's homepage [FUK]).

JANUS is a software developed by Jean-Claude SOGNO at INRIA, addressing the emptiness test of a system of affine constraints. The algorithm is presented in [SOG, SOG96]. However, there are two problems that we would like to address. Firstly, the original JANUS was written with $32-$bit integers. This really is a limit, so a wide range of large systems containing large numbers cannot be solved, especially when these numbers accumulate after some computational iterations. Secondly, there is no head-to-head comparison between equivalent methods that can show the efficiency of the method, except a comparison between JANUS and OMEGA test [Ome, PUG91] concerning the "nightmare" problem (see [SOG96]).

To overcome the first problem, we have modified *JANUS* using a generic integer called *Value* to hide 32-bit, 64-bit and multi precision computations from the original algorithm. The second problem is dealt with in Chapter 5.

An additional advantage of the modification is that, in spite of the overflow checking already integrated in JANUS, our implementation of JANUS using macros Value sometimes detects overflows that are not detected by the original JANUS (Chapter 5).

LRS (Lexicographical Reverse Search) is an ANSI C implementation of the reverse search algorithm for vertex enumeration/convex hull problems [Avi], by David AVIS. It comes with two main driver programs: lrs and redund, a set of demo drivers suitable for customization and a choice of three arithmetic packages (lrsmp, a multiple precision arithmetic package; lrslong, a fixed precision integer package; lrsgmp, a multiple precision arithmetic package, based on GNU Multi Precision). Input file formats are compatible with Fukuda's CDD package [FUK].

All computations are performed exactly in either multi-precision or fixed integer arithmetic. Output is not stored in memory, so even problems with very large output sizes can sometimes be solved. LRS converts an H-representation to a V-representation and vice versa. The possibility of estimating the number of vertices/rays or facets of a polyhedron is also implemented (using another algorithm that costs less than the reverse search algorithm). It also can remove redundant constraints from an $H - representation$, or find the extremal vertices in a $V - representation$. Here are some specific features about this library:

- There is an overflow handler implemented in lrsmp version; a number MAX_DIGITS need to be defined for this purpose;

- Memory is allocated at the beginning, then garbage collection is implemented to clean up after each problem has been solved;
- Arithmetic modes are integer $32 - bit$, $64 - bit$ and GNU Multi Precision, with and without overflow checking.

New POLKA is a library handling convex polyhedra, whose constraints and generators have rational coefficients. It was written by Bertrand JEANNET in ANSI C, but an interface to the language OCaml version 3.00 is also provided [JEAb, JEA00]. This library is currently used in the verification tool NBAC of the same author (see NBAC in Section 2.2, page 47), and also by other research teams working on static analysis and abstract interpretation.

It is mainly based on IRISA's POLYLIB library (see Section 3, page 54) and on the old library used in the POLKA tool inside the SYNCHRONE team of the VERIMAG laboratory [LUS]. The main motivation to develop a new library was the need for multiprecision integers and 64-bit integers (POLYLIB now has this feature, too). The interface and memory management have also been changed, and some new memory strategies have been included to save computation time.

Implemented operations include creation of polyhedra from constraints or generators, intersection, convex hull, image and preimage by linear transformations, widening operator. Here are some practical points about the library:

- There are no overflow handlers, thus infinite loop can be observed;
- A number POLKA_MAX_NB_RAYS must be defined for memory management;
- Arithmetic modes are integer $32 - bit$, $64 - bit$ and GNU Multi Precision;
- Using a *dense representation*: Variables and constant term are enumerated by integers $(0 \ldots n)$.

POLYLIB is a polyhedral library operating on objects made up of unions of polyhedra of any dimension, which was developed initially at IRISA, in Rennes, France, in connection with the ALPHA project [LOE, Wil, Loe99]. It was written in ANSI C, and designed to be general purpose and has since been used by several projects, e.g. PIPS [PIPb, IJT91].

Polyhedra are represented internally in their full dual form as a list of mixed constraints (equalities and inequalities) or a list of geometric features: vertices, rays, and lines. POLYLIB was originally written in 1993 by Hervé LEVERGE and Doran WILDE [HLV94, Wil97a] and was based on the Motzkin Double Description method for finding the dual representation of a polyhedron and on an implementation of Chernikova's algorithm written by Herv LEVERGE [Ver92, Ver94]. Philippe CLAUSS is the one who first signaled the possibility of counting the number of integer points in a union of rational convex polytopes by a special kind of polynomial called Ehrhart polynomials [Cla96]. Thus, Ehrhart polynomials are implemented in POLYLIB.

The following polyhedral operations are supported in POLYLIB: intersection, union, difference, simplification (widening) of a polyhedron in the context of another polyhedron, convex hull, image by an affine multi-dimensional transformation function and preimage by an affine multi-dimensional transformation function. Here are some practical notes about this library:

- Overflow exception handlers by longjumps are systematically implemented;
- A number MAX_NB_RAYS must be defined for memory management;
- Arithmetic modes are integer $32 - bit$, $64 - bit$, $128 - bit$ and GNU Multi Precision;
- Variables and constant term are represented by integers $(0 \ldots n)$.

Parma Polyhedral Library - PPL (Parma Polyhedra Library, [PAR, BRZaH02, BHRZ03, BHZ04]) is a C++ library for the manipulation of convex polyhedra. Studies on previous polyhedral libraries (including POLYLIB, Section 3, page 54) result in this very complete and complex library, as well as an excellent bibliography. Although PPL's interface is supposed to be general purpose, it has been influenced by the needs of the static analyzers. That is why the library implements a few specific operators, while lacking some other operators that might be useful in other fields such as computational geometry. Interfaces to other programming languages, including C and a number of Prolog systems, are implemented.

An important and original feature of the library is the support for a new domain called the domain of rational convex polyhedra that are not necessarily closed (denoted NNC), which is separated from the domain of rational convex polyhedra that are topologically closed (denoted C). While computing NNC convex polyhedra may provide more precise results, closed convex polyhedra can be represented and manipulated more efficiently. When an NNC polyhedron P is necessarily closed, we can ignore the closure points contained in its generator system (as every closure point is also a point). Similarly, P can be represented by a constraint system that has no strict inequalities. Thus a necessarily closed polyhedron can have a smaller representation than one that is not necessarily closed. Moreover, operators restricted to work on closed polyhedra only can be implemented more efficiently. Thus using NNC polyhedra is only recommended if you can actually see an increased accuracy.

The list of supported operators is found in the PPL documentation. Here are some quick notes for this library:

- Written in C++, exception handling and memory management are implemented;

- Three different arithmetic modes are 32-bit, 64-bit, GNU Multi Precision.

CONVEX is a Maple package to facilitate computations in convex geometry [Fra]. It provides functions to deal with rational polyhedral cones, polytopes, general polyhedra, faces of one of the above, and fans of arbitrary dimension or size.

PORTA is a collection of routines for analyzing polytopes and polyhedra [TC]. The polyhedra are either given as the convex hull of a set of points plus (possibly) the convex cone of a set of vectors, or as a system of linear equations and inequalities.

POLYMAKE is a tool for the algorithmic treatment of convex polyhedra and finite simplicial complexes [GJ00, GJ01]. The system offers access to a wide variety of algorithms and packages within a common framework. POLYMAKE is flexible and continuously expanding. The software includes C++ and PERL interfaces.

QHULL computes convex hulls, *Delaunay* triangulation, half-space intersections about a point, *Voronoi* diagrams, furthest-site Delaunay triangulation, and furthest-site Voronoi diagrams [BDH96]. It runs in $2-d$ and higher dimensions. It implements the *Quickhull* algorithm for computing the convex hull. QHULL computes volumes, surface areas and approximations (when handling roundoff errors from floating point arithmetic) to the convex hull.

Composite Symbolic Library proposes a common interface that is used in Action Language Verifier, in order to represent boolean logic formulae by BDDs and linear arithmetic formulae by polyhedral representations [YKTB01]. The current implementation uses the CUDD and OMEGA libraries; an experimental version uses PPL.

4 Conclusion

Static program analysis based on abstract interpretation (Section 3.2, Page 10) uses abstract domains such as polyhedra, octagonal and interval domains. In this chapter, we focused on differences and problems concerning utilization of these abstract domains. Important definitions of convex polyhedra, octagons and BDDs were presented, as well as a list of existing implementations: static analyzers and their underlying libraries.

As we have seen, the rich set of numerical abstract domains is diverse. However, the more precise we want to be, the longer execution time we have to deal with. In exceptional cases, the more powerful abstract domain does not guarantee a better precision, because of magnitude overflow, lack of memory space or unacceptable execution time.

Static analyzers using these abstract domains are tuned in order to achieve the best trade-off between precision and efficiency, so using several domains is becoming a trend [AST, NBA]. Abstract domain for specific cases is also studied. The possibility of sharing implementations of abstract domains, from the simple domains (e.g. intervals) to the more precise domains (e.g. polyhedra, Presburger formulae), is of high interest.

In the next chapter (Chapter 4), we aim at a generic interface for the design of which adaptations to existing libraries are needed. Developments of new abstractions and libraries should take into account the interface, whereas a mechanism that permits easily changing abstract domains should be constructed.

A robust and efficient set of libraries, sharing a generic interface which allows management of exceptions and that eases parameterization of all the implementations, is to be studied. A complete set of benchmarks built from real applications is used, in order to analyze performances of these libraries. The polyhedral benchmarks is presented in Chapter 5.

Chapter 4

Towards a Multi-Domain Interface for Abstract Interpretation

In Chapter 3, we introduced several abstract domains used in static program analysis and their libraries. In this chapter, we design a common interface for those different domains and libraries.

Section 1 explains why a common generic interface for those domains and libraries is useful. It also contains the first comparison of existing interfaces. Our approach, described in Section 2, includes the description of an interface, called *HQ*, which was built based on our analysis of the compatibility problems raised by these interfaces. Then, in Section 3, we present two related projects: APRON, to which we contributed HQ, and the Parma project. The main decisions made for APRON are discussed, and some comments on the Parma project are given. We conclude by comparing HQ and APRON.

1 The Need for a Generic Interface

In this section, we try to identify all the existing problems blocking the way towards a framework in which codes written for different abstract domains can be reused between abstract interpreters. In the first subsection, we describe our motivation for a common interface with a short background. Then important issues are followed in the next six subsections, and in the end the section conclusion.

1.1 Motivation for a Common Generic Interface

In the middle of Figure 4.1, there are five static analyzers: PIPS [PIPb, IJT91], NBAC [NBA, JEA00], ASTREE [AST, BCC$^+$03], the OMEGA framework [Ome, PUG91] and CHINA [CHI, PAR, BRZaH02] which are introduced in Chapter 3, Section 2. On the left hand side of the API, we have the current status: each static analyzer has its own library [1]

[1] For the list of these libraries, readers are referred to Chapter 3.

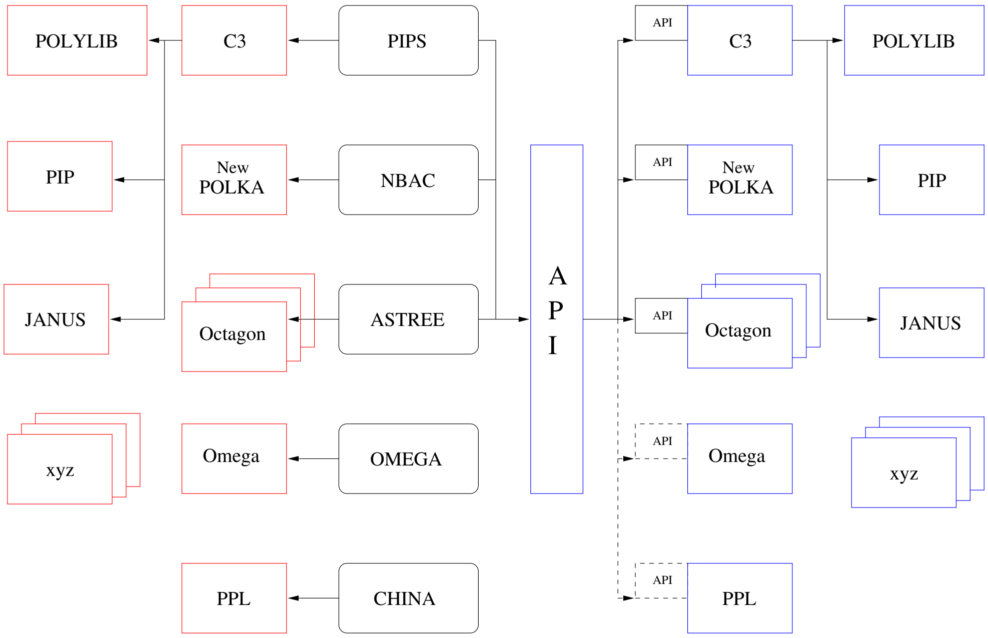


Figure 4.1: Why we need a common interface!

dealing with its own abstract domain(s). In practice, they all have problems when dealing with large scale applications.

However, recent developments from one team such as new abstract domains, e.g. the Octagon library [MIN, MIN01b], or algorithmic improvements, e.g. Cartesian factorization [HMPV03], cannot be readily exploited by other teams. Meanwhile existing static analyzers are mostly modular, and are mostly based on similar technologies.

On the right hand side of the API in Figure 4.1, we show how a generic API would let any analyzer benefit from specific features in other libraries. For example, this API would allow PIPS to use another abstract domain, e.g. octagonal domain or Presburger formulae, while keeping the infrastructure developed in PIPS for the polyhedral domain. Precisely, an unified framework could free its users from the compatibility burdens such as different signatures for the same semantic abstract operator, exception handlers and sometimes operator availability. Moreover, whenever a new improvement takes place somewhere for a given abstract domain, for instance the Cartesian factorization in PPL [HMPV03], then not only CHINA but also PIPS could profit from it without additional work. Finally, the new API should provide mechanisms to assure the robustness for available implementations, e.g. the timeout management.

As described in Section 3.1, page 81, a French project named APRON was launched in 2004 where such a common interface is of interest. Although OMEGA [Ome, PUG91] and Parma [CHI, PAR, BRZaH02] teams are not members of the project, the common interface must be compatible with their interfaces which are dot linked in Figure 4.1. Meanwhile, other teams have promised to support the new interface when it is finished.

In order to design an interface which can replace all interfaces already used in current implementations, first of all we need to study these interfaces and to find out common points and incompatibilities among them. Incompatibilities may happen at the interface level, that is to say the signatures of operators, the data structures, etc., as well as at the implementation level, with exception management, thread-safety features, underlying arithmetics, etc. We will discuss these problems later in this section.

At the interface level, different abstract domains lead to different data structures, then to different signatures for one generic operator. If we take the example of the octagonal and the polyhedral domains, we need an abstract object that represents an octagon and/or a polyhedron. Even for the same abstract domain, e.g. the polyhedral domain, many differences appear because each interface was designed to meet the need of their own developers. New POLKA was designed to be used in NBAC analyzer for automation analysis: a dense representation is chosen since the number of variables and hence the dimension is low, between 10 and 30 (see Chapter 3, Section 2.2, page 47 and Section 3, page 54). The C^3 library was designed for and used by PIPS which is an automatic inter-procedural analyzer. Hundreds of variables may be analyzed and a sparse representation is the logical choice (Chapter 3, Section 2.3, page 48 and Section 3, page 52). The octagon library is only used by ASTREE analyzer (see Chapter 3, Section 2.1, page 46).

In the following sections, we describe in a general way, the difficulties which have not been explored yet. Then, in Section 2, Page 68, we analyze them in greater details with respect to the API reference that was developed by the author. We begin with the comparison between polyhedral interfaces, and then the comparison with other abstract domains.

1.2 A First Issue: C^3, New POLKA, PPL and POLYLIB - Different Contexts

Being the most used abstract domain for advanced static program analyses, the polyhedral domain is implemented by several libraries containing many algorithmic improvements introduced over the years. However, existing polyhedral interfaces are not compatible as shown later in Section 2.3. Different choices were made by developers such as naming conventions, algorithms, exception handlers, etc. Due to their completeness, we choose to compare four polyhedral libraries namely C^3 [PIPa, CAI00], New POLKA [JEAb, JEA00], PPL [PAR, BRZaH02] and POLYLIB [LOE, Wil].

These four polyhedral libraries were designed and developed in different contexts for different languages C, C++, CAML. New POLKA and C^3 have been developed and used closely with their static analyzers, respectively NBAC and PIPS, as fundamental algebraic engines, while POLYLIB and PPL libraries are somehow independently developed. This leads to some particularities. For example, the PIPS analyzer does not use any widening operator, therefore its C^3 library does not implement this operator [2].

[2]This technique is described in the presentation of François IRIGOIN, September 2005 the 20th , which can be found at APRON's site [APRb].

New POLKA was formerly based on POLKA and POLYLIB implementation; however its interface and memory management is different from POLYLIB's. POLYLIB, designed for automatic parallelization, code transformations and code synthesis, used as a part of the C^3 library, does not provide a complete interface for static analyzers. For example, neither the manipulation of the dimensions of polyhedra nor the widening operator are not present in its interface. It however implements the computation of symbole Ehrhart polynomials and Z-polyhedra manipulations, which can be used for specific purposes.

In fact, the Ehrhart polynomials, which are a special kind of polynomials, form another abstract domain, where the set of integer points to be counted lies inside a union of rational convex polytopes, thus the number of points can be formulated by an Ehrhart polynomial. The Z polyhedra which are intersections of polyhedra with the integer lattice also form an abstract domain. In our work, we do not study explicitly these domains because of their given lower priority.

The PPL library, while being used by different projects such as the CHINA project [CHI, PAR, BRZaH02], the Action Language Verifier [ALV, BYK01], etc., will not have its final interface before version 1.0, according to the authors [PAR, BRZaH02]. Version 0.7 is the most recent version we had access to.

As a result, to construct a common interface for the above libraries, adaptations must be made for each and every library. In the next section, we consider a problem concerning not only the polyhedral implementations but all abstract domain implementations used in static analyzers.

1.3 A Second Issue: Control of Execution Time

We present here an example of the compatibility problems, not at the interface level but at a lower level. This problem is illustrated with the control of the execution times of analyses in PIPS (See Chapter 3 for PIPS and the libraries mentioned below).

Being an underlying component of PIPS, the library C^3 uses POLYLIB as an external library to perform polyhedral operations. To control the execution times of C^3, a timeout mechanism is added by the author. This mechanism requires modifications of certain functions of POLYLIB, e.g. Chernikova function. We here present several possible implementations and several functional interfaces. Then we propose a solution which we wished to implement in the POLYLIB's Chernikova function but we could not obtain a general agreement. This mechanism could be added to the magnitude control, i.e. integer overflow, which is already implemented in POLYLIB.

The pre-existing operators in the C^3 library, such as *boolean sc_feasibility_ofl_ctrl()* [3], *boolean sc_projection_ofl_ctrl()* [4] and *Psysteme sc_convex_hull()* [5], or POLYLIB's *void*

[3] The tests of satisfiability for a constraint system with overflow control, see Chapter 4, Section 7.

[4] It projects the constraint system along a dimension, with overflow control, see Chapter 4, Section 8.

[5] It computes the convex hull of two constraint systems, see Chapter 4, Section 10.

chernikova() [6] that is used by C^3, did not have a timeout mechanism properly established. Some analyses can last a long time whereas the abstract interpretation enables us to sacrifice the precision to obtain speed. The activation of this timeout mechanism in C^3 with the example ocean.f [7], has grealty reduced PIPS execution time, from three days to three hours with TRANSFORMER_INTER_FULL, PRECONDITIONS_INTER_FULL, MUST_REGIONS analyses [8].

Moreover, the execution times of some algorithms can be very sensitive to non-semantic modifications of the parameters, for example the order of successive projections of a set of variables. The exponential complexity of some operators can yield durations of some tens of seconds to hours, with no differences in the final result.

If an operation may last a long time, we want to put a timeout so that its execution finishes within a time limit set by the programmer. But we also want to preserve the current functionality. That is possible using one of the two solutions as follows:

- Implementation of an "alarm" call, which will stop the operation when the timeout is reached. The routine handling an exception is launched by the means of a longjump and can position a flag which is tested later on in the operator;
- Modification of POLYLIB so that it supports timeout management.

For the call of a *throw_exception* [9] using a longjump, we have a problem. If the main process is executing a *malloc*, i.e. allocation of memory, or a system call, the memory context is probably incoherent, which can lead to a core dump.

This first approach has however an advantage: it is not necessary to modify the operator implementation itself. It is enough to add a layer of wrapping. If in the process, there is no malloc or system calls, we can take the risk to use a longjump. If there is, we must modify the code somewhere in the operation, for example the main loop. The new code should take into account the existence of the flag. The disadvantages are: 1, a modification of the existing code which can be in an external library, such as for example POLYLIB; 2, a slightly slower algorithm since it is necessary to test the flag for timeouts in one, or several, main loops of the initial algorithm.

One can directly use a *throw_exception* in the case where there is no malloc, by releasing the memory used in CATCH. For the convex hull operator with many calls to malloc, or the satisfiability test using the simplex method with memory allocation of a hash table, we must modify the code.

The implementation of the additional code can be hidden by a mechanism of #ifdef which allows us to generate two versions of the original operator. This complicates the

[6] POLYLIB's implementation of Chernikova's algorithm that computes the polyhedral dual conversion, see Chapter 4, Section 6.

[7] see Chapter 5, Page 127.

[8] However, we did not study the output of PIPS in these two cases.

[9] A C implementation whose concept is similar to Java's throw/catch mechanism.

non-regression tests but guarantees the absence of changes for those who do not need the timeouts.

The second solution, which requires an activation of the timeout mechanism, is preferable. By default, the timeout mechanism is not active. To allow the use or not of the timeout, there are at least two possibilities in the implementation: 1, specify the timeout by using one or more variables of the environment, i.e. one always uses the same signatures of operators; or 2, establish a new interface that takes into account the timeout as additional parameters and additional pieces of code if a timeout took place. With 1, the old interface is preserved and used in a wrapper which can activate or not the mechanism of timeout. With 2, the old operator is modified to accept the necessary additional parameters.

However, POLYLIB does not support this approach in general [10]; in this way we can see the *human-factor* complexity of the problems. Consequently, in order to use the timeout in *Psysteme sc_convex_hull* [11], we must choose the first solution. POLYLIB has the same mechanism for exception management using TRY/CATCH as the C^3 library, which is used at CRI, thus the solution was implemented by the author without difficulty.

As mentioned above, in this chapter we propose a new interface (Section 2), and then compare it to existing interfaces. However, POLYLIB is not considered [12], since its interface is a subset of the other four that we have chosen which are C^3's, New POLKA's, Octagon's, whose interface was based on New POLKA's, and PPL's. In the next section, we present a comparison between the octagonal and the polyhedral interfaces.

1.4 A Third Issue: Octagons vs Polyhedra

The octagonal domain is recent [MIN04a], although similar ideas were used to speed up convex array region analysis [HK91]. It was introduced to avoid the high complexity of polyhedral operators without losing too much accuracy. Because it is a simpler domain, it is less precise but offers a shorter run time when used in static analyzers. Indeed, its operators execute faster with polynomial instead of exponential complexity.

The current implementation of the octagon library has an interface that is incompatible to existing polyhedral interfaces, even though this library is used in ASTREE [AST, BCC+03] in order to replace the more expensive polyhedral domain. However, since every octagon is also a polyhedron [13], we expect to use the octagonal domain instead of polyhedral one without major difficulties.

While the dual conversion with Minkowski representation is very important for polyhedral implementations, e.g. the most efficient algorithm for the convex hull operator implements this approach, it is not vital for the octagonal domain. We can compute the

[10] We posted this solution on the POLYLIB's mailing list, and the response was negative.
[11] It computes the convex hull of two constraint systems, see Chapter 4, Section 10.
[12] We do not consider the Ehrhart polynomials here [Cla96].
[13] In 2-D, it is a polyhedron of at most eight edges.

generating system of an octagon, by some algorithm derived from Chernikova's one, for example, but it does not imply a better performance.

When we study some missing operators, i.e. the functions that are available in the polyhedral interface but not in the octagonal interface, e.g. the dimension permutation operators, fortunately no algorithmic problem has been found yet: we can implement these missing operators without difficulty. Nonetheless, for some operators such as the closure operator, implementation efficiency suggests some differences in levels for these interfaces, since low-level oparations are required.

In fact, we can find some similarities between the Octagon library's interface and New POLKA's, since the former was built based on the latter. We even find the operators that convert an octagon to a polyhedron, and vice versa, which indeed help the move towards a common interface. But it is hardly enough: we cannot use them interchangeably, thus they are not compatible. Fortunately, the participation of the two groups in APRON project (Section 3) will make it easier to adapt to a new common interface.

We consider in the next section a problem with the Octagon library's interface, because dimension permutation operators are not available [14].

1.5 A Forth Issue: Variable Assignment

In this section, we present an example that shows one of the difficulties encoutered to use the Octagon library in the PIPS analyzer. We consider two interfaces, one of PIPS's C^3 library and the other of the octagon library [MIN, MIN01b].

In order to keep a generic architecture for PIPS, the *oct_t* oct_assign_variable()* operator which models the assignment command should belong to a higher level than the intersection or union of octagons. It is because the assignment command is independent from the abstract domain used, which can be, for instance, octagonal or polyhedral.

Moreover, a wrapping for octagon is preferable to modification of the existing interface. Accordingly, we have tried to implement the operator *oct_t* oct_assign_variable()* from octagon's lower level primitives, i.e. from the API of the library without *oct_t* oct_assign_variable()* itself. Retrospectively, it is not the best approach to deal with the problem since there are other operators in PIPS that are more interesting to implement using the octagons, but it is already complex. In fact, we encounter three problems:

- If we encode $I := 1$ and $J := 2$ independently, as we do in PIPS's bottom-up approach for transformers, I and J are represented by the first dimension in their corresponding octagon. Therefore, when we want to combine these two, we have to modify the binding of variable names to dimensions and to modify at least one octagon.

[14] C^3, New POLKA and PPL have dimension permutation operators but not POLYLIB. Thus, POLYLIB has this problem, too.

It is even worse when we have to translate a function call like $I = f(K)$, assuming f's transformer known.

- If we want to combine $I := I + 1$ with the transformer for $I := 1$ (see Chapter 2, Section 4.2, Page 15 for transformers), we have to play, at least temporarily, with the number of dimensions by adding something to represent the old value, the temporary intermediate value and the new value of I.
- In the operator *oct_t* oct_assign_variable()* of the octagon library, sometime non-octagonal constraints may appear. In this case they are approximated by their interval bounds. This kind of heuristics should not appear at a high level, which stops us from rewriting *oct_t* oct_assign_variable()* in a generic way.

To make it short, we need an operator to rename variables to solve the first two problems. For the third problem, we should find another way to approximate the result using only the API.

Finally, we notice that though POLYLIB is used in C^3, the only used part is the implementation of the Chernikova algorithm. The variable assignment is one of reasons that stop us from exploiting more effectively this library.

1.6 A Fifth Issue: Omega's Presburger Formulae vs Polyhedra

The domain of Presburger formulae can provide more accurate information than the polyhedral one in static program analysis. For example, dependence analysis using polyhedra is structured by Pugh as a decision problem: test simply answer yes or no [15]. Meanwhile, using Presburger formulae, i.e. the OMEGA test, one can construct a set of constraints that describes all possible dependency distance vectors which can be used directly in deciding the validity of program transformation [PUG91].

Presburger formulae contain affine constraints, the usual logical operators, and existential and universal quantifiers. The OMEGA test cannot simplify all Presburger formulae efficiently: there is a 2^{2^n} non-deterministic lower bound and a $2^{2^{2^n}}$ deterministic upper bound on the time required to test Presburger formulae satisfiability, where n is the length of the formulae.

Despite their worse case complexities, some algorithms implemented in Omega library are worth to be considered. For example, exact integer projection is implemented in Omega [PUG91], whereas C^3's algorithm only detects exactness when the exact projection is a polyhedron: A projection of a polyhedron along a variable is *exact* on an integer set if the existence of an integer point in the polyhedron after the elimination of the variable implies the existence of a corresponding integer point in the initial polyhedron.

The Omega test uses the *dark shadow* method, which is an extension of Fourier-Motzkin method; hence its complexity is exponential. However, in practice, for dependence

[15] We can indeed return a polyhedron of dependences.

tests, its time complexity is polynomial [PUG91]. The C^3 implementation, using three sufficient conditions defined in [AI91] and in [PUG92], performs the exactness test with lower complexity, while retaining a good percentage of exact responses in experimental results: the exactness rate of the dependence test developed in PIPS for the PerfectClub benchmark is 97, 95% (see [YAN93], page 70).

We have seen in Chapter 3, Section 1.5, page 41, differences between interfaces for the Omega library and polyhedra. For example, in Omega there is a structure which represents either a relation R, or a set S. This relation can be used to model a transformer (Chapter 2, Section 4.2 Page 15), which belongs to a higher level than the polyhedral domain. Therefore it is necessary to define different generic levels to accommodate all existing interfaces.

1.7 A Sixth Issue: Finite Union of Polyhedra

Arnauld LESERVOT, in his dissertation, has used the domain of lists of polyhedra [LES96]. Starting from the fact that all Presburger formulae can be represented in Disjunctive Normal Form (DNF) or Conjunctive Normal Form (CNF), an element of the domain can have two representations based on convex polyhedra.

The set of lists of polyhedra, with its operations like union, difference, inclusion, equality between two elements, emptiness test as well as conversion from one form to another, forms an abstract domain, which has been implemented in PIPS for the computation of Array Data Flow Graphs [LES96]. It is only used in PIPS for exact convex array region computation [CI96], and its operator signatures are unfortunately not compatible with polyhedral ones. For example, the satisfiability test does not use homogeneous names for functions: the disjunction of constraint systems has the function *boolean dj_empty_p*, whereas a constraint system has the function *boolean sc_feasibility_p* and *boolean sc_empty_p* has a different semantic [16].

As mentioned before, this interface is not taken into account for HQ. This abstract domain is less powerful than the Presburger formulae since it cannot manipulate infinite sets. Moreover, no experimental results comparing the performances between the Omega library and the implementation for lists of polyhedra [17] are available.

1.8 Conclusion

We have presented our motivations for a common generic interface, which could be used in static analyzers. Then, we have briefly compared six interfaces. We did not consider the interface of the interval and BDDs domains here, since those two domains are used in a different way (see Chapter 3, Section 2).

As we have seen, there exist many problems towards a common interface, even if

[16] It simply verifies whether the given constraint system is the constant SC_EMPTY or not.

[17] Which can be found in the package union of the C^3 library.

we study only the polyhedral domain with its existing polyhedra implementations (Section 1.2), or with its closest domains, the octagons (Section 1.4) and the Presburger formulae (Section 1.6). These problems are:

- different signatures;
- different levels of operators;
- exception handlers.

In the next section, we present our approach to analyze in greater details and then to deal with these problems. The Parma library PPL also proposed a general-purpose interface and implemented it. However, in our point of view, it is not satisfying (see our related work discussion in Section 3.2). Therefore, the construction of a prototype of a new interface seems to be required if we want to move forward toward our goal. APRON, another approach with a prototype developed by Bertrand JEANNET, which is also presented in our related work, Section 3, is based on our approach.

2 HQ Interface

In this section, we present our proposition of an interface called HQ, whose name is abbreviation of *Hieu Qua*, which stands for effectiveness in Vietnamese.

2.1 Introduction

HQ's interface is designed by taking into account four existing APIs: C^3, New POLKA, PPL and Octagon. The first three are polyhedral and the last one is octagonal. However, since an octagon is in fact a special case of a polyhedron, we can consider the four APIs as polyhedral. We also take into account the requirements needed for Omega library, the list of polyhedra implementation Section 1.7.

The idea behind this interface is to build a library to manipulate sets. Every polyhedron represents a set of points; so does every octagon, every Presburger formula and every interval. Since all these *sets of sets* form an abstract domain, which is more abstract than some others, we can provide the most abstract and basic manipulations, while hiding problems concerning differences among those abstract domains.

HQ is defined using *javadoc* utility, which permits an easy view of the API, even though the *C* language is favored by most of the projects. Our first intention was to generate a *C* version of HQ using another tool (JNI stands for Java Native Interface). However it soon becomes inappropriate: generated signatures are not human readable since there is always a Java context-related object in the signatures, as well as a long JNI prefix, which is not necessary nor user-friendly. We notice here that while a set of translation rules can be designed in order to automatically produce the equivalent C signatures, we have not created these rules yet.

HQ	Polyhedra	Octagons	Unions	Omega	Intervals	BDDs
HQSet	Polyhedron	Octagon	Union	Pres. formula	Interval	BDD
HQSysCon	ConSys	NA	ConSys	Pres. formula	NA	NA
HQConstraint	Constraint	Oct_elem	Constraint	Constraint	NA	NA
HQSysGen	GenSys	NA	GenSys	NA	NA	NA
HQGenerator	Generator	NA	Generator	NA	NA	NA
HQVariable	Variable	Var_t	Variable	Variable_ID	Variable	Variable
HQExpression	Expression	tab	Expression	Relation	Expression	Expression
HQDimension	Dimension	Number	Dimension	Variable_ID	Variable_ID	NA

Table 4.1: Comparison of abstract objects

As a consequence, HQ's documentation [QUEa] only serves to identify incompatibilities: the name of operators, the arguments of the operators, which operators are missing, where to put operators, i.e. levels of the API, etc.

Since written in Java, an object-oriented language, some of the implementation details are hidden, such as exception management, which is intended to simplify the interface problem. In fact, we divide it into two parts, one for the imperative signature, the other for *fragma* [18] decisions. Therefore, along with the API, another type of document is provided in order to deal with those problems.

We mostly focus on the incompatibilities among existing libraries. In our documentation [QUEa], the operators of the HQ set are discussed along with its corresponding operators available in the four libraries. However, in the following sections, we only study some important operators such as the satisfiability, projection, minimization and convex hull operators. For each operator in HQ, the names of the equivalent operator in C^3, New POLKA, Octagon and PPL if it exists are provided.

2.2 Presentation

HQ Classes: HQ is designed using Java with classes representing abstract objects. Six abstract domains' *equivalent* objects to HQ's classes are summarized in Table 4.1, knowing that non-relational domains break most of transformer analyses. The NA stands for not available.

The HQSet class represents abstract sets. An instance of the HQSet class, henceforth called HQSet for short, is *equivalent* to a polyhedron, an octagon, a set of Presburger formulae, or a list of polyhedra, i.e. a union. Each HQSet represents a set of points that belongs to a multi-dimensional space. Each dimension space can be represented by different names or different numbers. Those dimensions are associated to variables; thus every dimension is sometimes considered as a variable, depending on the context. Each dimension is represented by an instance of the HQVariable class. The space and the

[18] Fragmented issues which are not structured, concerning concrete implementations.

mapping between dimensions and variables are abstracted by the HQBasis class.

We have decided to define the HQBasis class for many reasons, but the most important one is that it permits a flexible dimension name management. As such, when program variables are passed from the analyzer to its abstract domain engine, that is to say our HQ implementation, their names are forwarded, too. This helps not only the debugging at the engine's level but also at the analyzer's level, since the origin of every variable is known. Precisely, the HQBasis class is similar to C^3's *base* object, which permits an easy debugging since variable names as strings are given by the analyzer PIPS instead of fixed numbers, as in other libraries.

The HQConstraint class represents an abstract constraint. Every abstract constraint is affine and equivalent to a polyhedral constraint, or an octagonal constraint, or a Presburger formula, with the comparative connectors such as $<$, $\leq$, $>$, $\geq$ and $=$ [19].

Equivalent to an affine expression is a HQExpression, which is in fact a HQConstraint but without any comparative connector.

Since a polyhedron can be represented by a constraint system or a generating system, we consider that an octagon or a set of Presburger formulae can be properly represented by an instance of the HQSysCon or/and HQSysGen class, where each HQSysCon instance consists of several HQConstraint instances, and each HQSysGen instance consists of generating elements of type HQGenerator.

Scoping: The HQSet class only *sees* the following classes: HQSet, HQSysCon, HQSysGen, HQBasis, HQVariable, HQExpression. Therefore, its relations with HQConstraint and HQGenerator must be exploited through getHQSysCon() and getHQSysGen(). This accessibility has the advantage to reduce the connexion between HQSet and HQConstraint. Note here that in the octagon library, we have octagons and constraints (binary and linear) manipulation, but not vectors manipulation.

Conventions: For the sake of simplicity, we have presented operators such as *void add_dimension* [20] or *void remove_dimension* [21] with only one parameter. However, as discussed in our documentation [QUEa], the several parameters support is suggested since it improves the performance of these operators at algorithmic level, e.g. convex hull.

We propose in HQ interface some additional operators that can be considered redundant such as re-map and sort for the HQBasis class. We also introduce the getSize() operator, which is an abstract size of a HQSet. This *size* of a HQSet should provide heuristically defined information about how long the principal operators might need to finish their task. We suggest as well to unify the debugging functions, such as those in C^3, where we have *boolean sc_consistent_p(Psysteme)* or *boolean sc_weak_consistent_p(Psysteme)* [22].

[19] For Presburger formulae, we also have the existential $\exists$ and universal $\forall$ quantifiers.
[20] This operator adds a new dimension to the constraint system.
[21] This operator removes a dimension from the constraint system.
[22] These two operators verify whether the constraint system in question is valid or not, due to possible

The HQ interface is proposed after having studied the other four interfaces; as a result, for adaptations, wrapping functions are suggested. However, as we have seen in Section 1.5, wrapping functions for Octagon are not yet possible since dimension permutation operators are not available. The example with timeout management for POLYLIB (Section 1.3) is another example that shows how hard this work can be.

In the next section, we discuss some important operators in detail. For the complete documentation of HQ, readers are referred to [QUEa].

2.3 Case Studies

In this section, we discuss in more details problems listed in the conclusion of Section 1 and related to some important abstract operators. There are two main categories of problems: the differences of signatures between existing implementations for an operator, such as naming conventions, returned codes; and the availability of some implementations. We begin with an easy operator that returns a boolean answer, the test of emptiness of a HQSet, i.e. the satisfiability of the constraint system used to define it.

2.3.1 Emptiness Test and getStatus

Description: The *emptiness* test, also known as *feasibility* or *satisfiability* test in case of polyhedra, verifies if a HQSet represents an empty set or not. A HQSet over D^n is said to be *not empty* when it contains at least one element in D^n, or *empty* if it contains no element.

The emptiness test of a rational polyhedron by means of its generating system is equal to the satisfiability test of its constraint system, since we have the duality of the two representations [23]. A constraint system is said to be *satisfiable* or *feasible*, if all its constraints are simultaneously satisfied or feasible; on the contrary, if we detect a contradiction among its constraints, it is said to be not satisfiable or *infeasible*.

Sometime this operator can be referred as, although it is not much in use yet, *is_bottom* test of an element. This is due to the fact that when we consider the lattice of the abstract domain, we call the empty HQSet the bottom element of the lattice, and the HQSet that spans all the space, the top element.

The emptiness test for a HQSet normally returns a boolean answer, whereas its semantics is different for integer and rational elements. The complexity of available algorithms is usually exponential. Indeed, it returns the answer *not practically calculable* when an exception is raised, due to computing complexity. In this case we cannot prove whether the HQSet is empty or not.

The emptiness test has only one integer implementation, JANUS [SOG], adapted by the author to be used in C^3. Therefore, existing implementations for rational and integer

programming errors.

[23] The Chernikova algorithm performing the dual conversion is rational, not integer

```
 HQBoolean getStatus(HQSetInterface.HQSetStatus status)
/*
 Valid status are CONSTANT_EMPTY, CONSTANT_UNIVERSE, UNDEFINED,
NOT_EMPTY, BOUNDED, BOUNDED_FROM_ABOVE, BOUNDED_FROM_BELOW,
CLOSED, MINIMIZED, NORMALIZED.

 Status BOUNDED can have answer TRUE, FALSE, TOP, i.e. bounded from
above, or under a hyperplane, and BOTTOM, i.e. bounded from below,
or above a hyperplane.

 Status CONSTANT_EMPTY, CONSTANT_UNIVERSE, UNDEFINED and CLOSED can
be TRUE, FALSE.

 Since expensive calculations of NOT_EMPTY, or SATISFIABLE, test can
generate exceptions, we can get TOP or BOTTOM answer instead of TRUE
or FALSE.
*/
```

Figure 4.2: Signature of getStatus

problem are:

- Only rational operators: New POLKA, PPL;
- Integer when possible, otherwise use rational operator: C^3.

About exception handlers, existing solutions using C language, with magnitude overflow and out-of-memory space are:

- CATCH/THROW mechanism: C^3;
- Integer returned codes which indicate exceptions: PPL;
- No exception handlers: New POLKA.

In the next section, we present our prototype signature for the emptiness test, which in fact can be used for status-related queries.

Our Proposal: getStatus In HQ, we consider the emptiness test of a HQSet is a query for the empty status of this HQSet. Thus the getStatus operator, whose signature is printed in Figure 4.2, permits testing the emptiness of a HQSet.

Constants terms: In fact, this operator signature permits testing not only the emptiness but also other attributes of the HQSet. We remark that some constants such as CONSTANT_EMPTY, CONSTANT_UNIVERSE, UNDEFINED are introduced. In the polyhedral case, they respectively represent an empty set, e.g. with only the constraint $0 == 1$, all the space (R^n), with no constraint, and an undefined HQSet object.

The undefined HQSet object is not available in PPL, Octagon and New POLKA but only in C^3. In C^3, there is a distinction between the pointer to an undefined constraint system and a standard ANSI C pointer. It is because sometimes programmers want to know whether the returned object is an empty or undefined constraint system [24].

Various existing implementations: For the bounded from above or from below status, we have seen the equivalent notions only in PPL. Besides the emptiness test *int ppl_Polyhedron_is_empty(ppl_const_Polyhedron_t ph)*, we have functions that are not available in C^3, such as *int ppl_Polyhedron_is_bounded (ppl_const_Polyhedron_t ph)*, *int ppl_Polyhedron_bounds_from_above(ppl_const_Polyhedron_t ph)*, or *int ppl_Polyhedron_bounds_from_below(ppl_const_Polyhedron_t ph)* [25].

PPL uses the notation *get_relation_with_*, which permits specifying many relations such as inclusion, equality, disjoint, strictly or not, etc. This notation is not used in the other libraries. In New POLKA, the notation *_versus_* is used. This notation is however in our opinion not intuitive enough.

Clarity of terminology in use: We also have other examples about the clarity of terminology in use. It is not clear how to define and use minimization, normalization, simplification in existing implementations. Similarly, the closure operator in the octagon library does not make much sense in the polyhedral domain. For more explanation, one is referred to [APRa], where the term canonicalization is also introduced.

In Octagon, we have *bool oct_is_empty(oct_t* m)*, *tbool oct_is_empty_lazy(oct_t* m)*, that test the emptiness of the octagon in question. *bool oct_is_universe(oct_t* m)* tests whether the octagon is the constant universe or not. *bool oct_is_closed(oct_t* m)*, is a low level function that tests if the octagon is closed or not, meanwhile all the polyhedra are closed (see Chapter 3, Section 1.3, page 34 for the octagonal domain).

In New POLKA, we have *bool poly_is_empty (const poly_t* po)* and *tbool poly_is_empty_lazy (const poly_t* po)* that test the emptiness of the polyhedron, *bool poly_is_universe (const poly_t* po)* that tests if the polyhedron is the constant universe.

Here, *bool* is the standard boolean type with two values, true and false, whereas *tbool* stands for the triple true, false and top. The top element serves for exceptional cases, i.e. do not know.

[24] The empty constraint has a semantic meaning, whereas the undefined constraint system is simply used as a programming trick. For example, the undefined constraint system can be used to indicate the case where an exception occurred.

[25] These three functions verify whether the polyhedron is bounded or not.

Incompatibilities: In C^3, the implementation of the emptiness test is complicated. The *boolean sc_empty_p()* and *boolean sc_rn_p()* functions quickly test whether the constraint system is the constant sc_empty or sc_universe [26], whereas the full rational and integer tests, which implement several algorithms such as Fourier-Motzkin, Simplex and cutting plane methods (see Chapter 4 for these methods), are also available. These functions are: *boolean sc_feasibility_ofl_ctrl(Psysteme sc)*, *boolean sc_rational_feasibil ity_ofl_ctrl(Psysteme sc)*, *boolean sc_integer_feasibility_ofl_ctrl(Psysteme sc)*. Inside these functions, heuristics are implemented for the selection of algorithms, which are based on the number of the constraints of the polyhedron in question.

It is somehow simpler in the other three libraries: there are no integer or rational signatures, no quick test signature and there is only one algorithm for this operator. Nonetheless, there are two different versions of the emptiness test which are not available in C^3 but in the others: in Octagon, we have *bool oct_is_empty(oct_t* m)*, *tbool oct_is_empty_lazy(oct_t* m)*; in New POLKA, we have *bool poly_is_empty (const poly_t* po)*, *tbool poly_is_empty_lazy (const poly_t* po)*. The lazy version is used in order to delay the closure computation, which is rather expensive. The similar technique is also applied in PPL with the *and_minimized* version.

Which value to return at the end of a function's execution also raises a compatibility problem since New POLKA uses the triple *true*, *false* and *dontknow* in case of exceptions. This links to its chosen policy for exception handling; meanwhile C^3 uses *true* for semantic true and *false* for *dontknow* [27]. We notice here that *sc_empty_p* is semantically different from not *sc_not_empty_p* because of exceptional cases. It then needs a *is_not_empty* test, knowing that *not_is_empty* is different from *is_not_empty*. We can also use *is_known_not_empty* and *is_known_empty*.

We remark that each implementation has its own naming convention; for example in C^3, *is_xxx_p* is different from *is_xxx* where the suffix *_p* is used for boolean test, p stands for predicate; in New POLKA, suffix *_t* is used for type, etc.

In the same way, the mechanism handling exceptions changes the operator's signature: some have exception handlers, e.g. C^3; some do not, e.g. New POLKA. The PPL library in its C interface proposes the standard returned codes of integer type, which tells the status of the operation: result or exception code. Otherwise, we can deal with internal errors, i.e. at lower level than the interface, while hiding exception-related issues from the interface's signatures.

When dealing with multiple algorithms for one operator, beside the interface problems, we expect to see some algorithms destroy the inputs by side effects, while some behave functionally. For example, there are several algorithms for solving the emptiness test, among which the Simplex and the Fourier-Motzkin methods. The former builds a hash

[26] In C^3, the term R^n, the whole space, is used instead of the constant universe.

[27] The NOT_EMPTY term is chosen for unknown answer, so the approximation step is hidden inside the operator.

table and then works on this table without modifying the given input which, in this case, is a constraint system, whereas the latter performs transformations of the corresponding constraint system, hence modifies its input. For this reason, we may have destructive and/or non-destructive functions. If we make a copy of each input, which helps the debugging process inside the library [28], the copying will penalize the overall speed performance.

Then the question of exposing or not the destructive functions (PPL uses the name recycle) directly impacts the interface, as well as the memory management by reference counters. However, weIn HQ, we have decided to use the destructive function alongside with the standard one.

Conclusion: we have seen that even the simplest operator raises lots of problems. Moreover, these problems are not specific to this operator but other operators.

2.3.2 Projection

Existing implementations for the *projection* operator and its sibling, the *add* dimension operator, also have many incompatibilities.

In C^3, a space is represented by a structure called *base*. A base can be associated with a polyhedron whenever it makes sense. To remove a dimension from a polyhedron, we remove it from the base and from the corresponding constraint system. To add a new dimension to the polyhedron, we simply add it to the base, by the function *void base_add_dimension(Pbase b, Variable v)*.

In PPL, Octagon, and New POLKA we have two versions: the first one adds a new dimension to the polyhedron and the variable corresponding to this dimension is not constrained. The second one with the suffix *_and_project* sets the variable value to zero.

New POLKA, besides the two above versions, offers the ability to add several dimensions at once.

As a consequence, for this operator, the interfaces are different. Of course, in C^3 we can obtain the *and_project* version by calling the projection operator after having added the new dimension, or we can add several dimensions in C^3's, PPL's, Octagon's implementations by calling several times the same function that only adds one dimension. It is clearly just a matter of choice.

In the very same way, the New POLKA, Octagon and PPL libraries propose to remove several dimensions at once with destructive option (Octagon), or with higher dimensions option (PPL, New POLKA). Sometimes the projection along a list of dimensions depends on the order of the dimensions; therefore this version may be algorithmically useful [29]. We emphasize here that the version that permits several arguments at once is useful (an example of the convex hull operator can be found in Page 77).

[28] For example, we can print out the constraint system if an exception is raised.

[29] However, we do not have any result concerning this issue.

Similar to the emptiness test, there are several versions of the projection operator in C^3, implementing algorithms of different complexities with the sparse representation. For all these differences, we need to unify these interfaces, by means of a common interface.

2.3.3 Minimization

We now consider the minimization operator, which basically suffers from the very same incompatibilities as the operators described in the two previous sections. However, we do not repeat ourselves here, but present only one minimization specific incompatibility.

In fact, the three polyhedral implementations are in essence optimized by the user using sequence of calls, since it can be very expensive to apply the minimization operator everywhere, whereas the octagon's minimization is hidden and integrated in its algorithms. The consequence is then that the polyhedral operators can have the suffix *_and_minimize*, or not at all, which raises a compatibility problem.

We have chosen above to present an operator-related point of view, and three operators as concrete examples. The contents of the three corresponding sections are not balanced since we do not want to repeat common problems. We sum up the problems in the conclusion part, Section 2.6. The next section discusses the availability of some operators.

2.4 Missing Operators

In this section, we list the operators that are available in some implementations but not in other implementations, as well as some recently proposed operators. In general we do not know why they are not available, except some special cases. We explain why some operators are important so that we suggest to implement them.

Since C^3, New POLKA and Octagon are libraries that are developed closely and used alongside with their analyzers, some operators which are not yet needed are not implemented. PPL library however proposes a very complete interface, which is designed for general purposes. For example, the operators corresponding to the *int is_disjoint_from* [30] test, *int expand_dimension* and *int fold_dimensions* operators [31] are available only in PPL.

For debugging purposes, only PPL and C^3 offer functions for the consistency check *int is_OK* operator, which is necessary. The mapping function that swaps dimensions is not available in Octagon. Also, polyhedral common objects such as the constraint and generating systems are not available in Octagon. Only in Octagon do we have low level access functions to its elements, which are octagonal constraints; although there are no generating systems, we can always compute them in theory.

[30] Test whether the polyhedron is disjoint from another.

[31] Grouping and separating similar dimensions of the polyhedron.

Not available in C^3's interface are the operators with *_and_minimized*, or *_lazy* versions, which perform the minimization inside the main operator. This is important at the algorithmic level since the minimization itself is an expensive operation which strongly depends on the size of its input. Let us consider an example by comparing the two approaches with the intersection operator: the first one computes the intersection of two given polyhedra to obtain a new polyhedron, then applies the minimization on this polyhedron; the second one applies the minimization on each of the two polyhedra, then computes the intersection of two minimized polyhedra. In the first case, if the result is a very large polyhedron, the minimization may need a very long time to execute, or exceptions may appear. The second case gives a more stable running time for average size inputs since they can be minimized faster.

At the same time, the absence of a version supporting a list of arguments for operators such as the projection or the convex hull in C^3, PPL and Octagon can penalize the performance of their implementation. For efficiency reasons, this is necessary since the computation of convex hulls is as follows: Consider three constraint systems A, B and C. If we compute the convex hull of A and B to obtain a constraint system D, in order to pass it and C to the same operator, we have to compute two more conversions to generating systems, which is not necessary because we can indeed merge several generating systems together into the final one, and then convert it to constraint system form. Moreover, since it is an expensive operator, we can have several possible approximations, thus we have to decide how to choose a specific version, for example, with an additional argument.

It is worse for with the widening operator, where a standard version is proposed in [HAL79], and another version is proposed later in [CH78]. These two versions are semantically different. In the PPL library, we have two other versions whose names are used in the signature of their function.

Moreover, we prefer having access to the numbers of equations and the numbers of inequalities by getNbInequalities and getNbEquations operators as in C^3 instead of having access only to the numbers of constraints as in PPL and New POLKA, since experience shows that heuristics can be based both on the numbers of equations or/and the numbers of inequalities (see Chapter 5).

We also propose new operators such as getAbstractSize or getAbstractWeight for the HQSet objects, which should be used to predict the running time of main operators on these objects. We also need to unify the printing functions, as well as the format conversion functions between different abstract domains and between different implementations of the same domain. New operators such as *weak_update* (see Section 3), *elapsed_time* operator (see [MIN, MIN01b] and [PAR, BRZaH02]) should be available in our interface.

2.5 Other Problems

In Section 2.3, we have studied the problems directly related to some important operators, which can be categorized into two parts:

- Signature-related problems between implementations of each abstract operator;
- Availability of some implementations.

In Section 2.4, we have discussed the availability of some operators. In this section, we discuss other problems that are domain-related, signature-related or implementation-related.

2.5.1 Domain-related Problems

As we have seen, the decision of implementing the HQBasis class or not raises many incompatibilities among the four interfaces; for instance, for generating system management, problems such as: when do we compute the HQSysGen of a HQSet if it is included in, do we have full support for access to constraints like in C^3 and PPL, or just low level functions like in Octagon or matrix-based functions like in New POLKA?

As an example, let us consider this problem: we choose HQSet$-$ >HQSysCon$-$ >nbIneq or HQSet.getNbIneq. If we used the second approach as currently do the four libraries, that is to say we expose the getNbIneq function, which returns the number of inequalities inside the HQSet, in the HQ interface, we somehow expose the nature of the HQSet class. In our interface, we decided not to use the second function because we want the HQSet to be as abstract as possible.

Another problem is that, when we call this function, we do not know whether the HQSet is minimized or not. Some transformations of the HQSet can result in different values of its number of inequalities. Therefore, for every operator, we need to pay attention to its detailed semantics.

As we have discussed in Section 1.4, problems concerning transfer functions are also not trivial. We need to justify our decisions for many other problems such as accessibility scope for objects like vectors, constraints, generators, constraint systems, generating systems, matrices, undefined objects, multiple arguments for the union, projection operators, object versus list of objects for constraints, generators, sets and expressions, signature in imperative mode with several effects, i.e. using returned code in PPL.

Another abstract domain introduced in PPL, the Not-Necessary-Closed, denoted *NNC*, is not considered in our work [BRZaH02].

2.5.2 Signature-related Problems

By the design of HQ, we also deal with, though not directly, problems of approximations. For each operator, we may have several implementations among which the approximate ones; thus we need a parameter for tuning between their precision and speed.

We have to deal with the differences among float, rational, real and integer computations; robustness, i.e. how to deal with exceptions; thread-safety, i.e. problems of when using threading; how to choose abstract domains using the common interface; or how to switch between the available abstract domains.

The proposed common interface for C^3, New POLKA, PPL and Octagon requires full support for every operator, thus operators being not ready are expected to generate exceptions when they are called. Furthermore, each library can implement and use its own specific operators, but we do not encourage this behavior.

2.5.3 Implementation-related Problems

Exception handlers in the four libraries are different, since PPL uses $C++$, and the others use C or *Ocaml*. Even with the C language, we have totally different ways of handling exceptions. Sub-libraries used in the four libraries are also different: we have for example the GNU multi-precision with floating point and integer versions. The control of memory space used by a *Max Object Size* is different.

For debugging purposes, variable names (strings) are preferable to numerical dimension, i.e. $0 \to (n-1)$ or $1 \to n$, but their use in some libraries is not supported. Furthermore, when the objects, which are passed to the abstract domain engine by the analyzer, are lost by side effects, we cannot debug them inside the engine, i.e. the abstract domain library, itself [32]. If every operator made backups of those objects, it would reduce the overall performance. If not, since exceptions are unpredictably raised, we might not be able to restore the objects for debugging. We then suggest the use of two versions for every operator that modifies its inputs. The debugging version should make copies of the inputs, so that in case of exceptions, it can restore the original ones.

As we have seen in Section 1.2, POLYLIB's implementation for timeout management was refused. However, it is important since our examples in Chapter 5 demonstrate that most of the polyhedral operators can be blocking. Each operator has its own complexity, hence a different value of timeout. It is then required to use an additional argument for every operator or a timeout field of values. We propose to use only one parameter p to define the timeout value, under user control, and a field of heuristic coefficients, since it is simple and flexible enough. For example, the emptiness test could require a timeout value of $2p$, the union operator $5p$, the projection $3p$, and so on. Notice that the values of heuristic parameters need to be defined throughout experiments.

In HQ, we decided to support the automatic selection by default and overridden selection by user, for algorithm selection, i.e. which algorithm to use among several algorithms. We also decide to interface explicitly the integer and rational algorithms. The minimized/lazy versions are made automatically without any additional user input. Memory management issues such as version recycle, reference counter and control of used memory space are explicit.

The problems raised by differences of 32-bit, 64-bit, 128-bit and gmp (GNU multi-

[32] This occasionally happens in C^3 when out-of-memory space or overflow exceptions are raised.

precision) computation are not dealt with in our interface, but at compilation time [33]. In HQ, we use an abstract type HQValue for numerical values, which hides these problems. Thus, this is an open problem. The approach that uses two arithmetics at the same time, implemented in LRS library Section 3, page 53, is not taken into account. We do not discuss the approach using product of abstract domains in this work, as well as recent approaches to speed up polyhedral operators such as Cartesian factorization, or dedicated servers for expensive operations that permit execution of several algorithms in parallel.

2.6 Conclusion

In this section, we have analyzed in details some important abstract operators, and proposed an interface called HQ. It reveals lots of compatibility problems among the four existing interfaces used in static analyzers. Compatibility problems have several origins:

- Signature related problems between implementations of each abstract operator;
- Availability of some implementations;
- Different abstract domains, different semantics;
- Implementation related problems.

Our work aims at unifying those interfaces, which requires some adaptations since simple wrapping is not enough. Then wrapping layers for the common interface, through which existing libraries can be reused efficiently, will be implemented.

The most visible application of our work is that an analyzer can use, through a wrapping API, the polyhedral domain, which is more precise, and the octagonal domain, which is less expensive. Thus we can adapt the behavior of the analyzer in static analyses and transformation of programs. Then, the impacts of using different abstract domains can be observed. One of the perspectives of this common interface is to be able to automatically change from one domain to another in an intelligent way, in order to obtain the best compromise between precision and speed.

The beginning of the APRON project, described in the next section, as well as the workshops VMCAI 2005 and NSAD 2005, do not have great impact on our HQ interface, since the workshops did not focus on the problem, and the APRON project has a different point of view to ours.

3 Related Work

In this section, we briefly describe two related projects, the APRON project, which built on my work, and the Parma project, where the Parma Polyhedral Library PPL is implemented with the same objective - a generic and robust polyhedral interface.

[33] In Chapter 5, we study the arithmetic differences in more details with the polyhedral domain.

3.1 APRON project

The introduction of this project is mostly retrieved from the proposal of the project, which can be found at the project website [APRb].

3.1.1 Introduction

Five research teams from Ecole des Mines de Paris, Ecole Normale Supérieure de Paris, Ecole Polytechnique, Verimag, IRISA, all active in abstract interpretation research, are dealing with problems limiting the effectiveness of current static analyses used to statically check safety and security properties, and to identify and locate origins of failures [APRb].

Many abstract domains for the analyses of numerical variables of programs, which offer common as well as specific functionalities, have been designed and used (see Chapter 3). However, approximate analyses must be used to overcome calculability limits, and using these abstract domains' engines is not problem-free. The current resulting accuracy must be improved to reduce areas of uncertainty, which may reach more than 98 percent of an application analyzed by commercially available automatic tools [APRb]. Execution time and memory space, as well as accuracy of analyses, must be controlled together: an analysis lasting a whole night and requiring more than 4GB of main memory does not have the same industrial impact as a five minute long analysis running on a laptop.

Existing APIs of those engines are not compatible. They are not context-of-use independent and some fundamental functionalities are not available in some implementations. It is difficult to compare those interfaces since a generic interface that would define the exact semantics of its signature is not available. For example, we can distinguish between the projection operator and the *projection_and_minimize* operator, or not, in the polyhedral abstract domain. Moreover, since the precision of different abstract domains and the efficiency of different implementations of the same abstract domain are not easy to compare, we need to study these questions.

Problems identified by the teams and addressed by the APRON project are at the conceptual, technological and experimental levels. How can floating point operations with rounding or non-linear expressions be mathematically modelized? How can we automatically control the trade-off between accuracy and speed by switching abstract domains by tuning the number of control points, by adding auxiliary variables to memorize part of the execution trace, by increasing the number of contexts for procedure analysis?

Breakthroughs at the conceptual level will require that technological barriers be pushed too: how to design a generic analyzer performing interdependent analyses? How can new abstract domains defined and implemented by one team be readily used by another one? The validation of the conceptual breakthrough will also require experimental breakthroughs: can we define abstract interpretation case benchmarks and abstract domain benchmarks?

The main targets for APRON analyzers are control applications with large numerical components, using floating point numbers, counters and arrays, that are intractable by finite state-based methods. These applications will be real-life applications, in the 100 to 500 KLOC range, with thousands of modules. Adaptative abstract interpreters are required to deliver the accuracy and the effectiveness these applications demand.

A generic analyzer performing interdependent analyses, using any available abstract domain, with experimental results is the final target. However, the APRON project is in its very first phase. The generic analyzer is divided into several layers, which are only defined at the lowest level, 0.

3.1.2 Our Contribution

Since this project's objectives overlap with our work, we have contributed to its starting up with our participation, namely in detecting incompatibilities of available libraries, in analyzing the existing problems, in proposing a functional API, in this chapter, and in providing experimental results for the polyhedral domain, in Chapter 5. Our HQ interface is then discussed at APRON meetings where propositions, modifications are presented to obtain common decisions. From these decisions, a common interface is then proposed. The next section summarizes the decisions [APRa].

3.1.3 Main Decisions

The objectives of a common interface is to identify the fundamental functionalities that an abstract domain used in static analyzers must supply, then to design a concrete API with data types, functional signatures and their semantic definitions, and finally its implementation. This interface should be generic and must satisfy the need of APRON's members.

In the context of APRON, since performance is important, simplicity and minimality can be sacrificed. Furthermore, the needed modifications to the existing implementations should be minimal, thus the common interface might not be optimized. Bertrand JEANNET has proposed a common interface and implemented a prototype which can be found at APRON's site [APRb].

Levels and Problems Two levels have been identified. The lowest layer, level 0, consists of implementations of different engines such as C^3, New POLKA, PPL, Octagon, OMEGA, CUDD, etc (see Chapter 3 for these libraries). At this level, overflow and timeout exception are dealt with. Memory management, thread-safety and performance are also considered at this level, as well as debugging functions such as printing functions which depend on the implementations.

Only at this level can we have abstract domain specific operators which are not shared with other domains. The interface for this level is minimal, except the case where algorithmic advantages can be achieved. Different implementations can be combined to produce a new library. Thus the Cartesian product implementation and dimension change approach [Mer05] can be integrated here.

The genericity versus comfortable use question has been raised, i.e., whether the interface should support or not functions with multiple arguments, e.g. convex hull of a list of constraint systems. It was decided that performance-related issues, which depend on the used domain, are treated at level 0, while comfortability-related issues are treated at level 1. However, it is difficult since some algorithms such as minimization, projection or convex hull require optimizations at level 1. It was decided that dimension management, implemented in C^3, is to be dealt with at level 1.

Problems at higher levels are dealt with in the static analyzers. For example, the question of precision and arithmetics for 8, 16, 32, 64, 128 bits and gmp (GNU Multi Precision) computing is dealt with at some level higher than 1, because it depends much on the techniques used by analyzers. Numeric types specified by *Value* in C^3 or *pkint* in New POLKA are defined at compile time.

Structure Manager It was decided that the interface does not permit to choose an implementation among several implementations or algorithms. It was also decided that exceptions are handled at level 0 but not by the interface. Instead, a special structure called *manager* is designed.

This has an impact on the interface: exception handlers and several versions of an operator, including approximations, are hidden from the interface.

Since multi-algorithms are supported, we need a mechanism to choose different algorithms for an operator. We can specify a default algorithm, and then the fastest to the most precise algorithm are enumerated. As the number of available algorithms is different between domains, the enumeration can be surjective [34]. The selection of an algorithm is executed by means of the manager. The default one can depend on the domain or the operator in question, which means that the initialization is not generic. The timeout value has to be differently defined by the manager for each operator.

The manager is used for many purposes and it contains several flags. For example, when an operator returns a polyhedron, in order to know if the result is exact or not, we can test the flag that is designed for this purpose.

Other Decisions The language of reference is the C language. Dimensions are typed and numerated from 0 to $n-1$. Any dimension should be accessible by level 1 operators. Note that only C^3 and PPL libraries have the dimension permutation functions. Thread-safety and timeout exception are supported. The exception handlers consist of *not_implemented*, *invalid_argument*, overflow, timeout and *out_of_space*, with interruptions by a catch and throw mechanism for overflow, timeout, *out_of_space* exceptions.

[34] For example, if we have five polyhedral algorithms and only two octagonal algorithms for the projection operator, then the enumeration is surjective (five to two) in the octagonal case.

Other decisions are: prefix for each implementation; functional or/and imperative signatures at level 1, with recycle/destructive version; no reference counter which is hidden if implemented; all implementations must cover the interface but may have their own extensions; improvements such as factorization are hidden from the common interface; constructor, destructor, debugging functions are required; using a system of *varargs* instead of a list or a table of arguments.

3.1.4 Open Problems

As we have seen, this work is in progress. The functions for conversion among domains between implementations need to be defined [35]. Since there are more than two formats, an universal format is suggested so that it can be converted to other format and vice versa, to reduce the number of needed conversions. Nonetheless, this approach influences the performance in some cases.

Concerning the signatures, there are decisions to make about function names, return codes, exception handlers or argument types. These decisions are dealt with in the prototype developed by Bertrand JEANNET. This prototype will be used to adapt New POLKA's interface, and then experimentations with the new version of New POLKA are planned.

A lot of work remains to be done to construct a common interface. Not treated in APRON are products of domains, dynamic adaptation for higher precision, or switching domain. In fact, a great deal of problems mentioned in our HQ interface also are problems for APRON. Since this is a collective work, the main question will be how to persuade the others, not only APRON members but also the static analyses community, to adopt our technology?

3.1.5 Conclusion about APRON

While APRON's members prefer to minimize the number of changes required to adopt the new interface, we are free of that obligation. This leads to several differences, among which the most observable is the way that we design the new interface and present the problems.

In HQ, we divide our approach into two different parts for clarity, and to simplify the problems. The first part focuses only on the interface, using a high level language like Java, with an object-oriented approach, thus hiding as many as possible implementation issues. The second part, alongside this interface, presents the problems not directly related to the interface, such as how we handle exceptions, how we convert from the javadoc-generated interface to other languages such as C, Ocaml, etc.

[35] In fact, these functions can be considered as some generic operators like the minimization, normalization or canonization operators.

In APRON however, one immediately goes into details, by attacking the *is_bottom* operator! Then a black box called *manager* is designed in order to keep information of algorithm selection, exception flags, etc. An advantage of this approach is that we can analyze the problems in more details.

More important, HQ appoach is in fact PIPS-oriented, since the author works in PIPS development group. The APRON approach is in fact a layered aproach that needs to take into account other analyzers, since its members are ASTREE and NBAC developers. In Figure 4.3, on the left hand side, we see APRON approach with several levels among which, the APRON project works on level 1. Level 0 concerns the abstract domains' implementations, which are in fact the existing libraries. On the right hand side, we have HQ approach and the objective to be able to use these available libraries.

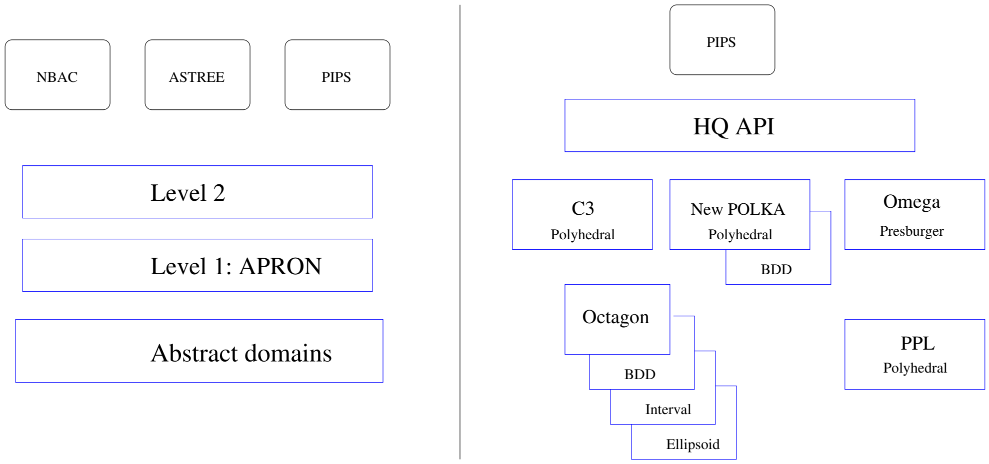


Figure 4.3: HQ and APRON approaches

Furthermore, a plus for the HQ approach is that it deals with the robustness issue, i.e. exception handling. In our opinion, these two approaches complement each other, at some levels; however, APRON's prototyped solution seems to be too specific, since many of its solutions are based on previous decisions that were made by APRON's members. For instance, the structure manager deals with too much information including exception handlers, while we would prefer a simpler, yet standard way that uses returned codes to indicate exceptions.

In conclusion, our HQ interface is built in order to serve the APRON's project to build the common interface, which itself will be accepted and implemented by APRON's members.

3.2 Parma Project

As we can state from our HQ's documentation [QUEa], the Parma library's interface is the richest one, compared to the other three libraries C^3, New POLKA and Octagons

[36]. Implemented in $C++$, it is designed to be a general-purpose library, with a strong support for any static analyzer. Therefore, its objectives overlap with our work. However, we consider that its current interface is not totally satisfying.

Firstly, since the interface of the Parma library PPL is not frozen until its official 1.0 release [37], it can still be changed.

Secondly, PPL is different from our approach since multi-algorithm signatures are not supported. PPL's important algorithms such as projection, minimization and convex hull are based on the Chernikova algorithm, i.e. dual conversion-based like in New POLKA. Therefore interesting implementations using constraint system manipulation algorithms in C^3 are not available. However, these algorithms are interesting due to the fact that some algorithms have very low complexity and can be used as approximations. As presented in Chapter 5, static analyses using polyhedral abstract domain are expensive, thus approximations are fundamental.

Thirdly, since PPL is developed in C++, issues like exception handlers for C language are not considered. And timeout exceptions are not yet supported.

Finally, since PPL library is designed as a polyhedral library, newly introduced libraries such as the octagon library with its interface, are not yet taken into account.

4 Conclusion

In this chapter, we have presented our approach to deal with compatibility problems in the context of static analyzers. The incompatibility of existing proving engines, i.e. existing abstract domain libraries, stop us from effectively using them (see Chapter 3 for these libraries).

Our proposition for a common interface for abstract set manipulation engines, described in Section 2, helps to identify the problems, and presents our very first solutions.

It consists of a prototype for a common interface, and practical issue-related documents that reveal different approaches in existing implementations, from which concrete decisions can be made. We have also introduced the state of the art of the APRON project, which is strongly related to this dissertation, in Section 3.

We compare our HQ approach and the APRON approach at the end of Section 3. In fact, the HQ interface was presented in APRON meetings, and helped starting deeper discussions on the subject. Our proposition is divided into two parts.

The first part directly addresses the interface with imperative signatures, which should describe what we need, how to present, introduce and expose it as clearly as possible. We discuss the name of operators, what this function does, why we need other versions of it, when we apply approximation, whether we should have a list of arguments instead of only one argument, etc. the level of this function, etc. These problems are already complex,

[36] See Section 3, page 55 for the PPL library introduction.

[37] At the time of this writing, we have the 0.7 release.

thus they should be separated from the second part.

The second part, which is as important as the first part, is where we discuss implementation issues such as how we handle exceptions, how we manage the memory, etc. In fact, there are many ways to deal with this kind of problems, so finally we just have to pick one that is the most appropriate. For example, from the initial HQ's signatures, we can use the JNI (Java Native Interface) tool to generates its C signatures. However, this approach is not satisfying since the generated code is not easy to understand, so we can consider building a set of rules for this conversion. Other problems such as memory management, destructive functions, destructive arguments are described in order to be decided later.

Recent developments show [38] that current polyhedral libraries such as POLYLIB, PPL and APRON, as well as polyhedra-related libraries such as Octagon library, are being worked on their interfaces. Likewise, new abstract domains are introduced, e.g. in [FER05b], which may have an important impact on our common interface. For the time being, the very first common interface is defined by the APRON project.

Some complex problems such as product of domains or the Presburger domain are not dealt with. At first, the implementation of the common interface will help the three static analyzers described in Chapter 3 to profit from those abstract domains. Then, given the compatibility among the libraries, other analyzers using abstract interpretation can use them, too.

We intend to continue working on the HQ interface, where the compatibility problems are documented [39], and some adaptations from the APRON's prototype are also encouraged. In the next chapter, we study the interface of the polyhedral domain, the underlying algorithms and related problems.

[38] At the VMCAI 2005 and NSAD 2005 workshops.
[39] APRON's prototype does not supply this kind of documentation.

chapterComparative section for polyhedral operators

Chapter 2 introduces the basic concepts of static program analysis, whereas Chapter 3 presents available abstract domains used in static program analysis, including the polyhedral domain. There exist several libraries to manipulate objects representing polyhedra. A class of either general-purpose or domain-specific libraries are used in static program analysis as abstract domain, implementing polyhedral operators.

In this chapter, we discuss which implementations were used for each operator, which algorithms have been chosen and why, what are the principal differences among them, in order to suggest a better use of those works in the context of program analysis and transformation. Our contribution to improve some of these operators is also presented in this chapter (Section 7.2.3, Page 98, Section 10.2.3, Page 113). For the list of these libraries, as well as convex polyhedra related definitions, readers are referred to Chapter 3.

5 Polyhedral Operators and Open Issues

5.1 Polyhedral Operators

Using convex polyhedra for the manipulation of numerical information is a key idea which is employed in several systems for the analysis and verification of hardware and software components. Unfortunately, polyhedral interface being composed of many operators, its design is not universal and strongly depends on the need of developers. However, in existing polyhedral libraries, we can find a common part for these operators, even though they may have slightly different names. We present in this chapter only the most important operators.

Note that a convex polyhedron can be over R^n, Q^n or Z^n, depending on the implementation. In this chapter, we use D^n if we do not want to specify the domain. General definitions related to convex polyhedra can be found in Chapter 3, Section 1.2, page 29. Given the incompatibilities among several polyhedral interfaces, our polyhedral operator naming is based on the work of the APRON project [APRb].

5.2 Open Issues

As mentioned in Chapter 2, Section 3.2, Page 10, *abstract interpretation* is a method of formalizing the approximation relation between the *concrete semantics* and the *abstract semantics*. Convex polyhedra are used to represent the semantics of a program as an abstract semantics. Constraint systems and generating systems representing polyhedra can be implemented with several different physical representations.

Some implementation integrates some dimension naming scheme (i.e. C^3 [PIPa]); others use a fixed enumerated space dimension, from 0 to n (e.g. New POLKA [JEAb], Octagon [MIN] and PPL [PAR]). The first one, with dimension names is called *sparse* representation, and the second one, with fixed enumerated space dimension, is called *dense* representation. Handling of constant terms is also implemented differently.

When using polyhedra, redundancy in its representations is a problem. For one polyhedron, we can have redundant constraints, rays, lines or points. The *minimization* operator that deals with redundancy is an expensive operation, thus we cannot apply it everywhere. In practice, the physical memory space requirement can be huge.

Since the representations of polyhedra are practically not *normalizable*, different implementations are not interchangeable. Problems of incompatibilities like format conversion make reuse of code and sharing experiences not obvious.

As the complexity of polyhedral operations is high in terms of execution time and memory space, exceptions are to be dealt with. The overflow exception, when the computation raises too large a number, is handled by most existing implementations, though not throughly. Meanwhile, the timeout exception is not systematically considered.

PPL [PAR], written in C++, deals with overflow exceptions using C++ exception handlers. C^3 library [PIPa], written in C, uses a set of macros and exception handlers using long-jumps, previously implemented in POLYLIB [LOE], and provides a good way to detect and handle this kind of exception. The LRS library [Avi] also implements its own mechanism that detects overflow exceptions, but like other libraries such as New POLKA [JEAb], CDD [FUK], it suggests using GNU Multi precision algorithms, or built-in C calculation for fast execution. This way, no overflow detection is needed, although strange behaviors can be observed when using this version. No overflow detection is implemented in New POLKA and CDD.

Timeout exceptions become vital when the running time of large scale applications is important. However, from the well-known POLYLIB to the recent development of PPL, there are no timeout handlers. In general, overflow exceptions come before timeout exceptions, when the problem size is not very large. Otherwise, when an elementary operation takes one or two minutes without overflow, the total performance can be disastrous, knowing that hundred thousands calls of the same operator can be expected on a typical session of program analysis and transformation.

In C^3 library, timeout exceptions are integrated beside overflow exceptions, but as it uses the POLYLIB library for some of its operators, the implementation is not straightforward.

Regarding C implementations, the mechanism for exception is using long-jumps. So thread-safety becomes a problem, which although not being yet a requirement for existing implementations, can prohibit the integration of these tools.

For different contexts of use, the computational engine of (polyhedral) libraries are sub-libraries with different precision: libraries dealing with integer, rational or floating point algorithms; computation for 32-bit, 64-bit, 128-bit or GNU multi-precision numbers, not to mention the different versions used in some libraries, or their variants. Fortunately, most polyhedral libraries support the C language, or are written in C++.

The impact of those differences on program analysis is not yet systematically analyzed. Hence only by experience developers remark the changes using different precisions of computation. For example, PIPS developers prefer not using the GNU multi-precision before precision becomes an issue, because the complexity for 64-bit computation already costs much time. Thus, decisions made by different projects can also limit integration of existing tools.

Many impact related questions are interesting. For example, which *precision* shall we obtain if we use 64-bit over 32-bit computations, or what do we lose if we are 64-bit, instead of multi-precision. Consider that in term of program analysis and transformation, if we lose almost nothing from moving more expensive computation like GNU multi-precision to 64-bit arithmetic, we should evidently be 64-bit. If it is a great lost, e.g. we actually cannot eliminate a dead code because 64-bit is not precise enough, then we certainly should try the GNU multi-precision. While the difference in results of program analysis is somehow not obvious to describe, comparison of the numbers of exceptions for example can be a simple and good approach.

The main question is then the issue of precision versus the approximation. The actual answer is divided into two categories: tuning among the abstract domains, where more precise domains go along with exceptions, or using the same domain while applying approximation techniques like the widening operator, or based on Galois connection (see [CC91] or [CC92] for a comparison of these two approaches). The technique developed in PIPS, described in Section 13 Page 120, is based on Galois connections.

Beside the availability of implementations, we also need to mention the absence of important algorithms, e.g. integer arithmetic algorithms. Depending on the need or sometimes because of algorithmic difficulties, developers do not implement such algorithms. However, whenever an important improvement is found, the others should be able to profit. As many libraries are built from analyzing pre-existing libraries (e.g. POLYLIB [LOE], New POLKA [JEAb] and PPL [PAR]), sometimes it is hard to know which one is more efficient. Any claim of improvement then needs to be evaluated. To our knowledge, it does not exist yet a mechanism for this kind of evaluation, so our contribution is to introduce a polyhedral benchmark in Chapter 5.

Given this lack of important algorithms, approximate analyses are used along with approximate algorithms. Utility of exactness is not fully studied. In some cases, it is shown that exact algorithms are not necessary. For example, an empirical study on dependence analysis and array reference in [SLY90] shows that the coefficients of array reference pairs usually are 1 or -1, which means a data dependence exists if and only if there are real solutions to the systems derived from their subscript expressions. Among 4105 pairs of two-dimensional array references, 97% (3997 pairs) verify this condition. Thus, the faster real version of the satisfiability test is enough, instead of the integer one. On the contrary, the exactitude of algorithms can help avoiding computations of empty polyhedra that are not detected by real algorithms.

Recent developments, such as factorization applied to large scale analyses in order to improve actual algorithms (e.g. [HMPV03]), are interesting but difficult to exploit. For example, continuous developments and changes in PPL library (since version 0.3) make it harder to implement the Cartesian factorization [HMPV03].

Backup algorithms applying approximations are important for each operator but they are not throughly studied. In case of magnitude overflow or insufficient memory space, correct approximate operators should be available to continue the analysis.

In the same way, each project has its own debugging techniques, and supports them at different levels. Thus, to help the re-utilization of equivalent works, co-operation among groups of researchers should take place.

In order to deal with the above problems, by analyzing existing tools, problems and the need, before going to the section dedicated to our discussions and suggestions on algorithms, heuristics, debugging techniques, exception handlers, etc., we limit ourself to a list of important operators.

We organize this chapter in an operator by operator way in order to fully describe operator-related problems, thus it is maybe sometimes redundant. Each operator has its particular issues, but we have tried to keep sections as balanced as possible. We have tried to describe our points of view on difficulties towards an integration of related work throughout the sections.

Based on this chapter, on our proposition of a common interface for polyhedral operators (Chapter 4) and on our system of benchmark with experimental results (Chapter 5), we have established a framework for analyzing polyhedral implementations.

6 Dual Conversion

6.1 Introduction

The decomposition of polyhedra, described in Chapter 3, Section 1.2, page 29, says that a polyhedron can be represented in such two manners [40], called *H-representation* (i.e. constraint system) and *V-representation* (i.e. generating system).

The *Dual Conversion* or *Double Description Conversion* operation consists of converting a polyhedral representation or constraint system to its dual form and vice versa. It is important because, for some operators, it is much more efficient applying algorithms with one representation than the other. For example, the convex hull of two polyhedra using the generating system representation, or the intersection using constraints.

The *vertex enumeration problem* is to compute V from H, and the *facet enumeration problem* is to compute H from V. These two problems are essentially equivalent under point/hyper-plane duality, thus any algorithm for the vertex enumeration problem can be used for the facet enumeration problem (see [Sch86] page 120).

This operator, though not being an interesting operator itself for use in program analysis, is key for a class of existing polyhedral libraries like C^3 ([PIPa]), New POLKA

[40] Also known as *double description*.

([JEAb]), POLYLIB ([LOE]) and PPL ([PAR]). It can be applied directly in useful operators such as the feasibility test, the projection along one dimension, the minimization and the convex hull of polyhedra.

6.2 Available Algorithms

The most used algorithm for the dual conversion is the Chernikova algorithm that finds an irredundant set of vertices and rays of a polyhedron, defined by a mixed system of linear equations and inequalities ([Che64, Che65, Che68]). In fact, this algorithm is a variant of Fourier-Motzkin algorithm ([TMT53]). It is also mentioned in [HAL79], and optimized by Le Verge, in the detection of redundant elements [HLV94].

In fact, these algorithms add a new dimension to the polyhedron to be converted to obtain a cone with the vertex at the origin. A cone is represented by a vertex (at the origin) and a set of rays and lines. The polyhedron is then viewed as the intersection of the cone and a hyper-plane. This technique simplifies the implementation of the algorithm, while it does not change the generality of the method.

There exist some other algorithms which can be used for solving the same problem ([MR80, Chv83]). Some algorithms (e.g. Chernikova's) fortunately return both representations in minimized form (i.e. redundant removal included); some others do not. However, being an operation of exponential complexity, execution of these algorithms can result in overflow exceptions, or unacceptable running time. To understand the differences among these algorithms, it is interesting to have a look at all available algorithms and implementations.

Given the duality of the dual conversion, which means one conversion is linear time reducible to the other, we restrict our discussion to finding the generating system, also known as the vertex enumeration problem. Yet a more appropriate definition of the problem is to require the *minimality* of the generating system, which is unique up to positive scaling when we assume the regularity condition that the cone is pointed, i.e. the origin 0 is an extremal point of the polyhedron. Geometrically, the columns of a minimal generating system are in a one to one correspondence with the extremal rays of the polyhedron. Thus the problem is also known as the extremal ray enumeration problem.

The extremal ray enumeration problem has been studied, directly or indirectly, by many researchers in mathematics, operations research, and computational geometry etc. Despite that, no efficient algorithm for the general problem is known.

In essence, there are only two main classes of algorithms for the problem: *pivoting* and *incremental* algorithms. The pivoting algorithms implement the idea of the simplex like in linear programming, where we start from a vertex, go to another adjacent vertex, then by some way travel all the set of vertices. The *gift-wrapping* in [CK70], *Lexicographical Reverse Search* in [AF92] belong to this class.

The incremental algorithms compute the vertex description by intersecting the defining half-spaces sequentially. An initial simplex is constructed from a subset of $n + 1$ half-spaces and its vertices. Additional half-spaces are introduced sequentially and the vertex description are updated at each stage. Essentially such an update amounts to identifying and removing all vertices that are not contained in the new half-space, introducing new vertices for all intersections between edges and the bounding hyper-plane of the new half-space, and generating the new edges between these new vertices.

The first incremental algorithm, known as *double description* method, is presented by Motzkin and al ([TMT53]), then it is rediscovered and refined by Seidel's *beneath and beyond* method ([Sei81, Sei87, Ede87]), Clarkson and Shor's *randomized* algorithm and Chazelle's *derandomized* algorithm. The method known as Chernikova's algorithm ([Che64, Che65, Che68]) is basically the same method.

Optimizations in Particular Cases - Better algorithms can be found for the case of two and three dimensions, where $O(n\ log\ n)$ time in fact suffices (instead of $O(n\ log\ n + n^{d/2})$ for example). But as the dimension of the space increases (beyond three), certain assumptions that were valid in lower dimensions break down. For example, any $n-vertex$ polygon in two dimensions has exactly n edges. Nonetheless, the relationship between the numbers of faces and vertices is more complicated even in three dimensions. A cube has 8 vertices and 6 faces, while an octahedron has 8 faces and 6 vertices.

In planar space R^2, the lower bound is $O(n\ log\ n)$ just like the sorting problem (see [vLa90]). If we know that the number of edges of the polyhedron is small, then Jarvis' March [Jar73] is appropriate. If the points of the polyhedron are already sorted, Graham's Scan, [Gra72], can be applied directly. For three dimensions, either the generalized Divide and Conquer of Preparata and Hong ([PS85]) or Chazelle's algorithm [Cha93] can be chosen, or randomized algorithms [41], all with expected running time $O(n\ log\ n)$.

Nonetheless, some of the above algorithms are not suitable for the implementation of general dual conversion problems. Moreover, algorithms in higher dimensions are also efficient for R^2 and R^3, therefore above algorithms are not implemented in existing polyhedral libraries.

For higher dimensions, the worst case lower bound is $O(n^{\lfloor d/2 \rfloor})$. In practice, the Chernikova's algorithm (or Fourier-Motzkin's Double Description method) are implemented with different techniques like Le Verge's optimization [Ver92] or CDD's [FP95]. Lexicographical Reverse Search [AF92, AF96] and the Quick Hull [BDH96] are also important implementations [42].

Approximation Algorithms are useful for applications that require rapid solutions even at the expense of accuracy. There are two way to approximate the result: conservative and liberal. The former computes the set that is included in the true result. The latter outputs an approximate set that is a superset of the true result.

[41] See Appendix page 189.

[42] The list of available algorithms can be found at Appendix page 189.

Parallel algorithms are introduced in order to improve the performance of double description computation. For example, it is known that the convex hull of n points in R^d can be constructed in $O(log\ n)$ time using existing $O(n\ log\ n + n\lfloor d/2 \rfloor)$ algorithms ([AGR94]). But it is surprisingly not yet employed in existing implementations.

Decomposition of polyhedra in high dimensions permits computation of polyhedra in lower dimension, which is interesting since experiments show that the running time is proportional to the dimension [Mer05].

6.3 Practical Problems

In practice, differences in implementation lead to incompatibilities in internal representations of database structures, which raises a problem of format conversion, computational capability, where libraries of different precision are used (i.e. rational, floating point and integer; 32-bit, 64-bit and gmp libraries), and exception management, where different programming languages are used, etc. In this case, dual conversion is implemented in CDD (Section 3, page 52), in LRS (Section 3, page 53), in New POLKA (Section 3, page 54), in POLYLIB (Section 3, page 54) and PPL (Section 3, page 55): no library is fully compatible with one another.

As we have seen in Chapter 3 Page 51, the CDD is rational, while the others are integer. It makes the reuse and performance comparison of these algorithms much more difficult. Moreover, some libraries support thread-safety, while some others do not. Indeed, in the context of program analysis, the result can be affected by computational precision, such as 32-bit, 64-bit and gmp computation. In Chapter 5, Section 2.8 our analyses on some $64-bit$ polyhedral databases reveal the percentages of the coefficients that are greater than $32-bit$ capacity. However, these numbers do not tell exactly the difference between the above computational precisions.

We do not have yet any backup algorithm for this operator in case of exceptions, and it is sometimes difficult to detect an implementation error for the above algorithms.

7 Satisfiability

7.1 Introduction

The *Satisfiability* test, also known as *feasibility*, or *emptiness* test, verifies if a polyhedron is empty or not. A polyhedron over D^n is said to be *not empty* when it contains at least one element in D^n, or *empty* if it contains no element.

The emptiness test of a polyhedron by means of its generating system is equal to the test of satisfiability of its constraint system since we have the duality of the two representations. The satisfiability test: a constraint system is said to be *satisfiable* (or *feasible*) if all its constraints are simultaneously satisfied (or feasible). On the contrary, if we detect a contradiction among its constraints, it is said to be not satisfiable (or *infeasible*).

In exceptional cases, when we cannot prove that the system is feasible or infeasible, we say that the system is *non practically calculable*. There exists a number of methods to perform this feasibility test, for example the Simplex method described in [Sch86] (page 129), as well as using the dual form (conversion from the constraint system to the generating system reveals the satisfiability).

Sometime this operator can be referred as, although it is not much in use yet, *is bottom* test of an element. This is due to the fact that when we consider the lattice of convex polyhedra, we call the empty polyhedron the bottom element of the lattice, and the universe polyhedron that spans all the space, the top element (see Chapter 4 for naming explanation).

7.2 Available Algorithms

We are interested in available libraries implementing this operator, that is to say C^3 ([PIPa]), CONVEX ([Fra]), JANUS ([SOG]), OMEGA ([Ome]), PIP ([FEA]), New POLKA ([JEAb]), POLYLIB ([LOE]), PPL ([PAR]) and PORTA ([TC]). Even though OMEGA library is not really polyhedral, its implementation of this operator might be interesting, because it can be used to solve our problem. Available algorithms can be divided into three categories: the Fourier Motzkin, the Simplex method, and the double description method. In general, these algorithms are all rational or real.

However, as an integer answer means more precision in program analysis and transformation, integer treatments in some stages of existing methods are implemented, including two approaches. The first one scans the solution domain to check if it is integer, whereas the second approach tries integer checks at every step of the algorithm. Following the second approach, in C^3 [PIPa], the Fourier Motzkin algorithm is modified so that it searches for rational answers, and for integer answers if only possible (i.e. the algorithm is integer/rational). JANUS, an implementation that combines both approaches, using the cutting plane method described in [Sch86] (page 129), proposes an integer algorithm that is relatively efficient [SOG96].

The double description method, while being rational, is implemented in most of the important polyhedral libraries: New POLKA, POLYLIB and PPL. For POLYLIB, since the generating system is always computed and kept in memory, we should take into account the dual conversion, i.e. the double description method, in the test of emptiness.

The question is then, which one among these algorithms is the most relevant to use in our context of static program analysis and transformation? In order to anticipate the performance of these algorithms, in this section, we discuss some important points by analyzing the algorithms. Finally, these are justified by our experiments, in Chapter 5.

7.2.1 Fourier - Motzkin

The well-known Gaussian method for solving a system of linear equations by successively eliminating variables has its variant for linear inequalities. It was described by Fourier[1827], Dines [1918] and Motzkin[1936], so it is called the Fourier - Motzkin elimination method. The idea of the method can be well illustrated by application to the following problem: *Given a matrix A and a vector b, test if $Ax \leq b$ has a solution, and if so find one.*

Let A be a matrix of dimension $m \times n$, b a vector of m components $b = (b_1, \ldots, b_n)^T$, then $x = (\xi_1, \ldots, \xi_n)^T$. As we may multiply each inequality by a positive scalar number, we can assume that all the entries in the first column of matrix A is 0 or ± 1. So the problem is to solve:

$$\begin{cases} \xi_1 + a_i x' \leq b_i, (i = 1, \ldots, m') \\ -\xi_1 + a_i x' \leq b_i, (i = m' + 1, \ldots, m'') \\ a_i x' \leq b_i, (i = m'' + 1, \ldots, m) \end{cases}$$

where $x' = (\xi_2, \ldots, \xi_n)^T$ and $(a_1, \ldots, a_m)$ are the rows of A with the first column entry deleted.

We have:

$$\max_{m'+1 \leq j \leq m''} (a_j x' - b_j) \leq \xi_1 \leq \min_{1 \leq i \leq m'} (b_i - a_i x') \tag{4.1}$$

The unknown variable ξ_1 can be eliminated by combining all the left side of 4.1 with those of the right side. By repeating this procedure, we can successively eliminate the first $n - 1$ components of x, and end up with an equivalent problem in one unknown, which is trivial. The question of satisfiability is then solved.

Since this method is not polynomial but exponential in general case, we need to implement a timeout mechanism inside this method to control the running time. In practice, it seems to work well with systems of less than 10 inequalities (with this size, the *sparsity* is not high, see [YAN93], page 93). We note that there is an important connection between the projection operator and this method: each step of elimination is a projection along one variable.

7.2.2 Simplex

DEFINITION 7.1 If A is a matrix of $m \times n$ dimension, b is a vector m components and c is a vector of n components, then the problem of finding a vector $x = (x_1, \ldots, x_n)$ that satisfies $Ax \leq b$ and maximize $f = cx$ is a *problem of linear programming*. The problem is then written:

$$\{\max cx | Ax \leq b\} \tag{4.2}$$

The possible *solutions* are all those that satisfy all the constraints in the problem. The *optimal solutions* are those possible solutions that have the maximum value of f. The linear function $f = cx$ is called *objective function* or *economical function*.

A problem is *linear* if these two conditions are satisfied:

1. The objective function f is linear.
2. The constraints are linear: $\sum a_{ij}x_j \leq b_i$

As we have mentioned, a generating system is a representation (though not unique) of a convex polyhedron, with extremal points, extremal rays and lines. The original idea of *Simplex* method is, when an initial solution is known, to visit all extremal points along extremal rays or lines, searching for the optimal solution. At each step, if the optimal solution is not found yet, the algorithm pivots to another extremal point.

The satisfiability test applies the Simplex method in a modified way: consider an *intermediate* problem that has a trivial initial solution; the solution of the satisfiability test can be derived from the optimal solution found for the intermediate problem.

Since Function $f = cx$ and the constraints are linear, we can suppose that $x \geq 0$ without lost of generality [43]. Thus the general problem of 4.2 can be written:

$$\{\max cx | x \geq 0,\ Ax \leq b\} \tag{4.3}$$

We can write $Ax \leq b$ as $A_1 x \leq b_1$, $A_2 x \geq b_2$, $b_1 \geq 0$ and $b_2 > 0$.

Consider the intermediate problem constructed as follows:

$$\{\max \mathbf{1}(A_2 x - \tilde{x}) \mid x, \tilde{x} \geq 0,\ A_1 x \leq b_1,\ A_2 x - \tilde{x} \leq b_2\} \tag{4.4}$$

where $\tilde{x}$ is a new variable vector, $\mathbf{1}$ is a vector unit denoting an all-one row vector. Then the ensemble $x = 0$ and $\tilde{x} = 0$ defines a solution of the new problem 4.4. Therefore, we can use Simplex method described above to solve this problem, having this initial solution. If the maximum value of 4.4 is $\mathbf{1}b$, say with a optimal solution $x*, \tilde{x}*$, then $x*$ is a solution of 4.2.

Algebraic variants and the table method. The table method is implemented in library C^3 [PIPa], in Q. From the initial problem, we create a corresponding table. From this table we find an initial base with the initial solution. The economic function is conventionally stocked in the first line of the table. We pivot by modifying the table until some conditions are satisfied.

In practice, when the problem becomes large, we observe that the method is sensitive to the order of the constraints, in term of execution time and of overflow exception. The running time on large examples suggests a timeout mechanism is needed for this method.

[43] Let $y_i = -x_i$ if $x_i < 0$, otherwise let $y_i = x_i$ and rewrite the problem using y_i.

7.2.3 JANUS

JANUS is an algorithm that was designed and developed by Jean-Claude SOGNO at INRIA, which can be used as an *integer* solver for satisfiability problems, in Z^n. The work is well illustrated in the paper [SOG96]. Nonetheless, there are two problems that we would like to address.

First, the original JANUS was implemented with integer coefficients stored in 32 bits. This really is a limit; a wide range of large systems containing large numbers cannot be treated, especially when these numbers accumulate after some computational iterations. Second, there is no head-to-head comparison between equivalent methods to show the effectiveness of the method, except a comparison between JANUS and OMEGA test [Ome] concerning the "nightmare" problem [SOG96].

To overcome the first problem, we have implemented an other version that uses 64-bit computations in JANUS, while keeping the 32 bits enabled. From now on, we call it *JANUS Value*, or just JANUS for short. The second problem can be handled by a polyhedral benchmark.

Experiences show that *Fourier-Motzkin* is not adequate for large constraint systems because of the large number of inequalities combinations, and that Simplex is not appropriate for small systems since it requires building and manipulating large tables. We now have a look at the strategy of JANUS.

The basic algorithm. The problem consists in proving the existence or not of an *integer* solution that satisfies a set of linear constraints with all integer coefficients:

$$\{Ax \leq b : x \in Z^n\} \tag{4.5}$$

The strategy is composed of two phases:

1. Reduce the formulation as follows:
   - eliminate all equalities;
   - eliminate as many variables as possible by selective *Fourier-Motzkin* method;
   - introduce constrained variables.
2. Apply a (dual) cutting-plane algorithm.

These ideas are discussed below:

Equalities elimination. Split the original system 4.5 into two subsystems:

$$\{A_1 x < b_1 : x \in Z^n\} \tag{4.6}$$

$$\{A_2x = b_2 : x \in Z^n\} \tag{4.7}$$

and let solve the subsystem 4.7 containing all the equalities of the original system. Using the Gaussian method ([Sch86], page 31), we can easily solve this system. If the solution does not exist, we can conclude the result of the original system. If it does exist, we need to substitute the variables to obtain a new and equivalent system that has only inequalities:

$$\{A'x < b' : x \in Z^n\} \tag{4.8}$$

Before the next steps, we can divide these inequalities by their greatest common divisor (GCD), to reduce the magnitude. Unitary coefficients (i.e. $[1, -1]$) may appear, which is useful for dummy elimination step (see below). Consider the inequality:

$$\sum_{j=1}^{n} a_j x_j \leq b \tag{4.9}$$

If $d = GCD(a_1, ...a_n) > 1$ then we can substitute 4.9 by:

$$\sum_{j=1}^{n} (a_j/d)x_j \leq \lfloor b/d \rfloor$$

Selective *Fourier-Motzkin* elimination. By projecting one variables by one, the method of *Fourier-Motzkin* ([Sch86], page 155) can explode because of inequality combinations. In this method, when projecting one variable such that its sign is positive in m inequalities and negative in n inequalities, we can end up with $m \times n$ new constraints. The heuristic used in JANUS assures the non-increasing number of generated inequalities: elimination takes place only if the variable to be projected appears less than two positive (or negative) coefficients. We can try the elimination in the case of two coefficients of same sign of the variable, but this heuristic is simple and works well in practice, knowing that after experiments, we can justify whether the strategy chosen is appropriate or not.

Introducing constrained variables with dummy elimination. Suppose an inequality includes a free variable whose coefficient has a unitary value (± 1), for instance x_1 :

$x_1 + a_{i2}x_2 + ... + a_{ij}x_j + ... + a_{in}x_n \leq b_i$

Introducing an integer slack (dummy) variable $y_i \geq 0$, we can replace it by an equality and solve for x_1 :

$$y_i + x_1 + a_{i2}x_2 + ... + a_{ij}x_j + ... + a_{in}x_n = b_i$$

$$x_1 = [b_i - y_i - a_{i2}x_2 - ... - a_{ij}x_j - ... - a_{in}x_n]$$

The system remains all integer, with one fewer inequality.

Single free variable cutting technique. (see [Sch86], page 339) Suppose an inequality includes a single free variable and its coefficient is not unitary, for instance x_1:

$$a_{r1}x_1 + \sum_{j=2}^{n} a_{rj}y_j \leq b_r \tag{4.10}$$

Then, the following inequality, referred to as a "cut" inequality, is satisfied:

$$(a_{r1}/|a_{r1}|)x_1 + \sum_{j=2}^{n} \lfloor a_{rj}/|a_{r1}|\rfloor y_j \leq \lfloor b_r/|a_{r1}|\rfloor \tag{4.11}$$

Since the coefficient of variable x_1 is unitary, we can add the "redundant" constraint 4.11 to the current system and apply a "dummy elimination". Inequality 4.10 is thus transformed:

$$-|a_{r1}|w_1 + \sum_{j=2}^{n}(a_{rj} - |a_{r1}|\lfloor a_{rj}/|a_{r1}|\rfloor)y_j \leq b_r - |a_{r1}|\lfloor b_r/|a_{r1}|\rfloor \tag{4.12}$$

Let us note two points, useful for the final step:

1. The right hand side of the new inequality 4.12 is not negative.
2. The number of inequalities is temporarily unchanged, however it will decrease in the final step in case constrained variable w_1 is involved in a pivoting operation, due to the "cut" feature of w_1.

Constrained Echelon Matrix. In order to apply the previous steps, JANUS chooses to modify the system through *unimodular* transformations on free variables. The process is repeated until one of the following cases occurs:

- No free variable remains;
- Every right hand side is not negative.

Our satisfiability problem is formulated as follows:

$$Ax \leq b \in Z_+^r \tag{4.13}$$

Simplex. In the Simplex method, we can have several strategies, the one in JANUS is to choose the inequality with the most negative right hand side, and the column with the most possible pivots. In fact, if every right hand side is non negative, the existence of a solution is trivial since 0 is a solution. However, if free variables remain, we cannot conclude and keep introducing constrained variables. As we do not need to compute the objective function, we choose the coefficients equal to zero. The pivoting rule is to choose the negative coefficient of an inequality with a negative right hand side, satisfying:

1. The inequality with the most negative "right hand side";

2. The column with the most possible pivots.

7.2.4 Dual Conversion

There is also the algorithm using dual conversion in order to detect the satisfiability of a polyhedron, which is presented in Section 6, Page 91.

7.3 Practical Problems

In the theoretical view, we showed that Fourier-Motzkin is appropriate for constraint systems of small sizes and constraint systems that are sparse, because of its great upper bound in memory space ($O(2^n)$ in term of number of constraints), while Simplex with its manipulation of a large table can be more adaptable for larger constraint systems. We can only compare these two methods with the dual conversion method by experiments, since their algorithms are very different.

In practice, the number of inequalities of a constraint system is by far the major problem for algorithms. JANUS attempts to reduce the number of inequalities, by some pre-processing. However, the used techniques are not applicable to all classes of systems: they need to satisfy some conditions. Thus, as presented in Chapter 5, JANUS encounters problems with large systems (Chapter 5, Section 3). Yet, with a loop preventing implementation, JANUS can execute without timeout mechanism, whereas it is necessary for the other methods.

Furthermore, JANUS only finds integer solutions; that means it provides a better precision than Fourier-Motzkin and Simplex: our experiments in Chapter 5 reveal the difference between integer and rational algorithms (Chapter 5, Section 3). Though integer algorithmics is more precise when integer solutions are required, it is in general more expensive than rational one. Note here again that important libraries such as the POLYLIB, New POLKA, and PPL implement only rational algorithms.

In order to use other libraries, format conversion is required and we have to take into account the format conversion time. Moreover, not every implementation supports the same set of arithmetics. In our case, original JANUS is implemented with C built-in algorithmics, which forbids the integration of more generic arithmetics, such as the GNU multi-precision, without an entire remake.

Our contribution to JANUS is its rewriting with a generic interface to enable the 64-bit computations that show great benefits in practice: studies on precision lost are presented with the comparison between JANUS 32-bit and 64-bit in Chapter 5.

In case of exceptions, we do not have any backup algorithm which can assure a better running time, since this operator only returns a boolean answer. However, but we can propose an equivalent operator which implements a different algorithm. For example, if the Simplex method raises an exception, we change to Fourier-Motzkin method.

8 Projection

8.1 Introduction

The *Projection* operation along one dimension using constraint system, uses the Fourier-Motzkin's algorithm to eliminate the *variable* corresponding to that dimension by computing the convex hull of the projections of all the rational points that belong to the initial polyhedron. The result is a convex polyhedron that may contain elements that are not the projection of an integer point. We can use some necessary and sufficient conditions for testing the integer exactness of the elimination of one variable [AI91, PUG92].

If we use a generating system, we can simply remove the coefficients concerning the dimension, on condition that the dimension to be projected on is completely independent of the other dimensions.

The projection operator is important to reduce the number of analyzed variables, to eliminate uninteresting variables. There are different algorithms implementing this operator, dealing with the H-representation and/or V-representation of the polyhedra.

8.2 Available Algorithms

Projection using Fourier-Motzkin elimination. In C^3 [PIPa], the Fourier-Motzkin elimination algorithm is implemented, which operates directly on the constraint systems. A description of the algorithm has been given in Section 7.2.1 Page 96. Libraries such as New POLKA and PPL, which use mostly generating systems, implement dual conversion for the projection operator.

Projection using Dual Conversion. The basis containing all the variables of the space dimension is supposed linearly independent. So, given a polyhedron P in D^n with basis $\{e_1, ..., e_n\}$, we have:

$$P = \{x | P(x), x \in D, x = a_1e_1 + ... + a_ne_n\}$$

Let $W = w_1, ...w_k$ be a subset of the basis, then the projection of P from D^n to W is the projection x' of all elements x in P from D^n to W such that:

$$\begin{aligned} x' &= projection_W(x) \\ &= \textstyle\sum_{i=1}^{k} < x, w_i > w_i \end{aligned}$$

Given Definition 3.2 (Chapter 3, Page 31), the dual conversion permits an easy implementation of projection operators, thanks to Theorem 8.1 [44]:

THEOREM 8.1

$$\begin{aligned} P' &= projection_W(P) \\ &= projection_W(v_1, \ldots, v_\alpha) + projection_W(r_1, \ldots, r_\beta) + projection_W(l_1, \ldots, r_\gamma) \end{aligned}$$

In order to project the polyhedron P along one dimension, in the form of a generating system, we can directly remove the coefficients corresponding to that dimension, and its elements (points, rays or lines).

Since the dual conversion using Chernikova's algorithm only deals with rational solutions, the projection implemented in most of existing libraries is not integer, whereas the modified Fourier-Motzkin elimination method implemented in C^3 supplies a preferable integer/rational precision: exact integer one when possible.

However, the projection using the dual conversion assures a reduced size of the resulting polyhedron in its form of vertices, rays and lines, when Fourier-Motzkin sometimes raises problems in size, which requires a minimization with redundancy removal, and possible an unacceptable running time, especially when the input size is large enough. Anyway, a generating system with reduced size does not imply that the size of its constraint system is reduced, too.

A comparison between these two algorithms has unfortunately not been yet conducted. Besides, in PIPS [PIPb], where large size problems are often encountered, the advantage of the reduced size using dual conversion projection is not yet exploited. Instead, a process of size controlling and timeout mechanism is implemented.

Finally, it is still an open question whether one can determine a suitable order of projections when we need to project a list of variables while caring about size explosion. Actually, these projections are handled without care of orders, even though extracted examples show that sometimes we can prevent exceptions by changing the order of projections.

8.3 Practical Problems

In practice, projection operators encounter explosion in memory space, magnitude overflows and unacceptable running time, with both the Fourier Motzkin elimination method

[44] The proof can be found in the Appendix page 189.

and the double description method. Serious consideration must be made to avoid these problems in our context of use.

When an exception is not avoidable, approximations are needed. But in fact, among existing polyhedral implementations, there is no work concerning this aspect. In general, to deal with exception returned by polyhedral computation, programmers have to approximate in a very limited way, depending on the context. For example, when a projection fails, a common way is to remove all the constraints that have a non-zero coefficient in the dimension to be projected. Of course, we can as well remove only the positive ones, or just the negative ones. It has many advantages like it is simple and rapid, but it is not flexible, and sometimes we lose all the information (i.e. when the variable appears in all the constraints of the system) [45]. Finally, in order to include these algorithms in a library, we have to justify these choices experimentally.

Existing polyhedral libraries that handle projections are New POLKA ([JEAb]), PPL ([PAR]), C^3 ([PIPa]), OMEGA ([Ome]). Other libraries, such as POLYLIB ([LOE]) CONVEX ([Fra]), PORTA ([TC]) do not implement any projection operator.

As mentioned above, dual conversion is used in New POLKA [JEAb] and PPL [PAR], whereas direct manipulation of constraints is chosen in C^3 [PIPa]. Actually, New POLKA and PPL implement their own version of the dual conversion, with rational precision, whereas C^3's implementation is integer/rational precision. This approach permits an easy integration of other implementations for dual conversion such as LRS ([Avi]), CDD ([FUK]), which may provide a better overall performance. In general, CDD tends to be efficient for highly degenerate inputs and LRS tends to be efficient for non degenerate or slightly degenerate problems.

Incompatibilities among libraries implementing projection operators make experimental comparisons more difficult. For example, CDD uses floating point arithmetics, which gives faster computation but very low precision, while the others uses integer arithmetics, thus their running time is higher.

Interestingly, for object-oriented reasons, C^3's interface exposes only constraint systems and hides generating systems, so most of its operators use algorithms dealing with constraints (except dual conversion for example). Hence, the implementation of a projection operator using dual conversion [46], while been used by New POLKA and PPL long ago, is not available in C^3.

Because the double description method assures the minimal form of its results, the impact on analysis size could be considerable. Nonetheless, as the dual conversion is an expensive operation itself, the running time and possibility of exceptions are to be studied. We compare the impact of these two approaches in our experiments, Chapter 5, Section 6.

The projection operator can return exceptions with both algorithms. Using Fourier-Motzkin algorithm when the variable to be projected appears in many inequalities with

[45] Several approximations for the projection are presented in Appendix page 189.
[46] We have proven the validity of this approach in Appendix page 189.

opposite sign coefficients, the combination of inequalities generates many new constraints. On the contrary, the dual conversion itself is an expensive operation (Section 6, Page 91).

9 Minimization

9.1 Introduction

Minimization addresses redundancy in polyhedral representations: redundant constraints in constraint systems and redundant rays, lines or points in generating system. Besides detecting redundancy, it tries as well to reduce coefficients of the polyhedral representations.

The most important part of the *minimization* is redundancy removal. However, for different purposes, three types of minimization on polyhedral representations have been established. For an ordered space dimension, only a *canonicalization* can assure the *uniqueness* of the two presentations of a polyhedron. Note that this uniqueness is relative, since each polyhedron can have several possible representations. This operation is quite expensive, because of redundancy removal, lexicographical sort, etc. Hence, instead of canonicalization, *minimization* is implemented, where it on one hand attempts to remove the redundant constraints, or the redundant vertices, rays and lines, and on the other hand, *normalize* coefficients (i.e. divided by greatest common divisor).

It sometimes happens that a representation that occupies the least physical memory space is not necessarily in its minimized or canonicalized form. In New POLKA library ([JEAb]) a version of minimization that reduces the physical size of representation is applied. We call it a *reduction* minimization.

In practice, because of minimization's complexity, several degrees of minimization can be applied. For example, a simple minimization deals only with the coefficient problem, or removal of some obvious constraints such as $x <= 1$ when we already have $x <= 0$. Often, one needs to consider when a minimization is needed (i.e. we do not have to apply the minimization all the time). A common example that illustrates well this idea is that one might want to apply a minimization before an equality test between two polyhedra, but for a satisfiability test, the minimization is indeed not necessary.

9.2 Available Algorithms

There are two main algorithms for minimization; one uses the dual conversion based on Chernikova's algorithm (e.g. New POLKA [JEAb], PPL [PAR]), while the other acts on constraints. The first one is based on a property of the double description method, such that the generating system converted from its constraint system is minimized, so if we convert it back, we have a minimized constraint system.

The second one checks if a constraint is redundant with the rest of the system, one by one. In this case, depending on the complexity of a particular situation, different levels of minimization can be applied. Though it is confusing to have such implementations for minimization operator, experiences show that it is profitable, especially when exceptions occur. Furthermore, as the dual conversion is expensive, we sometimes need to approximate the minimization.

9.3 Practical Problems

Problems concerning the minimization in practice is of reality, namely exceptions of magnitude overflows, of memory space and of unacceptable running time. It becomes disastrous when polyhedra of large size appear in analyses. In an example of PIPS execution on applu.f in SPEC CFP 95 benchmark ([SPE]) calculating interprocedural transformers, preconditions and array regions (see Chapter 2, Section 4), a constraint system with 214920 inequalities is passed to the minimization operator, and constraint systems five times larger than that occur after some iterations (the example can be found in Chapter 5, Section 1.2). The analysis is then blocked for several hours, before memory space problems occur. Thus, a mechanism that permits flexible application of minimization is of interest. However, with existing algorithms, we can see it is not evident.

Backup algorithms for the minimization operator should be designed to deal with exceptions. Actually, a common way to proceed is to do nothing, just return the non-minimized concerned polyhedron. Otherwise, a size controlling mechanism should take place, in order to predict the size of returned polyhedra and the running time of the operator.

Like the other operators, the existing implementations of minimization share incompatibilities regarding computational precision, and differences between integer and rational precision. Improvement such as decomposition of polyhedra can be integrated into minimization algorithms, when the size of polyhedra is important.

The aforementioned approaches of minimization permit an interesting comparison in performance, as well as tests of regression (i.e. comparison if there is a bug in an implementation) that we study in Chapter 5.

10 Convex hull

10.1 Introduction

The *Union* of two (or more) convex polyhedra is not necessarily convex, therefore the polyhedral domain using union operator does not comply with a lattice's requirement (e.g. uniqueness of element is missing). The smallest convex over-approximation of the union of two polyhedra A and B is their *Convex Hull*, denoted $A \sqcup B$. Thus, it may contain points that do not belong to the original polyhedra. Nonetheless, we can verify the exactness of the convex hull compared to their union.

This operator has unfortunately a high complexity. In fact, convex hull computation is one of the most time consuming operation in the polyhedral domain.

10.2 Available Algorithms

The method to compute the dual conversion of a polyhedron can be used to find the convex hull of polyhedra. From the V-representation, generating systems can be merged and sorted into one generating system. Then the result can be convert to constraint system form if needed. All the algorithms presented in Section 6, Page 91, can be used for this operator, but some algorithms do not return minimized results .

Main general-purpose libraries that include a convex hull of two polyhedra are POLYLIB ([LOE]), New POLKA ([JEAb]), PPL ([PAR]) and C^3 ([PIPb]). They all implement the Chernikova's algorithm for this operator. It is interesting to notice that actual implementations only deal with two polyhedra, so convex hull of more than two polyhedra have to be applied via several calls of convex hull for two polyhedra. It raises a problem of efficiency, knowing that most of above libraries have constraint systems as input.

Consider now three constraint systems A, B and C. If we compute the convex hull of A and B to obtain a constraint system D, in order to pass it and C to the same operator, we have to compute two more duals, which is not necessary because we can indeed merge several generating systems together into the final one, and then convert it to constraint system form.

There are many efforts in order to improve the strategy for calculating the convex hull; however they are all based on Chernikova's algorithm, thus only minor modifications took place. The first C-implementation for the algorithm by POLYLIB is reused in C^3, or re-implemented in New POLKA. A C++ version is developed in PPL library.

In the C^3 polyhedral library, there are two versions of the convex hull operator: a simple call to POLYLIB's function and a *partial factorization* version. This partial factorization tries to detect a common part of two inputs, therefore sometimes reducing the size of the polyhedra passed to Chernikova's algorithm.

For the same purpose, an extension for version 0.3 of PPL library integrates the polyhedral *Cartesian product* in order to speed up the computation by finding a good decomposition of polyhedra ([HMPV03]). In the next two sections, we study these approaches in more details.

10.2.1 Decomposition defined by Corinne ANCOURT and Fabien COELHO

In this section, we describe the decompostion of polyhedra defined by two PIPS members. This technique is used in C^3, in order to improve the convex hull calculation. When programs are analyzed, the same constraints appear again and again. The intuition behind this decomposition of two constraint systems P_1 and P_2 is to capitalize on P, the set of constraints shared by P_1 and P_2 before computing the convex hull of P_1 and P_2 in Q^n. When the sizes are reduced, the computation is faster. Moreover, exceptions for overflows or out-of-memory space are fewer.

Let $P_1 = P \cap X_1$ and $P_2 = P \cap X_2$. Then hopefully, $P_1 \sqcup P_2 = P \cap (X_1 \sqcup X_2)$, where

$\sqcup$ denotes the convex hull operator. This holds if the constraint matrix for P, X_1 and X_2 can be put in block form:

$$\begin{pmatrix} X_1 & 0 \\ X_2 & 0 \\ 0 & P \end{pmatrix} \tag{4.14}$$

The proof is based on Theorem **??** defining the generating system of a polyhedron with a block-decomposed constraint matrix:

$$XP = \begin{pmatrix} X & 0 \\ 0 & P \end{pmatrix}$$

If

$$gs\begin{pmatrix} X \\ 0 \end{pmatrix} = \left(\begin{pmatrix} xv_i \\ 0 \end{pmatrix}, \begin{pmatrix} xr_j \\ 0 \end{pmatrix}, \begin{pmatrix} xl_k \\ 0 \end{pmatrix}\right)$$

and

$$gs\begin{pmatrix} 0 \\ P \end{pmatrix} = \left(\begin{pmatrix} 0 \\ pv_l \end{pmatrix}, \begin{pmatrix} 0 \\ pr_m \end{pmatrix}, \begin{pmatrix} 0 \\ pl_n \end{pmatrix}\right)$$

where gs stands for generating system, then the generating system of XP is:

$$gs(XP) = ((xv_i + pv_l), (xr_j) \cup (pr_m), (xl_k) \cup (pl_n)) \tag{4.15}$$

Note that the number of vertices increases much faster than the numbers of rays and lines.

This block form can be obtained by matrix transformation. P is broken down into P' and P''. If a constraint c in P and a constraint in X_1 or X_2 have both a non-zero coefficient in the same dimension, c belongs to P''. Thus, P' contains the constraints in P that do not belong to P''.

In fact, this depends on the chosen basis, for example:

$$P_1 = \{(x, y)|x + y = 2, x - y = 0, x + y \leq 10\}$$

$$P_2 = \{(x, y)|x + y = 2, x - y = 0, x + y \geq 10\}$$

$$P = \{(x, y)|x + y = 2, x - y = 0\}$$

$$P'' = \{(x, y)|x + y = 2, x - y = 0\}, P' = \{(x, y)\}$$

If we change basis and use $u = x + y$ and $v = x - y$, then P' is constrainted by:

$$P = \{(x, y)|x - y = 0\}$$

As a result, for simplicity, we consider that P, X_1 and X_2 satisfy 4.14. In Figure 4.4, we have another example that shows the 4 points A, B, C and D representing the vertices of the generating system in 4.15. Each point can be computed using the generating system, for example: $C = \begin{pmatrix} xv_1 \\ 0q \end{pmatrix} + \begin{pmatrix} 0 \\ pv_1 \end{pmatrix}$. Thus, the rectangle $ABCD$ represents the convex hull XP.

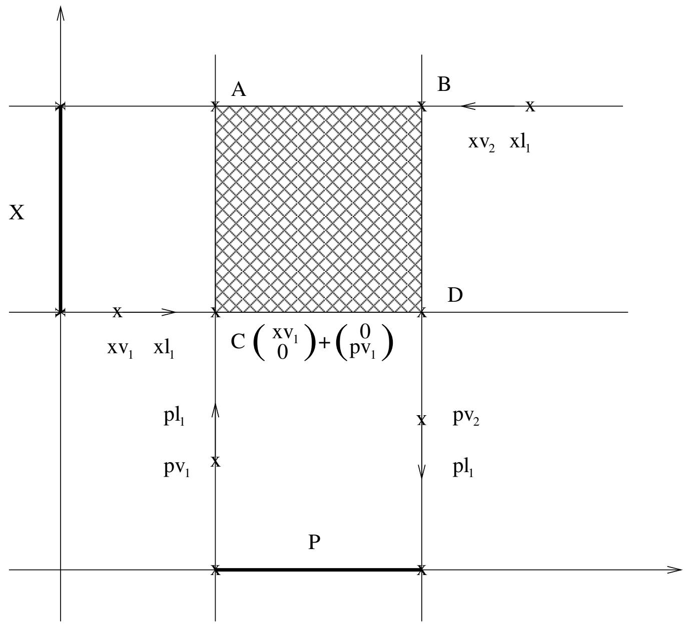


Figure 4.4: Example of the decomposition of two polyhedra

The proof of the distributivity,

$$(P \cap X_1) \sqcup (P \cap X_2) = P \cap (X_1 \sqcup X_2)$$

is performed in showing that:

$$gs((P \cap X_1) \sqcup (P \cap X_2)) = gs(P \cap (X_1 \sqcup X_2))$$

using the fact that both X_1 and X_2 meet the condition of the previous theorem with respect to P.

The decomposition algorithm by Ancourt and Coelho computes firstly C, the set of constraints common to P_1 and P_2: $cs(C) = cs(P_1) \cap cs(P_2)$ where cs stands for constraint system. This is a syntactic operation, whose result depends on the sets of constraints used for P_1 and P_2 and on the basis. The rational is that program analysis should not generate very different constraints.

DEFINITION 10.1 Then $cs(P)$ is defined recursively as the maximal fixed point of:

$$cs(P) = \left\{ c \in cs(C) | \begin{array}{l} \forall k_1 \in cs(P_1) - cs(P), \\ \forall k_2 \in cs(P_2) - cs(P), \\ \forall i \in [1, n], \\ c_i = 0 \vee (k_{1i} = 0 \wedge k_{2i} = 0) \end{array} \right\}$$

We need to find the maximum set of constraints $cs(P)$ which are independent from the others. This process is valid because the above recursive definition gives us monotonously decreasing iterations starting from C, thus its fixed point exists. In the worst case, the result is the empty set $\emptyset$ which is a solution.

THEOREM 10.1 Let A and B be two polyhedra of D^n. If their constraint matrices can be put in block form:

$$\begin{pmatrix} A & 0 \\ 0 & B \end{pmatrix} x \leq \begin{pmatrix} a \\ b \end{pmatrix}$$

then the generating system of $A \cap B$ is defined from the generating system of A and B:

$$gs(A \cap B) = (\{av_i + bv_l\}, \{ar_j\} \cup \{br_m\}, \{al_k\} \cup \{bl_n\})$$

PROOF. If $z \in A \cap B$ we have:

$$z \in A : z = \sum_i \mu_i \begin{pmatrix} av_i \\ 0 \end{pmatrix} + \sum_j d_j \begin{pmatrix} ar_j \\ 0 \end{pmatrix} + \sum_k c_k \begin{pmatrix} al_k \\ 0 \end{pmatrix} + \sum_b f \begin{pmatrix} 0 \\ e_b \end{pmatrix}$$

where e_b is the basis vector of B. Lower coefficients can be zeroed with f and e_b's vector.

$$z \in B : z = \sum_l \nu_l \begin{pmatrix} 0 \\ bv_l \end{pmatrix} + \sum_m g_m \begin{pmatrix} 0 \\ br_m \end{pmatrix} + \sum_n h_n \begin{pmatrix} 0 \\ bl_n \end{pmatrix} + \sum_a f'_k \begin{pmatrix} e_a \\ 0 \end{pmatrix}$$

where e_a is the basis vector of A. Lower coefficients can be zeroed with f' and e_a's vector.

The terms in f and f' can be computed thanks to the block decomposition and to the above two definitions of z.

Hence:

$$\begin{array}{rl} z = & \sum_i \mu_i \begin{pmatrix} av_i \\ 0 \end{pmatrix} + \sum_l \nu_l \begin{pmatrix} 0 \\ bv_l \end{pmatrix} + \sum_j d_j \begin{pmatrix} ar_j \\ 0 \end{pmatrix} + \sum_m g_m \begin{pmatrix} 0 \\ br_m \end{pmatrix} + \\ & \sum_k c_k \begin{pmatrix} al_k \\ 0 \end{pmatrix} + \sum_n h_n \begin{pmatrix} 0 \\ bl_n \end{pmatrix} \end{array}$$

This almost fits the definition of a generating system for $A \cap B$, but $\sum_i \mu_i \sum_l \nu_l = 2$ instead of 1.

The next step is to show:

$$\sum_i \mu_i \begin{pmatrix} av_i \\ 0 \end{pmatrix} + \sum_l \nu_l \begin{pmatrix} 0 \\ bv_l \end{pmatrix} = \sum_i \sum_l \mu_i \nu_l \begin{pmatrix} av_i \\ bv_l \end{pmatrix}$$

and $\sum_i \sum_l \mu_i \nu_l = 1$. Indeed:

$$\begin{aligned}
\textstyle\sum_i \sum_l \mu_i \nu_l \begin{pmatrix} av_i \\ bv_l \end{pmatrix} &= \textstyle\sum_i \sum_l \mu_i \nu_l \left(\begin{pmatrix} av_i \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ bv_l \end{pmatrix} \right) \\
&= \textstyle\sum_i \mu_i \begin{pmatrix} av_i \\ 0 \end{pmatrix} \sum_l \nu_l + \sum_l \nu_l \begin{pmatrix} 0 \\ bv_l \end{pmatrix} \sum_i \mu_i \\
&= \textstyle\sum_i \mu_i \begin{pmatrix} av_i \\ 0 \end{pmatrix} + \sum_l \nu_l \begin{pmatrix} 0 \\ bv_l \end{pmatrix}
\end{aligned}$$

since $\sum_i \mu_i = 1$ and $\sum_l \nu_l = 1$.

In the same way: $\sum_i \sum_l \mu_i \nu_l = \sum_i \mu_i \sum_l \nu_l = \sum_i \mu_i = 1$. □

THEOREM 10.2 Let P_1 and P_2 be two polyhedra whose constraint matrices can be converted to block form as:

$$\begin{pmatrix} X_1 & 0 \\ O & P \end{pmatrix}$$

and

$$\begin{pmatrix} X_2 & 0 \\ O & P \end{pmatrix}$$

then $P \cap (X_1 \sqcup X_2) = (P \cap X_1) \sqcup (P \cap X_2)$. Note here that, without lost of generality, we have hidden the fact that the constraints are actually in form $Ax \leq b$.

PROOF. By definition of convex hull we have:

$$gs(X_1 \sqcup X_2) = ((x_1 v \cup x_2 v), x_1 r \cup x_2 r, x_1 l \cup x_2 l)$$

$$gs(P \cap X_1) = (\begin{pmatrix} x_1 v_i \\ p v_l \end{pmatrix}, \begin{pmatrix} x_1 r \\ 0 \end{pmatrix} \cup \begin{pmatrix} 0 \\ pr \end{pmatrix}, \begin{pmatrix} x_1 l \\ 0 \end{pmatrix} \cup \begin{pmatrix} 0 \\ pl \end{pmatrix})$$

$$\begin{aligned} gs(P \cap (X_1 \sqcup X_2)) &= (\begin{pmatrix} x_1 v_i \\ p v_l \end{pmatrix} \cup \begin{pmatrix} x_2 v_i \\ p v_l \end{pmatrix}, \begin{pmatrix} x_1 r \\ 0 \end{pmatrix} \cup \begin{pmatrix} x_2 r \\ 0 \end{pmatrix} \cup \begin{pmatrix} 0 \\ pr \end{pmatrix}, \\ & \quad \begin{pmatrix} x_1 l \\ 0 \end{pmatrix} \cup \begin{pmatrix} x_2 l \\ 0 \end{pmatrix} \cup \begin{pmatrix} 0 \\ pl \end{pmatrix}) \\ &= gs((P \cap X_1) \sqcup (P \cap X_2)) \end{aligned}$$

□

In C^3, the implementation of this algorithm has shown a better performance of the convex hull operator. Thanks to our benchmarks, it is justified in Chapter 5, Section 7.

10.2.2 Cartesian Factorization by Nicolas Halbwachs and al.

Nicolas HALBWACHS and al. in [HMPV03] propose to detect when a polyhedron is a Cartesian product of polyhedra of lower dimensions, i.e., when groups of variables are unrelated to each other. Whereas the partial factorization mentioned above is only applied for the convex hull operator, the Cartesian factorization can be used in other operators as well.

We present here the definition of Cartesian factorization of polyhedra, and the problem concerning the convex hull operator. For more detail on other operators, readers are referred to the paper [HMPV03].

DEFINITION 10.2 Let I be a subset of the index set $\{1 \ldots n\}$. We denote $P \downarrow I$ the projection of the polyhedron P on variables with indices in I (i.e. the result in $Z^{|I|}$ of the existential quantification of all the variables with indices outside I).

Let $(I_1, I_2, \ldots I_l)$ be a *partition* of $\{1 \ldots n\}$. We say that a polyhedron P can be *factorized* according to $(I_1, I_2, \ldots I_l)$ if and only if:

$$P = P \downarrow I_1 \times P \downarrow I_2 \times \ldots \times P \downarrow I_l$$

We also note $p = (p_1, p_2, \ldots p_l)$ for each element $p \in P$, where $p_1 \in P \downarrow I_1, p_2 \in P \downarrow I_2, \ldots p_l \in P \downarrow I_l$. We say a element p consists of many components of some dimensions.

DEFINITION 10.3 A matrix A is *block-diagonalizable* according to a partition $(I_1, I_2, \ldots I_l)$ if for each of its rows A_i there is one $k_i \in 1 \ldots l$ such that $\forall j = 1 \ldots n, A_i^j \neq 0 \Rightarrow j \in I_{k_i}$.

We have then, for any polyhedron P, there is a greatest partition $(I_1, I_2, \ldots I_l)$ according to which P can be factorized. Moreover, if (A, B) is the constraint description of a polyhedron P, and if A is block-diagonalizable according to a partition $(I_1, I_2, \ldots I_l)$, then P can be factorized according to $(I_1, I_2, \ldots I_l)$. This gives an easy way to factorize a polyhedron, and to get the constraint description of its factors (i.e. polyhedra of lower dimensions): each constraint becomes a constraint of the factor P_{k_i}. Finally, for any pair P_1, P_2 of polyhedra, there is a greatest common partition according to which both polyhedra can be factorized (possible the trivial partition with $l = 1$).

The convex hull of two factorized polyhedra can be either less factorized, as factorized or more factorized than the operands. A proposition is introduced for the computation of the convex hull operator, with multiple factorizations.

PROPOSITION 10.1 Let $P = P_1 \times \ldots \times P_l$ and $P' = P'_1 \times \ldots \times P'_l$ be two polyhedra factorized according to the same partition. Let λ be a fresh variable and let us consider the polyhedra $(Q_k)_{k=1\ldots l}$ defined by:

$Q_k = (P_k \cap \{\lambda = 0\}) \vee (P'_k \cap \{\lambda = 1\}$

where $\cap$ and $\sqcup$ denote the intersection and convex hull operator. Then, the partition of $P \sqcup P'$ is obtained from $(I_1, I_2, \ldots I_l)$ by merging I_k and $I_{k'}$ whenever either λ is lower-bounded by a non constant expression in Q_k and upper-bounded by a non constant expression in $Q_{k'}$, or conversely.

Let $(J_1, J_2, \ldots J_h)$ be the resulting partition, each J_m being an union of some I_k, then: $P \vee P' = R_1 \times \ldots \times R_h$, where $R_m = \exists\lambda, \times_{I_k \subseteq J_m} Q_k$.

Based on this proposition, implementation for convex hull as well as other polyhedral operators has been integrated in an extension for PPL library. Unfortunately, only version 0.3 is extended, due to changes made to more recent versions of the PPL library.

10.2.3 Decomposition by Inclusion Test

DEFINITION 10.4 Let Y and Z be two polyhedra. The *convex hull* of these two polyhedra is a polyhedron denoted $Y \sqcup Z$ that satisfies:

$\forall x \in Y \sqcup Z$, then $\exists\lambda \in [0, 1], \exists y \in Y, \exists z \in Z$ such that: $x = \lambda y + (1 - \lambda)z$

Sufficient Condition for the Decomposition of Two Polyhedra

LEMMA 10.1 Let Y and Z be two polyhedra that can be factorized according to the same partition (I_1, I_2), so that we can write: $Y = Y_1 \times Y_2$ and $Z = Z_1 \times Z_2$, knowing that Y_1 and Z_1 have the same dimensions (variables), and similarly for Y_2 and Z_2. Then we have:

$$(Y_1 \times Y_2) \sqcup (Z_1 \times Z_2) \subseteq (Y_1 \sqcup Z_1) \times (Y_2 \sqcup Z_2) \tag{4.16}$$

PROOF. Let us call $X = (Y_1 \times Y_2) \sqcup (Z_1 \times Z_2)$, $X_1 = Y_1 \sqcup Z_1$ and $X_2 = Y_2 \sqcup Z_2$, we need to prove that $X \subseteq X_1 \times X_2$.

By definition of convex hull, we have: $\forall x \in X$, then $\exists \lambda \in [0,1], \exists x_1 \in (Y_1 \times Y_2), \exists x_2 \in (Z_1 \times Z_2)$ such that:

$$x = \lambda x_1 + (1-\lambda)x_2$$

Since $x_1 \in (Y_1 \times Y_2)$, $\exists y_1 \in Y_1, \exists y_2 \in Y_2$ such that $x_1 = (y_1, y_2)$. Similarly we have: $\exists z_1 \in Z_1$ and $\exists z_2 \in Z_2$ such that $x_2 = (z_1, z_2)$. Hence:

$$\begin{aligned} x &= \lambda(y_1, y_2) + (1-\lambda)(z_1, z_2) \\ &= (\lambda y_1, \lambda y_2) + ((1-\lambda)z_1, (1-\lambda)z_2) \\ &= (\lambda y_1 + (1-\lambda)z_1, \lambda y_2 + (1-\lambda)z_2) \end{aligned}$$

Thus, we can write $\forall x \in X$, then $\exists \lambda \in [0,1], y_1 \in Y_1, y_2 \in Y_2, z_1 \in Z_1$ and $z_2 \in Z_2$ such that:

$$x = (\lambda y_1 + (1-\lambda)z_1, \lambda y_2 + (1-\lambda)z_2)$$

Furthermore, by definition of factorization: $\forall x' \in X_1 \times X_2$, then $\exists x'_1 \in X_1, x'_2 \in X_2$, such that $x' = (x'_1, x'_2)$.

Thus, we have: $\forall x' \in X_1 \times X_2, \exists \mu \in [0,1], y'_1 \in Y_1, y'_2 \in Y_2, \nu \in [0,1], z'_1 \in Z_1, z'_2 \in Z_2$ such that:

$$x' = (\mu y'_1 + (1-\mu)z'_1, \nu y'_2 + (1-\nu)z'_2)$$

We need to prove that for every point x belonging to X, it also belongs to $X_1 \times X_2$. Indeed, $\forall x \in X$, if we choose $\mu = \lambda, \nu = \lambda, y'_1 = y_1, y'_2 = y_2, z'_1 = z_1, z'_2 = z_2$, we have $x' \equiv x \in X_1 \times X_2$ because:

$$\begin{aligned} x' &= (\mu y'_1 + (1-\mu)z'_1, \nu y'_2 + (1-\nu)z'_2) \\ &= (\lambda y_1 + (1-\lambda)z_1, \lambda y_2 + (1-\lambda z_2) \\ &= x \end{aligned}$$

This means $X \subseteq X_1 \times X_2$. $\square$

This lemma can be used to define a new, fast and precise algorithm to compute the convex hull by Cartesian decomposition. It is interesting to compare this new approach with the algorithm of [HMPV03].

THEOREM 10.3 Let Y and Z be two polyhedra that can be factorized according to the same partition (I_1, I_2): $Y = Y_1 \times Y_2$ and $Z = Z_1 \times Z_2$, where Y_1 and Z_1 share the same space dimensions, similarly for Y_2 and Z_2. We have:

$$(Y_1 \times Y_2) \sqcup (Z_1 \times Z_2) = (Y_1 \sqcup Z_1) \times (Y_2 \sqcup Z_2) \tag{4.17}$$

if:

$$((Z_1 \subseteq Y_1) \vee (Y_2 \subseteq Z_2)) \wedge ((Y_1 \subseteq Z_1) \vee (Z_2 \subseteq Y_2)) \tag{4.18}$$

where $\vee$ and $\wedge$ are the logical *or* and *and* operators.

PROOF. We need to prove that 4.18 is a sufficient condition for the decomposition of two polyhedra 4.17. By distributivity, 4.18 can be rewritten as:

$$(Z_1 = Y_1) \vee ((Y_1 \subset Z_1) \wedge (Y_2 \subset Z_2)) \vee ((Z_1 \subset Y_1) \wedge (Z_2 \subset Y_2)) \vee (Z_2 = Y_2) \tag{4.19}$$

Let us call $X = (Y_1 \times Y_2) \sqcup (Z_1 \times Z_2)$, $X_1 = Y_1 \sqcup Z_1$ and $X_2 = Y_2 \sqcup Z_2$. Thanks to Lemma 10.1, to show the sufficiency, we only need to prove that $X_1 \times X_2 \subseteq X$. This means that every point $x' \in X_1 \times X_2$ belongs to X.

We have $\forall x' \in X_1 \times X_2$, then $\exists \mu \in [0,1], \exists \nu \in [0,1], \exists y_1' \in Y_1, \exists y_2' \in Y_2, \exists z_1' \in Z_1, \exists z_2' \in Z_2$ such that:

$$x' = (\mu y_1' + (1-\mu) z_1', \nu y_2' + (1-\nu) z_2')$$

The condition 4.19 is divided into four cases:

1. $(Y_1 = Z_1)$: If we choose $\lambda = \nu, y_2 = y_2', z_2 = z_2', y_1 = z_1 = \mu y_1' + (1-\mu) z_1' \in Y_1 = Z_1$, we have:

$$\begin{aligned} x' &= (\mu y_1' + (1-\mu) z_1', \nu y_2' + (1-\nu) z_2') \\ &= (y_1, \lambda y_2 + (1-\lambda) z_2) \\ &= (\lambda y_1 + (1-\lambda) z_1, \lambda y_2 + (1-\lambda) z_2) \in X \end{aligned}$$

2. $(Z_2 = Y_2)$: The same by symmetry.

3. $((Z_1 \subset Y_1) \wedge (Z_2 \subset Y_2))$: Let us take $\lambda = 1, y_1 = \mu y_1' + (1-\mu) z_1', y_2 = \nu y_2' + (1-\nu) z_2', z_1 = z_1', z_2 = z_2'$, then:

$$\begin{aligned} x' &= (\mu y_1' + (1-\mu) z_1', \nu y_2' + (1-\nu) z_2') \\ &= (y_1, y_2) \\ &= (\lambda y_1 + (1-\lambda) z_1, \lambda y_2 + (1-\lambda) z_2) \in X \end{aligned}$$

4. $((Y_1 \subset Z_1) \wedge (Y_2 \subset Z_2))$: The same by symmetry.

□

LEMMA 10.2 Given two polyhedra Y and Z. If there exists a point $y_0 \in Y \setminus Z$ (i.e. $y_0 \in Y$ and $y_0 \notin Z$), then there exists at least an extremal element (vertex, ray or line) of Y that does not belong to Z.

PROOF. Since $y_0 \in Y$ and Definition 3.2 (Chapter 3, Page 31), there exists a generating system of Y such that:

$$y_0 = \sum_{i=1}^{\alpha} \lambda_i v_i + \sum_{i=1}^{\beta} \mu_i r_i + \sum_{i=1}^{\gamma} \nu_i l_i$$

If all $v_i \in Z$, $r_i \in Z$ and $l_i \in Z$, then $y_0 \in Z$, which is a contradiction. Hence:

$$(\exists v_i \notin Z) \vee (\exists r_i \notin Z) \vee (\exists l_i \notin Z)$$

□

Necessary Condition for the Decomposition of Two Polytopes

LEMMA 10.3 Given two polytopes Y and Z. If there exists a point $y_0 \in Y \setminus Z$ (i.e. $y_0 \in Y$ and $y_0 \notin Z$), then there exists a vertex $v_e \in Y \setminus Z$ such that v_e is not a convex combination of other points in Y and Z.

PROOF. From Lemma 10.2, since Y is a bounded polyhedron, there exists a vertex $v_e \in Y \setminus Z$.

Suppose that v_e is a convex combination of other points in Y and Z, then it is a convex combination of a generating system of Y and of a generating system of Z:

$$v_e = \sum_i \lambda_i v_i + \sum_j \mu_j z_j \tag{4.20}$$

where $v_i \in Y$, $z_j \in Z$, $\lambda_i, \mu_j \geq 0$, $\mu_k > 0$ for some k and $\sum_i \lambda_i + \sum_j \mu_j = 1$.

Then there exists a vertex $v'_e \in \{v_i \mid v'_e \neq v_e, v'_e \in Y \setminus Z\}$. Otherwise we have $\forall v'_e \in \{v_i \mid v'_e = v_e, v'_e \in Y \setminus Z\}$ which leads to a contradiction because if we denote $\{v_i \mid v_i = v_e, v_i \in Y \setminus Z\}$ by $\{v_{i1}\}$ and $\{v_i \mid v_i \notin Y \setminus Z\}$ by $\{v_{i2}\}$, we have:

$$\begin{aligned} v_e &= \sum_i \lambda_i v_i + \sum_j \mu_j z_j \\ &= \sum_{i1} \lambda_{i1} v_e + \sum_{i2} \lambda_{i2} v_{i2} + \sum_j \mu_j z_j \end{aligned}$$

where $v_{i2} \notin (Y \setminus Z)$ means $v_{i2} \in Z, \forall i2$ (since $v_{i2} \in Y$, $\forall i2$). If $\sum_{i1} \lambda_{i1} \neq 1$, then:

$$v_e = \frac{1}{1 - \sum_{i1} \lambda_{i1}} (\sum_{i2} \lambda_{i2} v_{i2} + \sum_j \mu_j z_j)$$

thus $v_e \in Z$ since it is a convex combination of points in Z, which is a contradiction. If $\sum_{i1} \lambda_{i1} = 1$, then $\sum_{i2} \lambda_{i2} + \sum_j \mu_j = 0$, which implies $\mu_j = 0, \forall j$. This is also a contradiction.

So $\exists v'_e \neq v_e$ such that $v'_e \in Y \setminus Z$. If v'_e is also a convex combination of other points in Y and Z, then we substitute all future occurrences of v_e by 4.20 in this new combination. Since the number of vertices is bounded, we always end up with a vertex that is not a convex combination of other points in Y and Z.

□

THEOREM 10.4 Let Y and Z be two polytopes that can be factorized according to the same partition (I_1, I_2): $Y = Y_1 \times Y_2$ and $Z = Z_1 \times Z_2$, where Y_1 and Z_1 share the same space dimensions, similarly for Y_2 and Z_2. We have:

$$(Y_1 \times Y_2) \sqcup (Z_1 \times Z_2) = (Y_1 \sqcup Z_1) \times (Y_2 \sqcup Z_2) \tag{4.21}$$

if and only if:

$$((Z_1 \subseteq Y_1) \vee (Y_2 \subseteq Z_2)) \wedge ((Y_1 \subseteq Z_1) \vee (Z_2 \subseteq Y_2)) \tag{4.22}$$

where $\vee$ and $\wedge$ are the logical *or* and *and* operators.

PROOF. Theorem 10.3 gives the sufficient part of this proof, since polytopes are bounded polyhedra. We now prove the necessity: if $X = X_1 \times X_2$ then we have 4.22.

Suppose that $Y_1 \not\subseteq Z_1$, then $\exists p_0 \in Y_1 \setminus Z_1$. According to Lemma 10.3, $\exists p_e$ a vertex of Y_1 such that $p_e \in Y_1 \setminus Z_1$ and p_e is not a convex combination of other points of Y_1 and Z_1.

We have $\forall x' \in X_1 \times X_2$:

$$x' = (\mu y'_1 + (1-\mu) z'_1, \nu y'_2 + (1-\nu) z'_2)$$

where $\mu \in [0,1], y'_1 \in Y_1, y'_2 \in Y_2, \nu \in [0,1], z'_1 \in Z_1$ and $z'_2 \in Z_2$. If we take $\mu = 1, \nu = 0$, then $x' = (y'_1, z'_2)$.

Consider now $y_1' = p_e, \mu = 1, \nu = 0$ and note that Y_1, Z_1 are independent from Y_2, Z_2 (by definition of factorization), so z_2' can span all Z_2.

For all $x \in X$:

$$x = (\lambda y_1 + (1-\lambda)z_1, \lambda y_2 + (1-\lambda)z_2)$$

where $\lambda \in [0,1], y_1 \in Y_1, y_2 \in Y_2, z_1 \in Z_1$ and $z_2 \in Z_2$. Since by hypothesis $X = X_1 \times X_2$, this applies to (p_e, z_2').

Thus there exists λ, $y_1 \in Y_1$ and $z_1 \in Z_1$ such that the chosen value p_e of x' corresponds to $p_e = \lambda y_1 + (1-\lambda)z_1$.

Follow Lemma 10.3, p_e is not a convex combination of other points in Y_1 and Z_1, hence: $p_e = y_1, \lambda = 1$.

Therefore we have then $z_2' = y_2$ where $y_2 \in Y_2$. This applies to all $z_2' \in Z_2$, which expresses that $Z_2 \subseteq Y_2$.

So far, we have shown that:

$$(X = X_1 \times X_2) \Rightarrow ((Y_1 \not\subseteq Z_1) \Rightarrow (Z_2 \subseteq Y_2))$$

which is:

$$(X = X_1 \times X_2) \Rightarrow ((Y_1 \subseteq Z_1) \vee (Z_2 \subseteq Y_2))$$

By symmetry, we also have:

$$(X = X_1 \times X_2) \Rightarrow ((Y_2 \subseteq Z_2) \vee (Z_1 \subseteq Y_1))$$

The necessity is proved. □

Unfortunately, Theorem 10.4 cannot be extended directly to polyhedra. Figure 4.5 presents a counter-example.

$Y_1 = \{2\}, Y_2 = \{2\}$, $Z_1 = [3, \infty[, Z_2 = [3, \infty[$. We have: $Y_1 \times Y_2 = (2, 2)$, Z_1 and Z_2 are two half lines on the axes, thus the condition 4.22 is not satisfied. However, $Y_1 \sqcup Z_1 = [2, \infty[$ and $Y_2 \sqcup Z_2 = [2, \infty[$, and $(Y_1 \times Y_2) \sqcup (Z_1 \times Z_2) = ([2, \infty[, [2, \infty[)$ which means we have 4.21.

When we compute $(Y_1 \times Y_2) \sqcup (Z_1 \times Z_2)$, its closure is necessary because if we choose a point a that belongs to $(Y_1 \times Y_2) \sqcup (Z_1 \times Z_2)$, the closure assures that the point a' on the segment Yb becomes the point a.

10.3 Practical Problems

As mentioned before, the Chernikova's algorithm is chosen in all polyhedral libraries for the convex hull operator, thus any improvement or comparison concerns the decomposition

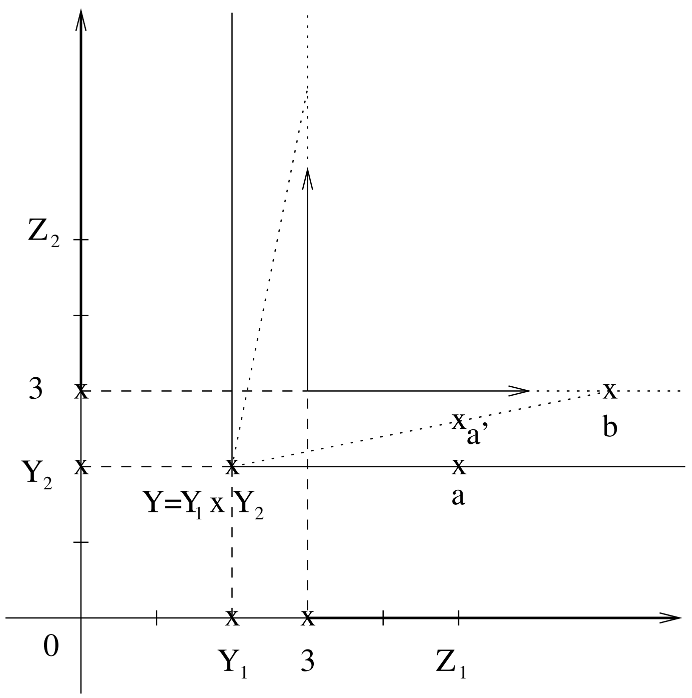


Figure 4.5: A counter-example

of polyhedra. In C^3, partial factorization is used instead of direct calls to Chernikova's algorithm, thanks to its improved performance. Experiments conducted in [HMPV03, Mer05] show encouraging results using Cartesian factorization, especially for polyhedra of large size.

There are not yet algorithms for approximations, so in case of exceptions, an universe polyhedron is returned, except in C^3, where a common part of operands is computed and returned as approximation.

Decomposition of polyhedra to speed up running time requires integration at low level, thus the work is considerable: Depending on the internal representation of polyhedra, the task can be more or less difficult ([Mer05], page 85 to 91). Moreover, the improvements are recent and need to be evaluated in order to be implemented in polyhedral libraries.

The last issue concerning the convex hull is that, the output (i.e. the returned polyhedron) is often larger in size, in term of numbers of constraints. Hence, more exceptions are raised, which penalize program analyses. Experimental evaluation is needed for re-use, for regression test, etc. In case of exceptions, better approximations can be made using the common part found by partial factorization, instead of simply returning the empty polyhedron.

11 Intersection

The *Intersection* of two (or more) polyhedra is in fact the intersection of all their hyper-planes (representing the equalities) and half-spaces (representing the inequalities), because a polyhedron is a finite intersection of hyper-planes and half-spaces. The intersection of two convex polyhedra is a convex polyhedron, given by the union of the constraints of its operands.

In constraint system form, the intersection operator is performed by concatenating the list of constraints. The result is a polyhedron, which can be minimized via the minimization operator. In practice, for efficiency reasons, the expensive minimization is not always applied, thus the redundant constraints appear in polyhedra. They are removed only when needed.

12 Difference

The *Difference* of two convex polyhedra A and B, denoted $A \setminus B$, is the set of elements of A that do not belong to B. It is equivalent to intersection of A with the *inverse*, or *complement* of B, denoted $\overline{B}$.

Since the inverse of B is an union of polyhedra, the difference of two polyhedra is an union of polyhedra. Hence, polyhedra are not closed under the operation difference. Not all available libraries implement the difference operator, because of this property.

To overcome this problem, two approaches are chosen: using polyhedra-derived domains, such as lists of polyhedra, Presburger's formulae, or octagons (see Chapter 3), or approximate the difference. An over-approximation is obtained by computing the convex hull of the resulting polyhedron, whose exactness can be checked as for the union.

13 Widening and Narrowing

The *Widening* and *Narrowing* are two operators; one enforces the termination of the abstract interpretation process by upward-approximating polyhedra, while the other improves the approximation by doing the reverse [CC77]. The first widening operator described in Nicolas HALBWACHS's PhD thesis [HAL79], on two polyhedra P and Q, is to keep all the constraints of P that are satisfied in Q. This operator requires as a precondition that $P \subseteq Q$. In general, the widening operator is used in the approximation of fixed point.

There are now two proposed widening operators for the domain of polyhedra. The first one mainly follows the Halbwachs's specification. This operator also requires as a precondition that $P \subseteq Q$.

The second widening operator is an instance of the specification provided in [BHRZ03]. This operator also requires as a precondition that $P \subseteq Q$ and it is guaranteed to provide a result which is at least as precise as the first one.

When approximating a fix point computation using widening operators, a common tactic to improve the precision of the final result is to delay the application of widening operators. The usual approach is to fix a parameter k and only apply widening starting from the k-th iteration. The implementation for this tactic is called widening with *tokens* operator (also described in [BHRZ03]).

A token is a sort of wild-card allowing for the replacement of the widening application by the exact upper bound computation: the token is used only when the widening would have resulted in an actual precision loss, as opposed to the potential precision loss of the classical delay strategy. Thus, all widening operators can be supplied with an optional argument, recording the number of available tokens, which is decremented when tokens are consumed. The approximated fix point computation will start with a fixed number k of tokens, which will be used if and only if needed. When there are no tokens left, the widening is always applied.

In PIPS, in order to obtain a modular analyzer and to limit execution times, state transformers are computed instead of state predicates (also known as pre- and post-conditions). As a result, PIPS does not use the well-known widening/narrowing operators but computes over-approximations of the transitive closures of loop transformers using finite differences and summation ([IRI05]).

14 Other Operators

As important as the projection operator but of lower complexity, the operator adding a new dimension (or several dimensions) to the vector space provides two distinctive options. The former *embeds* the polyhedron P into the new vector space and returns the polyhedron Q defined by all and only the constraints defining P; the variables corresponding to the added dimensions are unconstrained. The latter *projects* the polyhedron P into the new vector space and returns the polyhedron Q whose constraint system, besides the constraints defining P, will include additional constraints on the added dimensions. Namely, the corresponding variables are all specified to be equal to value zero.

In most existing libraries, the *space dimension* of a polyhedron P is simply the dimension $n \in N$ of the corresponding vector space D^n. Sometime, for generic reasons, the space dimension is considered as a set of dimension names, instead of a fixed number $n \in R$ (i.e. the C^3 library, [PIPb]). The space dimensions of constraint systems and generating systems are defined similarly. Thus, polyhedra are said to be *dimension compatible* if and only if they have the same space dimension. As such, operations requiring space dimension compatibility (e.g. convex hull or intersection of two polyhedra) sometimes need to *map* an operand's space dimension to the other's, or to *interchange* (*switch*) between two dimensions of a space dimension.

An operator that adds m new dimensions to a polyhedron $P \in R^n$ with $n > 0$, so that dimensions n, $n+1$, …, $n+m-1$ of the result Q are exact copies of the i-th space dimension of P, where $i \in [O, n]$, is call *expand* dimension operator. On the contrary, given a polyhedron $P \in R^n$ with $n > 0$, the operator *fold* folds a set of dimensions $J = j_0, \ldots, j_{m-1}$, with $m < n$ and $j < n$ for each $j \in J$, into the dimension $i < n$, where $i \notin J$. The expanding and folding operators are proposed by D. GOPAN and al. in [GDD$^+$04].

For each function mapping $\phi : R^n \mapsto \mathbb{R}^m$, we denote by $\phi(S) \subseteq \mathbb{R}^m$ the *image* under ϕ of the set $S \subseteq R^n$. Similarly, we denote by $\phi^{-1}(S') \subseteq R^n$ the *preimage* under ϕ of $S' \subseteq R^m$, that is the largest set $S \subseteq R^n$ such that $\phi(S) \subseteq S'$. For a given variable x_k and *linear expression* $\sum_{i=0}^{n-1} a_i x_i + b$, this variable and expression determine an affine transformation ϕ that is to be used by two operators that compute the affine image and affine preimage of a polyhedron P.

Sometimes we need operators that verify relations among polyhedra, for example to test if a polyhedron P is *disjoint* from or *included* in another polyhedron Q. Besides, if c is a constraint and Q is the set of points that satisfy c, we can test whether P is disjoint from c (i.e. $P \cap Q = \emptyset$, or adding c to P yields an empty polyhedron), P strictly intersects c (i.e. $P \cap Q \neq \emptyset$ and $P \cap Q \subset P$, or adding c to P yields a non-empty polyhedron strictly smaller than P) and P is included in c (i.e. $P \subseteq Q$ or adding c to P leaves P unchanged).

The test of equality between two polyhedra is not a trivial operator, because the polyhedral representations are not unique. A common way to verify the polyhedron P is equal to the polyhedron Q is to check if one is included in the other, in both directions.

An *interval* in R is a pair of bounds, called *lower* and *upper*. Each bound can be either closed and bounded, open and bounded, or open and unbounded. If the bound is bounded, then it has a value in R. An n-dimensional *box* B in D^n is a sequence of n intervals in R. The *bounding box* operator for a polyhedron P returns the smallest n-dimensional box containing P.

Non-regression testing and memory space analyzing require more operators such as debugging functions, etc. but we choose not to discuss them in this chapter. In the next section, we study into detail each and every operator, where the first one is dual conversion operator because it can be used by other operators.

Along with the presented operators, we have many other operators that are not really interesting to go into details, such as printing functions, dimension swapping functions, etc.

As we have known, the idea of an abstract domain like the polyhedral one is to model instructions, objects, effects, etc. in program analyses. Important operators such as assignment, guard, etc. can be built from elementary operators described above, at least for the polyhedral domain, so that they do not appear in our list of operators.

Nevertheless, in another abstract domain, i.e the octagons, these operators are introduced because it cannot be built from other operators. We have verified by building the assignment operator, based on other operators of the octagon library, and we found out that the swapping function is missing. In fact, this is an example of incompatibilities that might be avoidable (i.e. not an implementation specific problem), noting that abstract domains often have common purposes.

15 Conclusion

In this chapter, we have presented an operator by operator view of the common polyhedral API, surveyed algorithms and existing implementations, and discussed problems concerning each operator. Problem of different operator names is clarified.

In Section 6, we have presented the available algorithms for the dual conversion operator, an elementary operation that can be used in other operators such as projection, satisfiability, convex hull, etc.

Then, in Section 7, we have detailed four algorithms that are implemented for the test of satisfiability: the Fourier-Motzkin's, the Simplex for rational test, that were already available in C^3 library, the JANUS for integer test that was ported to 64-bit by me, and the algorithm based on dual conversion operator, implemented by me.

In Section 8, we have introduced two main algorithms for the projection operator, one uses the dual conversion, implemented by me, and the other uses the Fourier-Motzkin elimination method.

Similar to the projection operator, the minimization operator, which has two main algorithms, is presented in Section 9. One is based on dual conversion, implemented by me, while the other directly manipulates the polyhedron's H-representation form.

The convex hull operator is exposed along with three recent improvements: the partial factorization (Section 10.2.1), the Cartesian factorization (Section 10.2.2) and our decomposition using inclusion test, inspired by the Cartesian factorization (Section 10.2.3).

Finally, other operators such as the intersection, difference, widening, narrowing, etc. were briefly described.

For each operator, differences among existing libraries at the interface level were analyzed, in order to study the possibility of an integration among nearly-equivalent works. Being mostly an experimental work, practical issues such as incompatibilities among polyhedral libraries are relevant.

Details on the most important algorithms are given in this chapter. Some propositions for improvement (e.g. the decomposition using inclusion test, Section 10.2.3) as well as properties (e.g. projection using double description Page 102) are exposed and discussed. Backup algorithms for approximations to deal with polyhedral high complexity are also discussed.

The question of precision versus approximation is raised throughout this chapter, as well as computational issues like 32-bit, 64-bit, 128-bit or GNU multi-precision modes.

Our contribution in order to improve some of these operators is also presented in this chapter (Section 7.2.3, Page 98, Section 10.2.3, Page 113).

In Chapter 5, we present our framework to analyze the performances of the mentioned implementations. Then we introduce our experimental results. They are strongly related to the content of this chapter.

Chapter 5

Benchmarking existing libraries

In Chapter 4, we present the polyhedral interfaces with some differences in names, a survey of algorithms and existing implementations, as well as problems concerning each polyhedral operator. Details on the most important algorithms and some propositions for improvement are given. The question of precision versus approximation is raised, as well as computational issues like $32-bit$, $64-bit$, $128-bit$ or GNU multi-precision arithmetic. Practical issues, such as incompatibilities among polyhedral libraries, are listed.

In this chapter, we motivate our benchmarking effort with several examples to showing the impact of exceptions and large operands on abstract interpretation result (Section 1.1). Then we introduce our POLYBENCH framework, which we designed to analyze automatically the run-times and exceptions of the many available libraries presented in Chapter 3 with respect to thousands of polyhedral operations traced from PIPS [PIPb, IJT91] static analyzer execution (Section 1.2). We used our POLYBENCH tools to obtain experimental results about the cases used and about five different key polyhedral operators: the integer and rational satisfiability (Section 3), dual conversion (Section 6), projection(Section 4), minimization(Section 5) and convex hull(Section 7).

1 Benchmarking

Before introducing our framework, we explain why it is important to have an automatic benchmarking mechanism. Our examples originate from static analyses of standard benchmarks, which are presented in Section 1.2.

1.1 Motivation: Impact of exceptions on accuracy

The polyhedral interface consists of several operators namely satisfiability test, inclusion test, projection, minimization, dual conversion, convex hull, intersection, difference and widening/narrowing. To perform each operator, many algorithms have been implemented. They can be used on integer or on rational domains to prove properties of programs. These algorithms often have a polynomial complexity but the worst-case exponential complexity hits sometimes. Unfortunately, this worse-case exponential complexity can block an analysis, especially when memory space and time are limited.

Adding information about a program should always result in more accurate analyses. It is however not true when 32-bit or 64-bit integers are used. When information is added, more overflows may occur in the linear algebra algorithms, then approximations must be made, resulting in longer and less accurate analyses. We take an example of a simple FORTRAN code, the polynomial in Figure 5.1, and use PIPS to analyze this example.

```
I = 1, J = 0, K = 0
DO WHILE(K.LT.100)
   K = K + J
   J = J + I
   I = I + 1
ENDDO
PRINT *, I, J, K
IF(X.GT.0.) THEN
   READ *, J, K
   PRINT *, I, J, K
ELSEIF(Y.GT.0.) THEN
   READ *, I, K
   PRINT *, I, J, K
ELSE
   READ *, I, J
   PRINT *, I, J, K
ENDIF
```

Figure 5.1: Example: Polynomial code

```
      IF (X.GT.0.) THEN
         READ *, J, K
         PRINT *, I, J, K
      ELSE
         IF (Y.GT.0.) THEN
            READ *, I, K
C  first iteration:
C  P(I,J,K) {2<=J}
C  second iteration:
C  P(I,J,K) {3<=J}
C  third iteration:
C  P(I,J,K) {6<=J, J<=300}
C  fourth iteration:
C  P(I,J,K) {}

            PRINT *, I, J, K
         ELSE
            READ *, I, J
            PRINT *, I, J, K
         ENDIF
      ENDIF
```

Figure 5.2: Evolution of preconditions for Polynomial code over four iterations

The preconditions and transformers are computed to obtain information about each variable I, J, K independently for the tests on uninitialized variables. A technique used in PIPS in order to obtain more information is to compute several times the preconditions and transformers using their previously computed results. It is illustrated by the evolutions of transformers and preconditions in Figure 5.2 and Figure 5.3 where the results of four iterations are displayed for one of the statement in Figure 5.1 (see Chapter 2, Section 4 for PIPS's transformers and preconditions analyses).

In Figure 5.3, we can see that the first transformer only contains information about the variable I and its initial value: $I\#init \leq I$. The second transformer computed with the first precondition gives information not only about the variable I, but also the variable J: $I\#init \leq I, I + J\#init \leq I\#init + J$. In the same way, the transformer computed the fourth time seems richer of information than any previous one. However, if we look at the Figure 5.2, everything seems to be fine until the fourth loop: the information at the

```
      I = 1,  J = 0, K = 0
C  first iteration:
C  T(I,J,K) {I#init<=I}
C  second iteration:
C  T(I,J,K) {I#init<=I, I+J#init<=I#init+J}
C  third iteration:
C  T(I,J,K) {I#init<=I, I+J#init<=I#init+J,
C    6I#init+3J+K#init<=6I+3J#init+K, I#init+J+K#init<=I+J#init+K}
C  fourth iteration:
C  T(I,J,K)
C    {1379460I#init+895055J+454903K#init<=1379460I+895055J#init+454903
C    K, 1063137I#init+639920J+364213K#init<=1063137I+639920J#init+
C    364213K, 6748I#init+1469J#init+479K<=6748I+1469J+479K#init,
C    1802I#init+899J+299K#init<=1802I+899J#init+299K,
C    287I#init+10J#init+7K<=287I+10J+7K#init,
C    41I#init+5J#init+2K<=41I+5J+2K#init,
C    10I#init+4J+K#init<=10I+4J#init+K,
C    2622I+2622J#init+263K<=2622I#init+2622J+263K#init,
C    28497I+28497J#init+109K#init<=28497I#init+28497J+109K,
C    30061I+30061J#init+673K<=30061I#init+30061J+673K#init}

      DO WHILE (K.LT.100)
         K = K+J
         J = J+I
         I = I+1
      ENDDO
```

Figure 5.3: Example: Evolution of transformers for Polynomial code

third loop, $6 \leq J, J \leq 300$, is lost at the fourth loop. It is because an overflow/magnitude exception has occurred.

Let us take another example. With a hardware configuration PC 2.4GHz, 2GB RAM, we analyze the program *ocean.f* with size of 4373 LOC from the PerfectClub benchmark. We have 11916 calls to satisfiability test and 424 overflows; 3521 calls last longer than three seconds and the largest constraint system contains 906530 constraints. In fact, the computation on polyhedra can be very expensive: the larger the size of analyses becomes, the more problems appear. Our motivation is to quantify these problems.

Firstly, the operators manipulating the polyhedra at the lowest level should be the most efficient as possible. Note that in a typical program analysis section, an analyzer can call up to a hundred thousand elementary operations, or even more. Experiment shows that the impact is significant. It is however difficult to tell whether the implementations of these operators are efficient or not.

Secondly, heuristic-based approaches are thus needed to avoid infinite precision arithmetic and to maintain an execution speed fast enough to process large real applications reaching 100, 000 lines of code, with hundreds of variables linked by hundreds of constraints. The problem of time and memory space must then be reduced as much as possible. Moreover, constraint coefficient magnitude is also a complexity issue. Suboptimal or heuristic solutions must be found when arithmetic overflows occur, in order to limit or control the information loss. Heuristics can only be found and validated through experimentation.

Finally, the dynamic behavior of programs is often controlled by integers, boolean variables and character strings. They can all be mapped on integers whereas linear algebra is mostly based on real and rational numbers, and linear algebra does not provide exact set operators very often. Different implementations of set operators by linear algebra algorithms can be exact, over- or under-approximation. Thus, accuracy and speed comparisons should be made.

1.2 Large Operands

In this section, we introduce three examples that show the high complexity of static program analyses using the polyhedral domain. These examples are extracted during PIPS execution on two benchmarks, PerfectClub and SPEC95.

1.2.1 Normalization

From PIPS execution on SPEC95's applu.f program, with computation of transformers, preconditions and regions, we observed constraint systems of very large sizes. For example, the printed out constraint system number 23499 which was passed to C^3 normalization operator, takes 52, 4 MB of disk space, with only 7 dimensions, no equation but a whooping 906530 inequalities. Figure 5.4 is the first fragment of the system.

We remark that the coefficients of this constraint system are quite large in the million to the billion range. Linear combinations such as Fourier-Motzkin elimination are likely to raise integer overflows with a 64-bit representation. When this system is passed to other operators, it raises exceptions, thus block our analyses. The origin of the system is identified as follows: from constraint systems of some hundred of constraints, operations on those systems such as projection, normalization, convex hull, etc sometimes yield up to constraint systems of thousand of constraints, mostly inequalities.

Moreover, when the satisfiability test cannot handle those constraint systems, we cannot remove redundant constraints, thus those systems remain very large and increase from operation to operation and we cannot obtain useful information.

1.2.2 Projection

The constraint system, whose first fragment is printed in Figure 5.5, represents a case that our projection operator cannot handle. It raises an exception in memory space by the

```
#DIMENSION (7)  INEGALITES (906530)  EGALITES (0)
VAR PHI2, PHI4, PHI3, PHI1, NX, NZ, NY
  {
- 311573790 NY - 213825150 NZ - 207715860 NX  <= -1570087530 ,
- 6479550 NZ - 12583230 NY - 6294420 NX  <= -50720010 ,
- 207900990 NY - 213825150 NZ - 207715860 NX  <= -1259069130 ,
- 207900990 NY - 220304700 NZ - 207715860 NX  <= -1265548680 ,
- 12779580 NY - 6479550 NZ - 6294420 NX  <= -50916360 ,
- 6300030 NY - 6479550 NZ - 6294420 NX  <= -37957260 ,
...
 }
```

Figure 5.4: Example: first fragment of a large constraint system which fails the normalization operator

Fourier-Motzkin method, thus the execution of PIPS is not possible without approximation on the program applu.f in SPEC95 benchmark.

We have analyzed this example by applying successively the projections along the variable NY, NZ, NX, PHI1, PHI3, PHI4. The projection along NY or NZ or NX raises immediately an out-of-memory space exception, whereas the projection along PHI2 or PHI3 or PHI4 reduces greatly the size of the constraint system, so that projections along the other variables become possible. Beside the order of the list of variables to project, the Fourier-Motzkin elimination method used in the projection is also sensitive to the order of the constraints in the constraint system. In this algorithm, the integer results can be retained if we only project the variable along unitary coefficients (1 or -1) [1]. However, we did not have time to build a better projection algorithm.

1.2.3 Convex Hull

As mentioned above, the program *ocean.f* in the benchmark PerfectClub is a special case, which PIPS has difficulty to deal with. The first phenomenon is observed by the very expensive computation of convex hull operator in PIPS for the two constraint systems in Figure 5.6.

The number of generating vertices grows exponentially with the number of constraints just as for an hypercube. The implementation of Chernikova algorithm available in POLYLIB and used in C^3 cannot handle the dual conversion of the first constraint system, thus an exception “out of table space” is detected. Even after the partial factorization by Corinne ANCOURT and Fabien COELHO (see Chapter 4, Section 10.2.1) the polyhedra remains too large.

[1] Implemented by Fabien COELHO, though we do not have any documentation.

```
#DIMENSION (7)  INEGALITES (2372)  EGALITES (0)
VAR PHI2, PHI4, PHI3, PHI1, NX, NZ, NY
  {
 - 183823200 NY - 6066165600 NZ - 183823200 NX  <= -13414772448 ,
- 6066165600 NY - 6066165600 NX - 6066165600 NZ  <= -48708826848 ,
- 5924160 NY - 195497280 NZ - 5924160 NX  <= -432324288 ,
- 195497280 NY - 195497280 NX - 195497280 NZ  <= -1569763008 ,
- 195497280 NZ - 5924160 NX - 11499840 NY  <= -449051328 ,
- 201072960 NY - 195497280 NX - 195497280 NZ  <= -1586490048 ,
- 195323040 NZ - 5749920 NX - 5749920 NY  <= -430756128 ,
...
  }
```

Figure 5.5: Example: first fragment of a large constraint system which fails the projection operator

1.3 Related Work

Much research has been concentrated on possible improvements to these algorithms, based on a mathematical background (see Chapter 3). A list of implementations concerning these operators has been made, including some "complete" polyhedral libraries like the *POLYLIB* [LOE, Wil], the *New POLKA* [JEAb, JEA00], the LINEAR C^3 [PIPa, CAI00], or just libraries that focus on some particular operators, like *CDD* [FUK], *LRS* [Avi] or JANUS [SOG, SOG96]. For example, JANUS is a piece of software that deals with the integer satisfiability problem, whereas the calculation of convex hull of two or more polyhedra is maybe the most discussed polyhedral operator in the literature [BAY].

However, our bibliographic study shows that it doesn't exist yet a mechanism to evaluate effectively existing polyhedral libraries, especially in the domain of program analysis and transformation for real-life examples. Published evaluations, e.g. [SOG96] and [Ba], are based on at most two hundreds instances, mostly theoretical, without analyses on polyhedral characteristics, e.g. dimension space, numbers of constraints, etc., on numbers of exceptions, cases where algorithms fail because of resource limits, etc. Apart from clarifying whether CPU or memory efficiency or both are the intended measures of interest, we offer problem-related analyses, such as polyhedral criteria and their origin, stability comparisons, i.e. the ability of coping with difficult cases, incoherent results checking, computing precision comparisons, etc. [2] Our comparisons cover not only the satisfiability test and dual conversion operators but also other important polyhedral operators, such as projection, minimization and convex hull operators.

[2] Non-regression testing can be improved by making comparisons possible between implementations of the same algorithm, since bugs are sometimes hard to detect in complex algorithms.

```
#DIMENSION: (69)  INEGALITES(72)  EGALITES (13)
VAR N2M#new, LPN#new, LZN#new, N2P#new, N1H#new, N2#new, MEMSIZ#new,
NWH#new, NWEIG#new, LEIG#new, NW#new, NWQ#new, LZX#new, LPY#new,
LZY#new, LZO#new, NUMBER#new, NXXXIN#new, NXXXIN#init, NPTS#new,
NPTS#init, NSKIP#new, NSKIP#init, MTRN#new, MTRN#init, MSKIP#new,
MSKIP#init, ISIGN#new, ISIGN#init, NXLOG2#new, NXLOG2#init,
NFTVMT#new, NFTVMT#init, NXACAC#new, NXACAC#init, NXXOUT#new,
NXXOUT#init, KZN#new, KZN#init, KZO#new, KZO#init, K#new, K#init,
NUSHUF#new, NUSHUF#init, NXXCSR#new, NXXCSR#init, NXSCSC#new,
NXSCSC#init, NXPRNT#new, NXPRNT#init, ZETAPH:NCALL#new,
ZETAPH:NCALL#init, KPN#new, KPN#init, NXXRCS#new, NXXRCS#init,
TEMPHY:NCALL#new, TEMPHY:NCALL#init, LVECT#new, LVECT#init, LSKIP#new,
LSKIP#init, NVECT#new, NVECT#init, NVSKIP#new, NVSKIP#init,
NSTEPS#new, NSTEPS#init
  {
- NSTEPS#new - TEMPHY:NCALL#init + NSTEPS#init + TEMPHY:NCALL#new  <= 0 ,
- NSTEPS#new + NSTEPS#init  <= -1 ,
...
  }

#DIMENSION: (69)  INEGALITES (0)  EGALITES (26)
VAR N2M#new, LPN#new, LZN#new, N2P#new, N1H#new, N2#new, MEMSIZ#new,
NWH#new, NWEIG#new, LEIG#new, NW#new, NWQ#new, LZX#new, LPY#new,
LZY#new, LZO#new, NUMBER#new, NXXXIN#new, NXXXIN#init, NPTS#new,
NPTS#init, NSKIP#new, NSKIP#init, MTRN#new, MTRN#init, MSKIP#new,
MSKIP#init, ISIGN#new, ISIGN#init, NXLOG2#new, NXLOG2#init,
NFTVMT#new, NFTVMT#init, NXACAC#new, NXACAC#init, NXXOUT#new,
NXXOUT#init, KZN#new, KZN#init, KZO#new, KZO#init, K#new, K#init,
NUSHUF#new, NUSHUF#init, NXXCSR#new, NXXCSR#init, NXSCSC#new,
NXSCSC#init, NXPRNT#new, NXPRNT#init, ZETAPH:NCALL#new,
ZETAPH:NCALL#init, KPN#new, KPN#init, NXXRCS#new, NXXRCS#init,
TEMPHY:NCALL#new, TEMPHY:NCALL#init, LVECT#new, LVECT#init, LSKIP#new,
LSKIP#init, NVECT#new, NVECT#init, NVSKIP#new, NVSKIP#init,
NSTEPS#new, NSTEPS#init
  {
- NXXXIN#new + NXXXIN#init  == 0 ,
- NPTS#new + NPTS#init  == 0 ,
...
  }
```

Figure 5.6: Example: first fragments of two large constraint systems which fail the convex hull operator

An example of previous comparisons can be found in [SOG96], where JANUS is compared with the Omega test [PUG91], whose results discover performance-related issues concerning only one problem, the *nightmare* problem, due to overhead factors in the Omega tool.

There is also a set of tests provided by *CDD* [FUK] and *LRS* [Avi], then used by PPL developers in order to evaluate the PPL library's performance [Ba], compared to other libraries. These evaluations are based on the vertex/facet enumeration problem with a set of less than two hundred inputs, which are supported by the libraries *POLYLIB* [LOE, Wil], *New POLKA* [JEAb, JEA00], the LINEAR C^3 [PIPa, CAI00], *CDD* [FUK] and *LRS* [Avi]. Though these inputs supply varied complexity operations for these programs, they are far from satisfying, given the differences among the applications. It is known that even with the same application, big variations may be observed for different inputs. Thus we can only measure the performance of the application with the biggest variety of inputs we can come up with. The results then can be used to choose the library that gives us the best overall performance.

1.4 Building a Polyhedral Benchmarking System

All these applications motivate the construction of a large polyhedral benchmark which enables us to analyze implementations experimentally. Our benchmark, named POLYBENCH, provides an easy and straightforward way to compare methods, with a great quantity of operations encountered in program analysis and transformation.

Early experiments were carried out, focusing on the satisfiability test with only three implementations: JANUS [SOG, SOG96], Simplex and Fourier-Motzkin [PIPa, CAI00]. They revealed important differences among these algorithms. Then, we extended our framework and added the projection, minimization, dual conversion and convex hull operators. The following section describes this benchmarking environment.

2 Constitution of a Polyhedral Benchmark - POLYBENCH

It is acknowledged that robustness and execution time are two important criteria for elementary operations. But precisely, how do we define these two criteria?

2.1 Benchmarking Conventions

An operator returns for a given operation an answer or an exception. When we compare two implementations, we say that an implementation is more *robust* than the other if it generates fewer exceptions. We also say that one is more *efficient* than the other if it returns a valid answer sooner than the other does. Finally, we consider that one implementation is more efficient than another if it raises an exception earlier, on condition that both implementations return an exception with the given operands (see Section 2.5 for our implementation).

The robustness refers to the ability of dealing with exceptions, which can be quantified. Exceptions can be either magnitude overflows, i.e. when a number exceeding the maximum that can be stored in a memory word, or timeouts, i.e. no operation may last too long. The timeout exception means that the operator implementation cannot solve the given task in a pre-defined time.

In fact, memory managements implemented in available libraries are different: we have virtual memory space for some algorithms and limited for others. Due to this difference, we consider memory overflows, e.g. out-of-memory space, as magnitude overflows when it happens before the timeout exceptions.

2.2 POLYBENCH Overview

In order to provide a tool comparing similar methods, we build up a polyhedral benchmark, made of a large number of polyhedral operations that we extract from the execution of real-life program analyses and transformations.

In this section, we present our polyhedral benchmark framework, called *POLYBENCH*, by commenting Figure 5.7 step by step. PIPS [PIPb, IJT91], which stands for "Paralléliseur Interprocédural de Programmes Scientifiques", is an analyzer-transformer of programs that uses polyhedral abstraction [PIPb, IJT91]. A PIPS phase can perform analyses and transformations on Fortran programs, such as *Array Privatization, Dependence Testing, Memory Effects, Preconditions, Expression Optimizations, Forward Substitution, Parallelization*, etc...

Standard benchmarks like the PerfectClub [BCK$^+$89] and SPEC CFP 95 [SPE], henceforth called SPEC95 for short, have been chosen for polyhedral data generation as PIPS's input (1). For our benchmark, we have chosen the transformers and preconditions since they are often used in other analyses, and array regions due to its high computational demand (2). During PIPS execution on these benchmarks, polyhedra in form of constraint systems are traced and stored in a large database of directory-structured files, divided by polyhedral operators [3] (3). This database is compressed because it is quite large: over 20 GB. Then we apply some sampling for each sub-database in order to reduce the number of constraint systems to be tested when needed (4). Since we want to measure implementations' performance in general cases, these sampled databases should be representative. However, we are sometimes interested in a biased database, for special purposes, therefore we also use some criterion-based filters, for instance to retrieve only the larger polyhedra (4) (see Section 2.8).

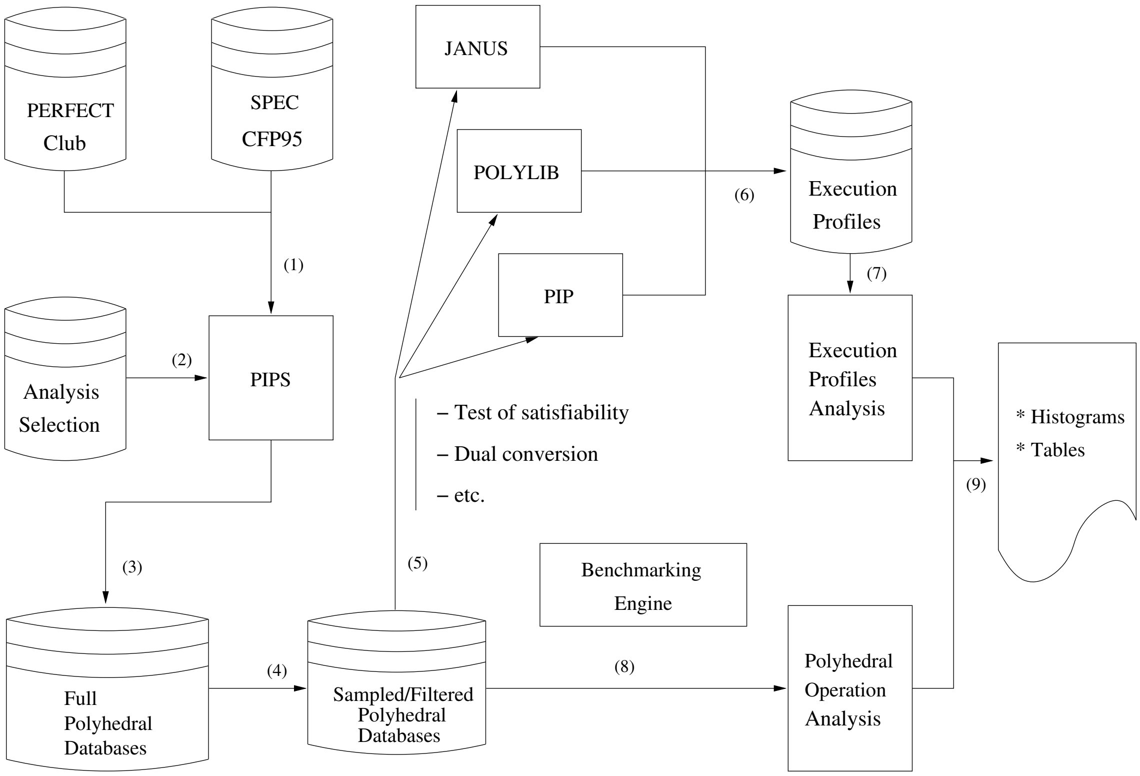


Figure 5.7: POLYBENCH's schema

The databases contain operands for most available implementations of polyhedral operators, such as LINEAR C^3 [PIPa, CAI00], *CDD* [FUK], *JANUS* [SOG], *LRS* [Avi], *POLYLIB* [LOE, Wil], PPL [PAR, BRZaH02] and *New POLKA* [JEAb, JEA00]. The implementations having different data structures are encapsulated by converters, which we implemented for each library. We measure the performances of similar implementations $(5, 6, 7)$, then display the comparisons in graphical form (9). We also analyze some characteristics of those databases (8).

2.3 Execution Time Measurements

In order to measure the run time of varying implementations without too much code modification, we use the *time* utility, available in $UNIX$-based operating systems. This

[3] Each operator has its own databases/directories.

brings portability to the framework, but has a consequence on the time resolution which can be not very fine and depends on the hardware. It's common for the internal processor clock to have a resolution somewhere between hundredths and millionths of a second. In our case, we have millionth of a second.

When the run time of one operator is small, e.g. the satisfiability test, we repeat the operation many times, e.g. n times, to increase the accuracy of time measurement. This also reduces uninteresting overheads such as input/output run time, since these factors are divided by n, which is chosen as 100. However, given that run times of complex operators such as convex hull are too long, we do not repeat these operations, i.e. n is 1.

2.4 Size Parameters

The execution times of these implementations are displayed with respect to polyhedral size criteria, so that we can observe the relation between the performances and specific sets of polyhedra. In fact, the chosen criteria for filters and statistical characteristics of the databases are numbers related to dimensions, number of constraints (equations and inequalities) and coefficients:

- number of dimensions, i.e. number of variables;
- number of constraints, i.e. number of equations and inequalities;
- the largest coefficient, i.e. magnitude;
- density or sparsity, which is the polyhedron's representative matrix's density (see [YAN93]) [4].

The above *size* parameters of constraint systems are determining factors for every polyhedral operation. For practical reasons we have chosen the numbers of dimensions, of constraints and density information. Other factors are expensive to compute, e.g. matrix's determination, and heuristics, which could take advantage of this information to select the most appropriate algorithm, should not require CPU intensive decision procedures.

2.5 Implementation

The framework is fully automatic, in order to rebuild new results when necessary. However, such procedures take quite a while. POLYBENCH consists of several shell scripts and C programs which take advantage of C^3 available parsing mechanism. We use GNU make to generate POLYBENCH executable from the source.

[4] In fact, we use the sparsity index: $\sum_{i=1,j=1}^{n,m} a_{ij}/n$, which is the sum of all coefficients a_{ij} divided by the number of constraints n, and m is the number of variables. The computation is cheap and it contains information about the density of non zero coefficients in the matrix representing the polyhedron.

For polyhedral inputs, we use the C^3 sparse format, i.e. ASCII, for every implementation, thus an internal format conversion is needed for every new library. Examples of this format can be found in Section 1.2.

For each new library, we need to:

- write a wrapper including the internal format conversion and compile it;
- write a simple pb_2time batch and launch it with the polybench_create_2time_criteria_data.sh script;
- write a simple pb_hg batch and launch it with the polybench_create_histograms.sh script.

Outputs are graphs, created by gnuplot with the histogram and datastring enabled patches, and tables in ASCII, that illustrate differences in execution times and in numbers of exceptions. Examples of wrappers and batches are available.

Head-to-head comparisons: We use head-to-head comparisons between two implementations. These comparisons are about execution times and numbers of exceptions. In cases where one of the implementations generates an exception (timeout or overflow), we do not compare their execution times, and exclude them from the total execution times. Thus, the total execution times mentioned later in this chapter can be understood as execution times for cases solvable by both implementations only. We also exclude the input/output execution time in our tests.

POLYBENCH's implementation for the automation of comparisons consists of several UNIX shell scripts and C programs. It has been tested on platforms Debian Linux and Sun Solaris. The database generation part is strongly related to PIPS and C^3 infrastructure. Nonetheless, once the polyhedral databases are created, the POLYBENCH can be used with only a fragment of C^3 code.

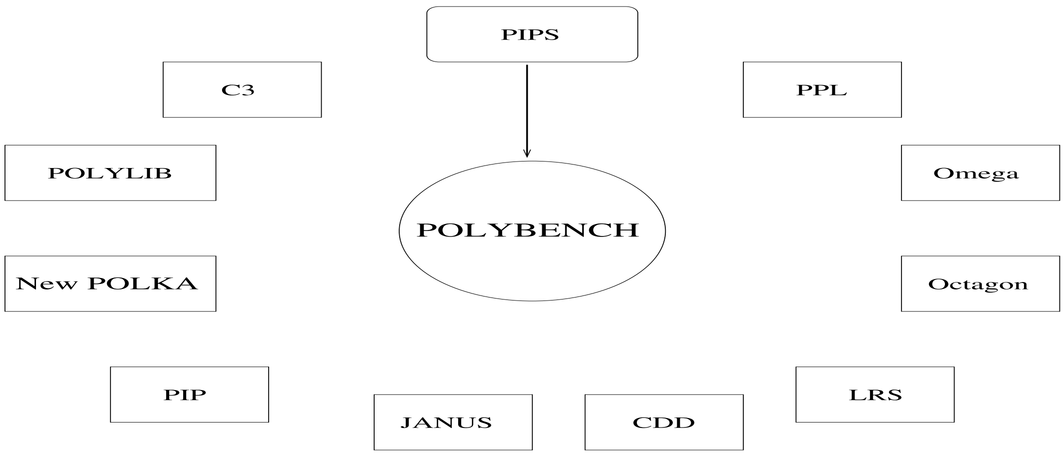


Figure 5.8: Libraries tested with POLYBENCH

Figure 5.8 represents some possible applications of POLYBENCH. Polyhedral libraries can use our POLYBENCH's set of tests so that their performance can be evaluated.

2.6 Target Machines

Experiments have been carried out on a 2.4 GHz Pentium IV machine with 2 GB of RAM, running Debian Operating System version 3.0, and on a 270 MHz UltraSparc IIi with 128 MB of RAM, running Sun Operating System Edition 5.8 Generic_108528-10. The gcc version used under Debian is 3.3.5, and under Solaris is 2.8.1.

As our experimentation have been carried out on two platforms, GNU Linux and Sun Solaris 8, we have chosen the compilation options of gcc as following; Under GNU Linux, we use *-g -Wall -W -pipe -march=pentium -malign-double*; under Solaris, we use *-g -Wall -W -pipe -msupersparc*.

2.7 Presentation of Results

The histograms presented in this chapter use the following conventions:

- For one operation, i.e. one input with one or two constraint systems, an implementation is considered *better* than another one if its measured execution time is strictly shorter;
- Two implementations are considered *equivalent* if their execution times are not distinguishable. This is relative because it depends on time resolution;
- The caption may contain the speed-up or slowdown between two implementations, e.g. JANUS and C^3 Simplex;
- The vertical axis represents the number of operations/tests executed in the labeled database, and the horizontal axis corresponds to the criterion considered, for instance the operand dimension.

Some of implementations use integer computations such as JANUS and C^3, the others are using rational computation. Possible result differences are observed and analyzed in our results.

An interesting question is related to the availability of several implementations for one same task. What would we get if we run them all in parallel? Would we get the best performance? This question is also answered by our benchmark.

Parallel Algorithm: If we simultaneously execute two algorithms and only take into account their best run time, we will have a performance of a new algorithm that we call the *parallel algorithm*, or just PA for short. Here, we can imagine that we have two independent machines or processors with multiple cores or threads, each runs an algorithm, and we only take the better result.

Moreover, if we consider that this parallel algorithm raises an exception only when both algorithms fail, then its number of exceptions should also always be lower.

We consider the parallel algorithm for two algorithms A and B on a set of three tests as an example to illustrate this idea: the first test, both A and B fail; A fails the second test and B fails the third. So the parallel algorithm only counts one exception, with the first test, whereas A and B each counts two exceptions. Obviously, we can also define a parallel algorithm for more than two algorithms.

2.8 Polyhedral Databases

Database description: As mentioned above in Section 2, our polyhedral databases are generated by tracing PIPS execution on public benchmarks, and stored in directory-structured files. There is a possibility that we have a small redundancy in our polyhedral databases. The chosen analyses are transformers, preconditions and must regions (see Chapter 2, Section 4).

Two Types of databases: We have decided to use two different databases to study the effect of different databases: one is biased, i.e. towards large size polyhedra, and the other is periodically sampled since it contains many polyhedra. We have considered two types of POLYBENCH databases, for each operator. For instance, with the satisfiability test in Table 5.1:

- *Sampled* Database: constraint systems are periodically selected with probability at intervals of 100, i.e., for every hundred constraint systems, we take only one, which means we use only 1% to speed-up experiments and obtain average results. We have chosen 1% for the sampling adapting to our experimentation: 1% is still quite large for satisfiability databases that relatively assures accurate measurements. And even with only 1%, the execution of the satisfiability test databases still last a week: while the input/output, database compression/decompression, computations of polyhedral characteristics, e.g. number of vertices, for each input using the Chernikova algorithm, output comparisons and timeout exception cases, which are not counted in our displayed total execution time as explained in Section 2.5, already require important execution times, our implementation of head-to-head comparisons sometimes requires several executions of the same implementation on the same databases, which adds up to the total execution times. For example, JANUS versus Simplex and JANUS versus Fourier-Motzkin need two executions of JANUS on our satisfiability databases[5]. Since other databases are smaller, we use no sampling when possible;

- *Filtered* Database: only *large size* constraint systems, large according to some size criteria, are selected to analyze better the impact of exceptions. Thus this database

[5]This can be improved by implementation of a memorization scheme which belongs to our future work

	PerfectClub	SPEC95
Sampled Database (1%)	12668	16303
Filtered Database	1310	4676

Table 5.1: Satisfiability Test: Numbers of constraint systems

is biased on purpose. We have used the following criterion: $((dimensions > 80) \vee (nb\_constraints > 50) \vee (density > 200) \vee (magnitude > 800))$. It presents the case where the "size" of constraint systems is relatively large, which might lead to large run times according to our experimentation on some extracted examples. We notice here that, for some cases, these databases are very small or even empty, thus they are only used for statistical purposes.

Incoherent Databases: In fact, the generation of POLYBENCH databases is not trivial, since it was built during PIPS development. At the beginning, we extracted a database from the original PIPS using the PerfectClub and SPEC95 benchmarks. At this time, the implementation of operators such as convex hull, normalization and projection was not armed with timeout mechanism, thus in case of trouble, PIPS executions lasted very long or did not even terminate correctly. For example, PIPS execution on PerfectClub's ocean.f was blocked several days on account of expensive convex hull computation, then its database was generated with many polyhedra of large size. Later, the timeout mechanism that helped reduce PIPS execution times for *hard* cases was implemented, and the newly generated database contained many more polyhedra but of smaller size.

Moreover, the POLYBENCH framework at this time was experimental, thus only the satisfiability operator was tested, and its databases was also used for other operators such as projection, minimization. Due to changes made during PIPS development, we cannot reproduce the original databases. To save time, we decided to use two different databases, which are the initial filtered and the new sampled ones.

Chosen Criteria: We found that 1% sampling and the chosen filter criterion were reasonable for the total execution times of all the necessary operations. Moreover, during our early experiments, we have tested some other databases with different intervals and criteria to compare their results, and found out that our choice were good enough to obtain reliable results, with only one exception described in Section 7. For instance, we have basically the same conclusions with 10% and 1% sampling for satisfiability test. The 1% sampling and full projection databases also give similar results.

Note here that even with the 1% sampling rate, we still have a very large number of constraint systems that relatively assures accurate measurement. Table 5.2, Table 5.3 and Table 5.4 present the two types of databases for the projection, minimization and convex hull operators. We remark that their sampled databases are smaller than satisfiability

	PerfectClub	SPEC95
Sampled Database (1%)	1789	2583
Filtered Database	3	5

Table 5.2: Projection: Numbers of constraint systems - filtered databases useless

	PerfectClub	SPEC95
Sampled Database (1%)	3894	4608
Filtered Database	11	12

Table 5.3: Minimization: Numbers of constraint systems - filtered databases useless

test's, and their filtered databases are useless.

2.9 Distribution of Dimension Space

Analyzing the dimension-based histograms generated from all the databases, we remark a relation between the database generated from the analyzed program, and the program itself. Let us take the satisfiability test for example.

The program *Ocean.f* in the PerfectClub database: Observing the distribution based on the numbers of variables of constraint systems in Figure 5.9 we can see a peak corresponding to dimension of 105, that shows the effect of ocean.f. Figure 5.10, Figure 5.11 and Figure 5.12 show distribution of the other databases in PerfectClub and SPEC95, without explanation of the peaks. The phenomenon is due to the complex structure of the ocean's main module with 807 lines of code, where a great number of constraint systems are to be manipulated, and average sizes of these constraint systems are much larger than the others because of the large number of global variables visible and modified at the main level.

In our experimental sections, in order to reduce the number of graphs and tables in this dissertation, we only present the results of PerfectClub or SPEC95, which are direct comparisons of implementations and characteristics of databases, and do not consider all the criteria. Any difference between results of the two benchmarks is revealed if it does exist. The full set of experimental results can be found at POLYBENCH's website [QUEb].

	PerfectClub	SPEC95
Sampled Database (1%)	194 pairs	240 pairs
Filtered Database	0	0

Table 5.4: Convex Hull: Numbers of constraint systems - empty filtered databases

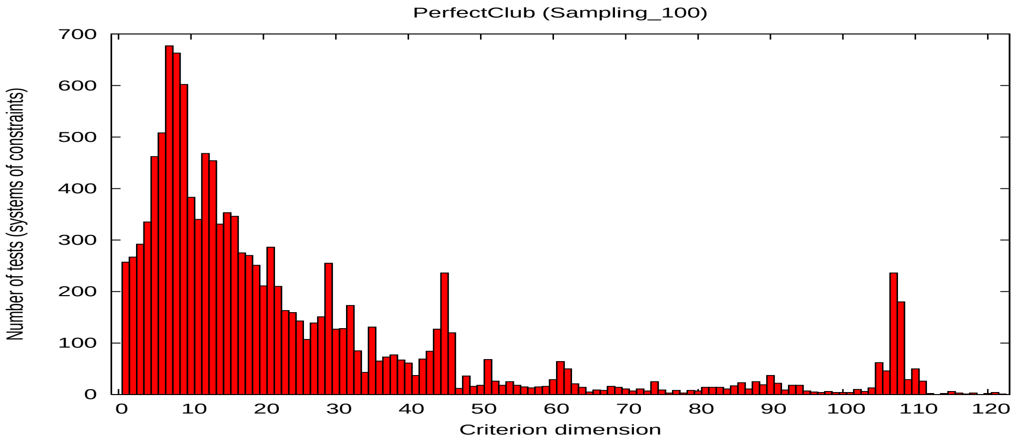


Figure 5.9: PerfectClub: dimension distribution of sampled database for satisfiability test

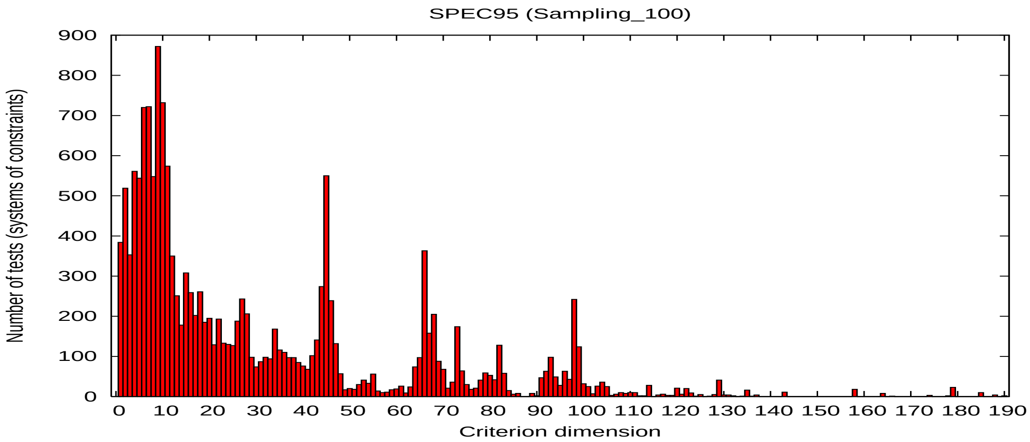


Figure 5.10: SPEC95: dimension distribution of sampled database for satisfiability test

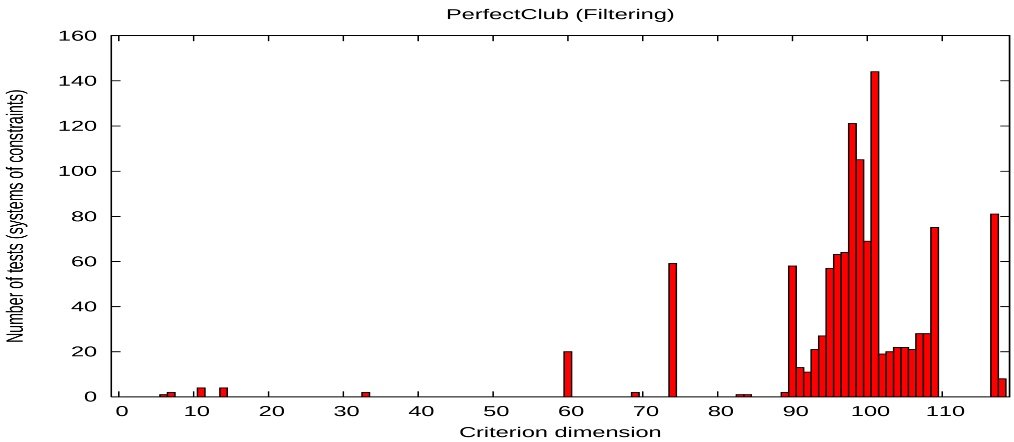


Figure 5.11: PerfectClub: dimension distribution of filtered database for satisfiability test

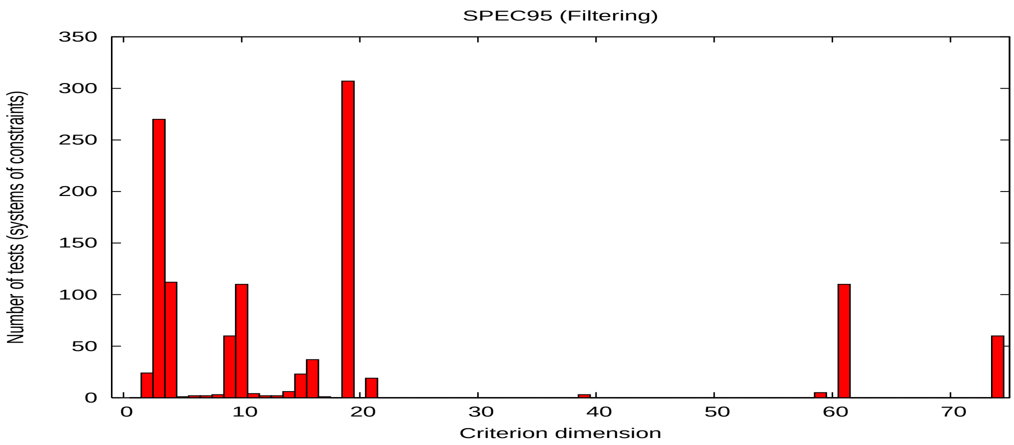


Figure 5.12: SPEC95: dimension distribution of filtered database for satisfiability test

2.10 Evaluation of POLYBENCH and Future Work

In this section, we talk about the advantages of our approach, its limitations and the future of POLYBENCH. First of all, we summarize some advantages as follows:

- Methodology: Easy and straightforward comparison of performance and stability;
- Quality of comparison: large databases from an existing static analyzer;
- Presentation of results in graphical form and tables.
- Execution is fully automated;
- Static program analysis related test sets;
- POLYBENCH's input format is human legible;
- POLYBENCH supports variable names and thus provides variable origin tracking in debugging.

Alongside the above advantages, limitations of our approach are also present:

- Work required for new implementation;
- Comparisons can be inhomogeneous;
- Missing memory usage comparisons;
- Chosen criteria are not fully satisfying, i.e. heuristic-based approach is not applicable;
- Head-to-head comparisons require several executions of the same implementation, since exception cases are excluded from execution time measurements;

- PIPS-dependence: version development.

Since we intend to remove these limitations, here we discuss them in more details before presenting the experimental part.

The diversity of algorithms and implementations yields the most important limit of our framework. The diversity consists of availability of algorithms, algorithmic differences, implementation differences, etc. As such, the comparisons sometimes are not homogeneous.

For example, many integer algorithms are not yet designed because of complexity, e.g. the convex hull operator, or JANUS [SOG] was originally implemented with C built-in arithmetic, thus only 32-bit computation [6] is enabled, whereas most polyhedral implementations support 64-bit integer arithmetic. Some libraries implement GNU multi-precision, e.g. New POLKA [JEAb, JEA00], some others do not, e.g. C^3 [PIPa, CAI00]. Therefore, although the 64-bit vs GNU multi-precision computing question seems to be more interesting than the 32-bit vs 64-bit computing question in our point of view, the corresponding implementations are not available.

In particular, CDD [FUK] implements C built-in floating point and GNU multi-precision arithmetic, and it sometimes uses both at the same time though usually one uses only one.

The second most important limit of our framework is that it does not support yet the memory consumption comparisons, due to its already complicated structure. We intend to implement this feature in the future.

In order to analyze a new algorithm or implementation, an internal representation conversion from our format is required. This is not a simple work, errors or incompatibilities may appear. The conversion time is included in comparisons, which is not trivial[7]. Some of available implementations are not tested, due to problems of time and bugs. Furthermore, in order to compare the results computed by different implementations, each result in its own internal format must be converted to our format. This conversion is different from the above-mentioned conversion, thus requires additional work. That is why, in case of New POLKA, the output is not yet verified.

In our framework, representations of results are based on run times and polyhedral criteria, such as number of dimensions, number of constraints, etc. Thus, for operators that take several arguments, e.g. the convex hull operator, the representation is not trivial since run times depend on all arguments. We can for instance take the criteria of the largest argument, but we actually take the first polyhedron's criteria, in the convex hull evaluation.

Another limit of our framework is that, while a great number of tests is generated in order to increase the exactitude of experiments, it requires days of experimentation. If we deploy powerful computers, we can reduce the time, but then the time of each execution

[6] Ported to 64-bit by me.

[7] In C^3 point of view, it should be included since it actually impacts the execution time.

may go under the time resolution which is based on quanta, thus comparisons becomes imprecise.

On the contrary, if we use a small set of tests, deploying slow computer, we obtain more precision in timing comparison. Thus a sampling mechanism is implemented in our framework. For execution times that are too small, a repeating mechanism for time measurement is implemented since it is the cheapest solution. We plan to implement an adaptive schema in the future.

Our database creation schema depends strongly on PIPS's infrastructure, in order to put the evaluations in the program analysis and transformation context. Thus, these databases may contain polyhedra that are not suitable in other context. We notice that though there is redundancy in our databases, it is not considered a problem, since the databases are constructed and used independently for each operator, and because it reflects the reality of static analyzers. In fact, we can test a memorization scheme for this, as well as for head-to-head comparisons to avoid several executions of the same implementation, but we decide to implement them in the future due to their lower priority.

3 Results for Satisfiability Test

From Section 3.1 to Section 3.5, we compare JANUS [8] 64-bit, denoted JV64 for short, with the three implementations found in C^3: Simplex 64-bit, denoted LS64, Fourier-Motzkin 64-bit, denoted FM64, and the satisfiability test using the double description method 64-bit, denoted FDD64 [9].

Then in Section 3.6, we compare two different versions of each implementation: JANUS 64-bit versus JANUS 32-bit, denoted JV32; C^3 Simplex 64-bit versus C^3 Simplex 32-bit, denoted LS32; C^3 Fourier-Motzkin 64-bit versus C^3 Fourier-Motzkin 32-bit, denoted FM32; and finally C^3 satisfiability test using the double description method 64-bit, denoted FDD64, versus C^3 satisfiability test using the double description method 32-bit, denoted FDD32. The polyhedral databases used here are the satisfiability test databases (see Section 2.8).

3.1 JANUS 64-bit versus C^3 Simplex 64-bit

3.1.1 Random Sampling Database of PerfectClub

In Figure 5.13, the histogram legends show three colors red, blue and green. The meanings of these legends are: the red zones present the cases where JANUS 64-bit is faster than C^3 Simplex 64-bit; the green zones correspond to the cases where C^3 Simplex 64-bit is faster than JANUS 64-bit; the blue zones mean that our implementation cannot tell which one is faster than the other, i.e. the run times cannot be compared because of the time resolution used.

This histogram has two axes representing the numbers of variables in the constraint systems, also known as the space dimensions of the corresponding polyhedra, and the number of constraint systems available in PerfectClub polyhedral sampled database, for the satisfiability test operator. We can see this information in the first line of the figure header, as well the percentages: JANUS 64-bit is faster for 25 percent of all tests, C^3 Simplex 64-bit is never faster. It also means that in 75 percent of cases we cannot compare the execution times, due to the time resolution.

Furthermore, total accumulated run times for all tests are compared, from which the global acceleration is derived and displayed in the second line of figure header: the total run time of C^3 Simplex 64-bit is divided by the total run time of JANUS 64-bit which shows that JANUS 64-bit is approximately 22 times faster than C^3 Simplex 64-bit on average.

The same information is represented in Figure 5.14 and Figure 5.15, with respect to the number of equations and number of inequalities respectively.

[8] JANUS was developed with 32-bit arithmetic. It was extended to 64-bit and added exception management by me, see Chapter 4, Section 7.2.3, Page 98.

[9] Developed by me, using Chernikova's algorithm.

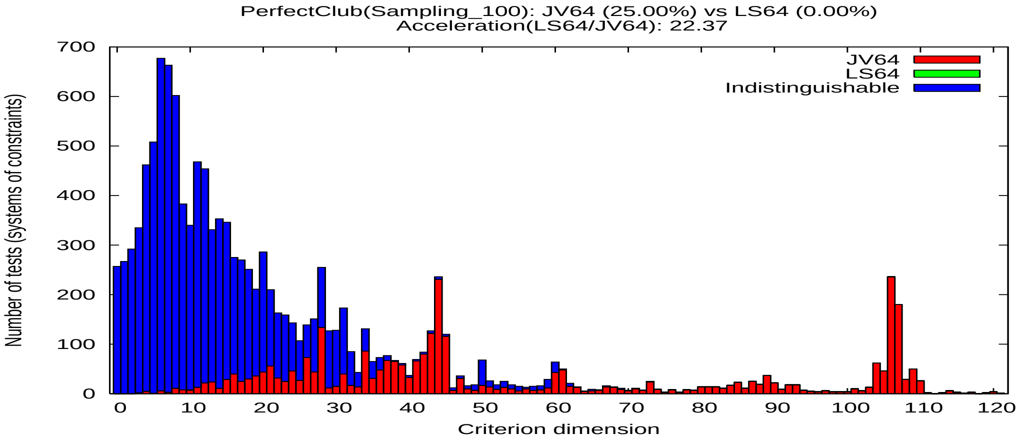


Figure 5.13: PerfectClub: Dimension JANUS 64-bit vs C^3 Simplex 64-bit in sampled database

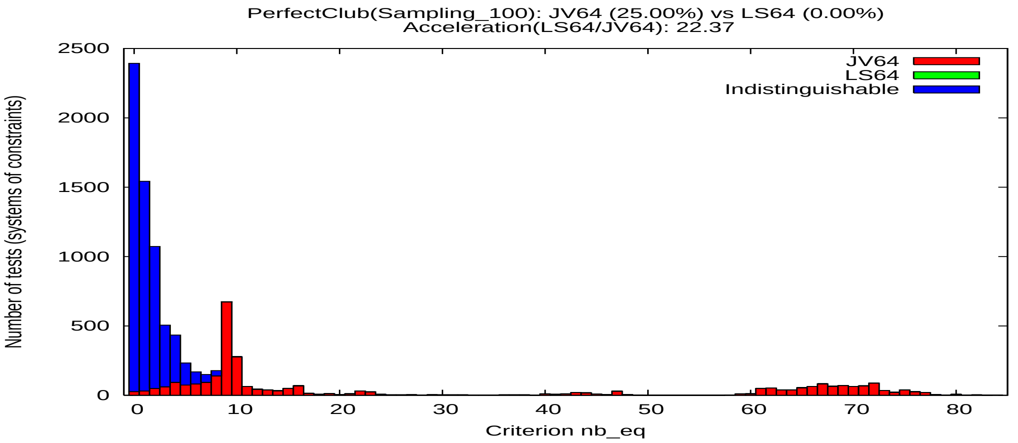


Figure 5.14: PerfectClub: Numbers of equations JANUS 64-bit vs C^3 Simplex 64-bit in sampled database

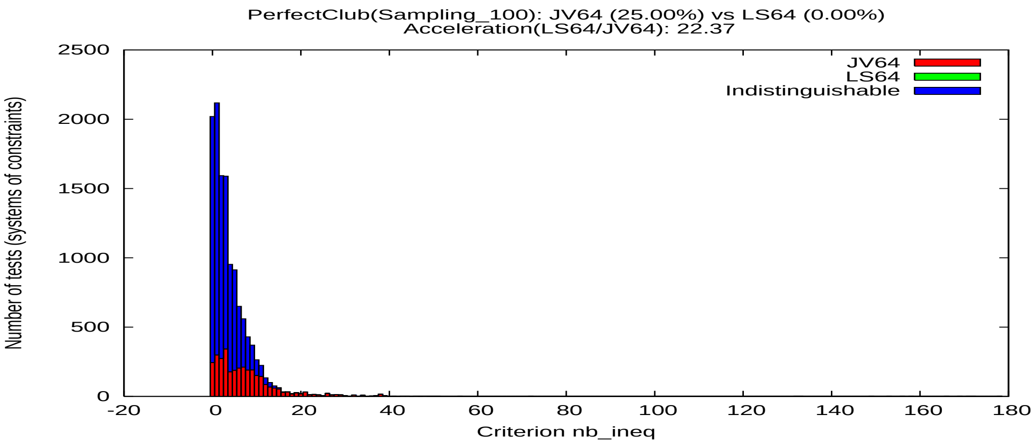


Figure 5.15: PerfectClub: Numbers of inequalities JANUS 64-bit vs C^3 Simplex 64-bit in sampled database

We can see that the histogram with dimension axis is more informative than the other two. This remark holds for all the results, hence from now on we only present histograms with respect to the dimension.

3.1.2 Biased Database of PerfectClub

Figure 5.16 represents the experimental results using a filtered database, which is a biased database as explained in Section 2.8, for JANUS 64-bit and C^3 Simplex 64-bit.

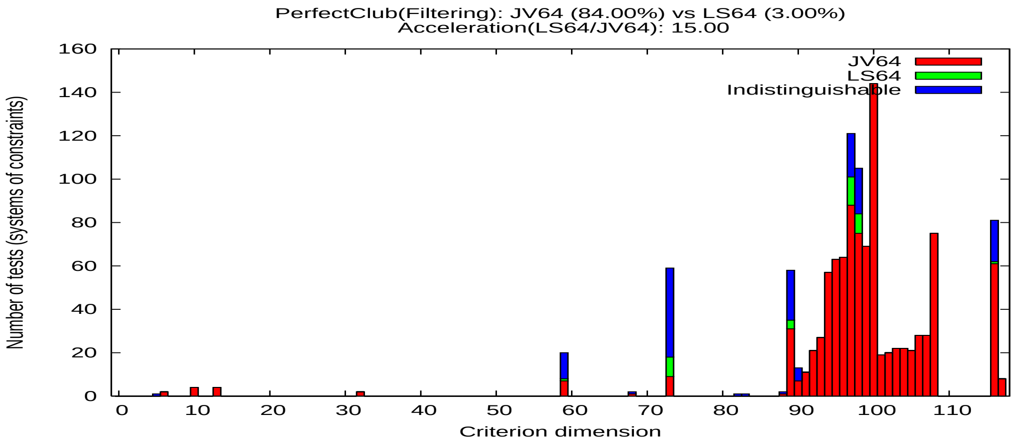


Figure 5.16: PerfectClub: Dimension JANUS 64-bit vs C^3 Simplex 64-bit in filtered database

We notice that there are cases where JANUS 64-bit is slower, shown by the green zones, but the conclusion for run times of these two algorithms is that JANUS 64-bit is faster than C^3 Simplex 64-bit, for both PerfectClub sampled database and filtered database. The results of SPEC95 benchmark are not presented here because they are very similar to PerfectClub.

3.1.3 Parallel Algorithm

We particularly notice that in the case of sampled databases, the performance of the parallel algorithm[10] is equivalent to JANUS's, which is about 22 times faster than C^3 Simplex in PerfectClub case and 73 times faster in SPEC95 case. For the filtered databases, the parallel algorithm PA is as fast as JANUS, and about 15 times faster than C^3 Simplex, for PerfectClub benchmark; JANUS is 30 times faster than C^3 Simplex for the SPEC95 benchmark.

3.2 JANUS 64-bit versus C^3 Fourier-Motzkin 64-bit

Here we compare JANUS 64-bit and C^3 Fourier-Motzkin 64-bit: Figure 5.17 and Figure 5.18 represent the experimental results of PerfectClub sampled and filtered databases.

[10]See Page 137 for the definition of our parallel algorithm.

We can observe that JANUS 64-bit is faster: 46 percent with sampled database and 32 percent with filtered database [11]. For the SPEC95, we have 74 and 17 percent, respectively.

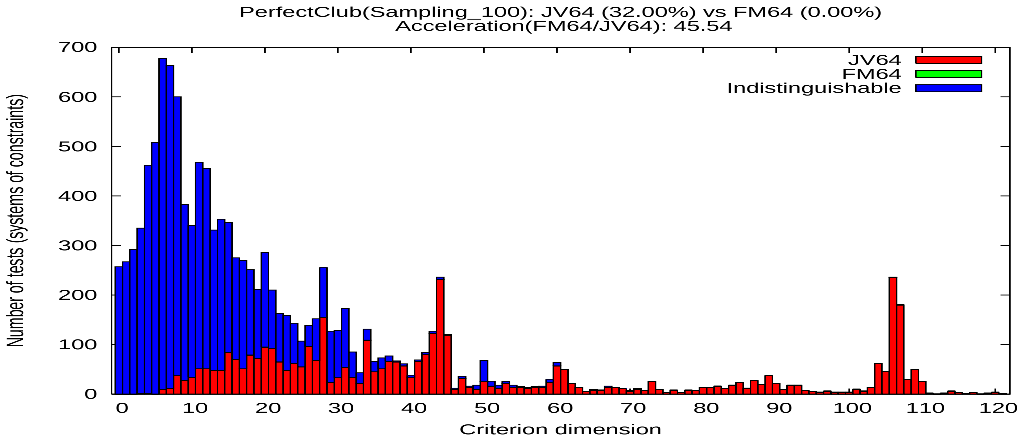


Figure 5.17: PerfectClub: Dimension JANUS 64-bit vs C^3 Fourier-Motzkin 64-bit in sampled database

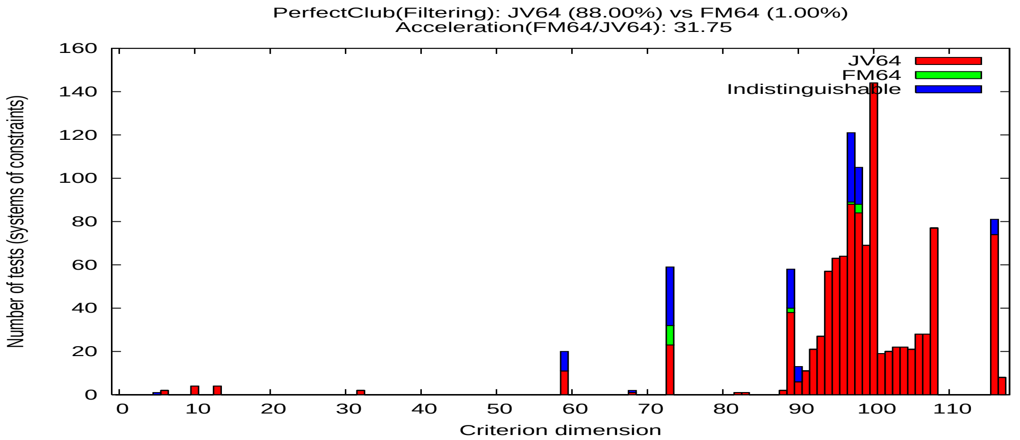


Figure 5.18: PerfectClub: Dimension JANUS 64-bit vs C^3 Fourier-Motzkin 64-bit in filtered database

Parallel Algorithm: As in the previous section, Section 3.1.3, we remark the equivalent performance between the parallel algorithm PA and JANUS in the case of sampled databases and filtered databases.

3.3 JANUS 64-bit versus C^3 Double Description Method 64-bit

Performance comparison between JANUS 64-bit and Double Description Method 64-bit is illustrated in Figure 5.19 and Figure 5.20. They represent the experimental results of

[11] Explications for this kind of graphs can be found at 3.1.

PerfectClub sampled and filtered databases.

We remark a very poor performance of FDD64-bit compared to JANUS 64-bit: from 404 to 520 times slower. Exceptionally in SPEC95 sampled database results, JANUS 64-bit outruns FDD 64-bit with a very large ratio of 5485. Therefore, we conclude that in the satisfiability test, it is much faster avoiding the dual conversion using the Chernikova algorithm. This conclusion is important since only C^3 implements other methods than the Chernikova algorithm. However, as we will see later in the next section, we need the Chernikova algorithm in some cases.

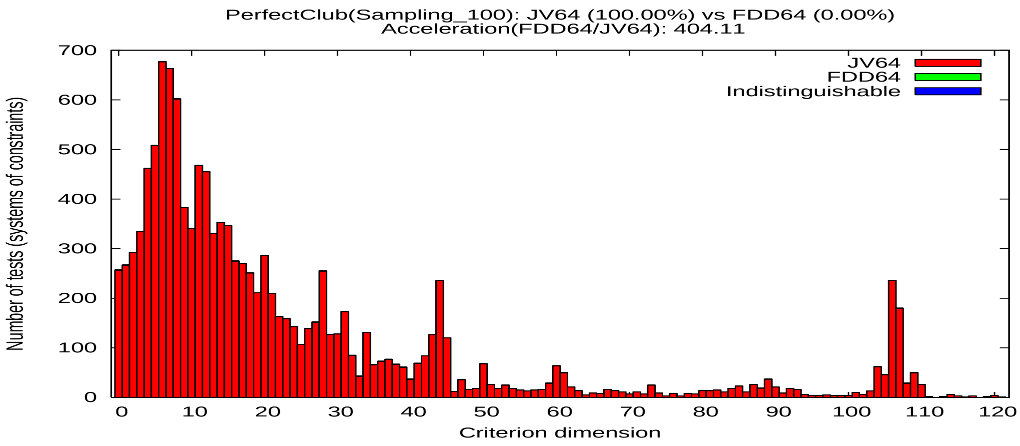


Figure 5.19: PerfectClub: Dimension JANUS 64-bit vs C^3 FDD 64-bit in sampled database

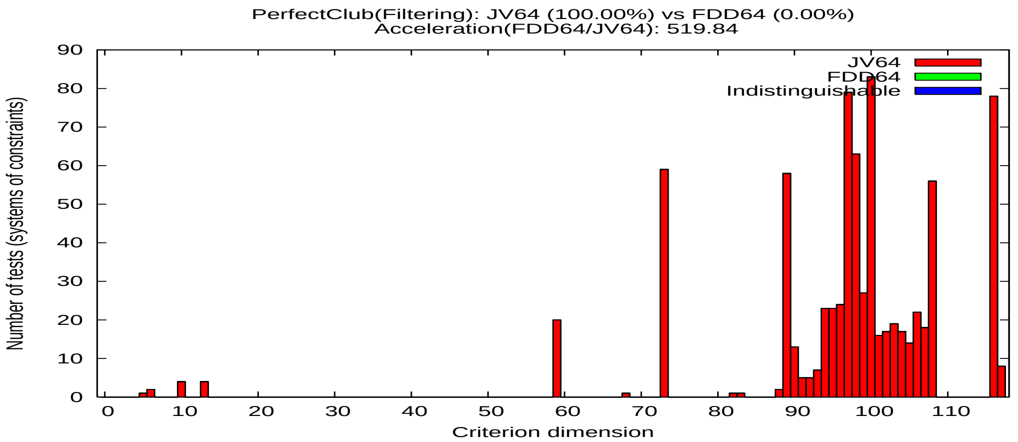


Figure 5.20: PerfectClub: Dimension JANUS 64-bit vs C^3 FDD 64-bit in filtered database

The experimental results for SPEC95 are similar to those obtained with PerfectClub, thus they are not displayed here.

Parallel Algorithm: Because we present the results here as JANUS 64-bit versus the other three, we have a similarity to the previous sections: the equivalent performance

64-bit	**JANUS/PA**	**LS/PA**	**FM/PA**	**FDD/PA**
Sampled	1/1	22/1	46/1	404/1
Filtered	1/1	15/1	32/1	520/1

Table 5.5: PerfectClub: Global execution times compared to the parallel algorithm PA, 64-bit

	PerfectClub		
timeout = 2 minutes	**#overflows**	**#timeouts**	**#operations**
JANUS 64-bit	5	0	12668
C^3 *Simplex 64-bit*	9	0	12668
C^3 *Fourier-Motzkin*	0	2	12668
C^3 *Double Description*	2	7	12668

Table 5.6: PerfectClub: Numbers of exceptions in sampled database

between the parallel algorithm, denoted PA[12], and JANUS using sampled databases and filtered databases.

Table 5.5 summarizes the comparison of global execution times for satisfiability test, 64-bit. We remark that JANUS 64-bit outruns the other three and its performance is equivalent to the parallel algorithm PA. Now we study the problem of exceptions in the next section.

3.4 Overflow and Timeout Exceptions: 64-bit

Previous comparisons between JANUS 64-bit and C^3's three algorithms only concern measured execution times, which is in fact not complete. In this section, we compare the numbers of exceptions: overflow exceptions, which occur with numbers too large for computation, and timeout exceptions, when execution time of a test is longer than two minutes, which is arbitrarily considered not acceptable.

Sampled databases: Table 5.6 shows numbers of exceptions for JANUS 64-bit, C^3 Simplex, Fourier-Motzkin and Double Description method, and the number of tests in PerfectClub sampled database. One notices that JANUS is more robust than C^3 Simplex: 5 to 9 overflows. It has no timeout exception, while C^3 Fourier-Motzkin raises 2 timeout exceptions and C^3 Double Description method raises 2 overflows only and 7 timeout exceptions that may block our analysis. We notice that Fourier Motzkin has no overflow exception.

With SPEC95, sampled database, we have another result: JANUS 64-bit has more exceptions than C^3 Simplex (52 to 51), and more than C^3 Fourier Motzkin method (52

[12]See Page 137 for the definition of our parallel algorithm.

	SPEC95		
timeout = 2 minutes	**#overflows**	**#timeouts**	**#operations**
JANUS 64-bit	52	0	16303
C^3 *Simplex 64-bit*	51	0	16303
C^3 *Fourier-Motzkin*	1	14	16303
C^3 *Double Description method*	717	0	16303

Table 5.7: SPEC95: Numbers of exceptions in sampled database

	PerfectClub		
timeout = 2 minutes	**#overflows**	**#timeouts**	**#operations**
JANUS 64-bit	0	0	1310
C^3 *Simplex 64-bit*	2	0	1310
C^3 *Fourier-Motzkin*	0	0	1310
C^3 *Double Description*	0	407	1310

Table 5.8: PerfectClub: Numbers of exceptions in filtered database

to 1) Table 5.7. We also observe that Fourier Motzkin method has 14 timeout exceptions, and there are many overflow exceptions with the Double Description method: 717.

Fourier-Motzkin Additional Tests for Timeout Exceptions: Since the number of timeout exceptions raised by the Fourier-Motzkin algorithm is rather small, 2 in Table 5.6 and 14 in Table 5.7, and from the fact that the inequalities combination in this algorithm is explosive (see Chapter 4, Section 7.2.1), we have tested these constraint systems again without the 2 minutes timeout activation. The result is that they all raised out-of-memory space exceptions. This means that if the Fourier-Motzkin method takes more than two minutes, it likely results in much longer execution time before an out-of-memory space occurs.

Thus our conclusion for the sampled databases is that JANUS 64-bit and Simplex are more robust than the other two algorithms. Now we consider the filtered databases.

We need to remind the reader that Tables related to filtered databases cannot be compared to Tables related to sampled databases.

Filtered databases: Table 5.8 shows that with PerfectClub filtered databases, JANUS 64-bit, Fourier-Motzkin and Double Description method has no overflow exception, whereas C^3 Simplex has only two overflows. Only C^3 Double Description method raises as many as 407 timeout exceptions that may block our analysis.

It is however very different with SPEC95, filtered database , as shown in Table 5.9: C^3 Simplex has 3714 overflow exceptions, 727 more than JANUS 64-bit. C^3 Fourier Motzkin

	SPEC95		
timeout = 2 minutes	**#overflows**	**#timeouts**	**#operations**
JANUS 64-bit	2987	0	4676
C^3 *Simplex 64-bit*	3714	0	4676
C^3 *Fourier-Motzkin*	8	626	4676
C^3 *Double Description*	22	0	4676

Table 5.9: SPEC95: Numbers of exceptions in filtered database

Filtered databases	**PerfectClub**	**SPEC95**
ASCII file size	5596.73	2496.27
Average #Dimensions	46.97	10.28
Average #Equations	17.40	2.35
Average #Inequalities	38.89	52.56
Average #Constraints	56.30	54.91
Average #Sparsity index	3.90	3.86
Average #Vertices	1118.76	18.66
Average #Rays	69.10	14.07
Average #Lines	6.78	0.53

Table 5.10: Average polyhedral size statistic of filtered database

method has only 8 overflows and 626 timeout exception. And the winner in this case is surprisingly the C^3 Double Description method, with only 22 overflows and no timeout exception. To understand this difference, we decided to investigate these two filtered databases.

Polyhedral size statistics: For the filtered databases, we have seen different results between PerfectClub and SPEC95 tests: the C^3 Double Description method has many timeout exceptions with PerfectClub tests but it has only 22 overflows and no timeout exception, much better performance compared to the other three algorithms[13].

Table 5.10 compares the two databases, where the differences in space dimensions, numbers of vertices, rays and lines are important since the Double Description method is related to these factors[14]. Therefore, the cause of the difference is the polyhedral characteristics of the filtered databases.

Parallel Algorithm: The parallel algorithm could be useful if we wish to reduce the number of exceptions. In fact, we have implemented in C^3 some variations of the parallel

[13]We recall that the total measured execution times is another aspect, where JANUS is proved faster.

[14]The Double Description implementation in fact computes the vertices, rays and lines from given constraint systems using Chernikova algorithm.

	PerfectClub	
integer vs rational	**declared feasible**	**#operations**
JANUS 64-bit	0	12668
C^3 *Simplex 64-bit*	5	12668
C^3 *Fourier-Motzkin*	1	12668
C^3 *Double Description method*	8	12668

Table 5.11: PerfectClub: Numbers of *not precise* results in sampled database

algorithm which uses the three algorithms JANUS, Simplex and Fourier-Motzkin. For example, we try the Fourier-Motzkin method when JANUS has an exception.

The implementation of POLYBENCH permits us to verify that in Table 5.6, the five overflow exceptions from JANUS and the exceptions from other three algorithms come from different constraint systems. It means if we use the parallel algorithm using JANUS and another algorithm, we can resolve all the tests. For instance, for this database, the parallel algorithm of JANUS and Simplex has zero exception. For the SPEC95 sampled database, JANUS and Simplex share the same 39 overflow exceptions, which represents the number of exceptions that their parallel algorithm has. If we use the parallel algorithm of the four algorithms, we have only one exception, instead of 52 exceptions with JANUS.

In the case of filtered databases, we can for example use the Fourier-Motzkin in Table 5.8 and the Double Description method Table 5.9 for the constraint systems that JANUS cannot deal with.

3.5 Integer versus Rational: 64-bit

In above comparisons, we have ignored a fact that algorithms implemented for the satisfiability test can be integer or rational. The integer algorithm tests if the constraint system contains integer points or not, whereas the rational tests if the constraint system contains rational points or not. The algorithm implemented in JANUS is integer where C^3 Simplex and Double Description method are rational. C^3 Fourier-Motzkin is a rational algorithm with an *add-on* test that in some cases can verify if the solution are rational or integer. Thus, it is in fact an *integer/rational* algorithm. In our context, integer answer means more precision than rational one, therefore in this section we compare the differences between results of those algorithms.

Table 5.11 and Table 5.12 show numbers of cases where C^3 Simplex, Fourier-Motzkin and Double Description method give the answer *notempty* while the constraint system contains no integer point but only rational points, with sampled databases [15]. We can see that the percentage of *not precise* results is rather small. This suggests that the difference is not significant.

[15] While less precise, it is not a problem for program analysis since the approach is conservative.

	SPEC95	
integer vs rational	**declared feasible**	**#operations**
JANUS 64-bit	0	16303
C^3 *Simplex 64-bit*	4	16303
C^3 *Fourier-Motzkin*	0	16303
C^3 *Double Description method*	3	16303

Table 5.12: SPEC95: Numbers of *not precise* results in sampled database

3.6 Arithmetic Precision: 64-bit versus 32-bit

Here we compare the difference between 64-bit and 32-bit implementations of algorithms for the satisfiability test. This comparison answers the question: which precision should we adopt, 32-bit, 64-bit or GNU multi-precision? This question is raised since we do know that 32-bit computation can be faster and requires less memory space than 64-bit, and much faster than GNU Multi Precision, but we do not know exactly how much of precision we lose because of computing overflows. Unfortunately we do not have a complete set of implementations supporting GNU multi-precision, therefore comparisons between 64-bit and GNU multi-precision are not yet available.

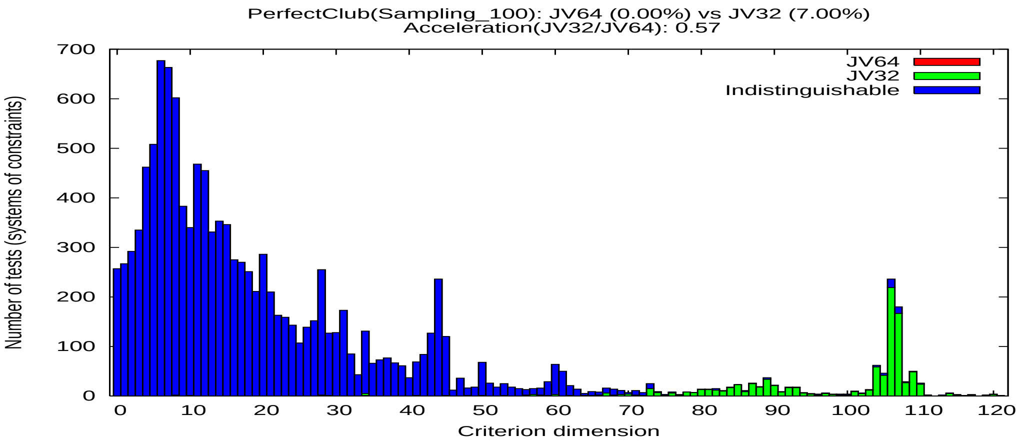


Figure 5.21: PerfectClub: Dimension JANUS 64-bit vs JANUS 32-bit in sampled database

Sampled databases: We compare the run times between two versions, 32-bit and 64-bit, of each algorithm. Then we compare the numbers of exceptions between 32-bit and 64-bit versions of these implementations. In Figure 5.21, Figure 5.22 and Figure 5.23, we can see that the execution time ratios are between 0.57 and 0.84, thus the sacrifice in execution time for using $64-bit$ (higher precision) instead of $32-bit$ is to be considered.

Table 5.13 shows that for PerfectClub sampled database, numbers of exceptions are fewer using the higher precision as expected. We have similar results with SPEC95.

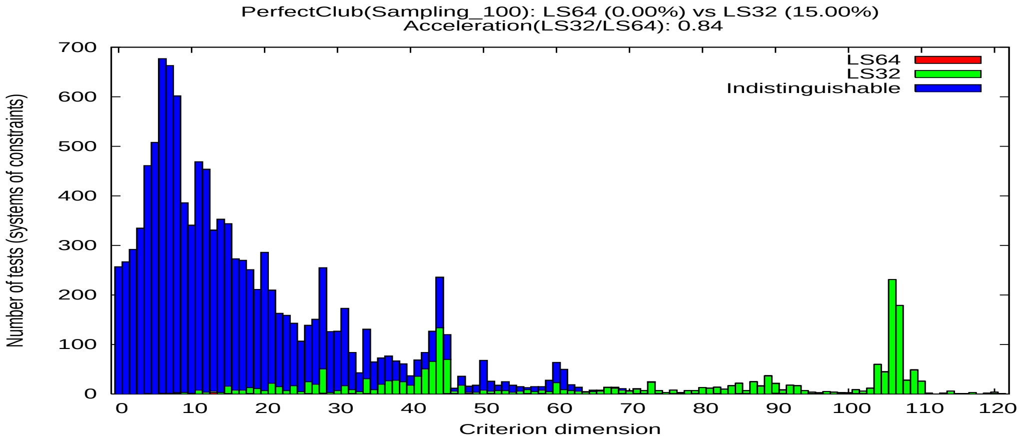


Figure 5.22: PerfectClub: Dimension C^3 Simplex 64-bit vs 32-bit in sampled database

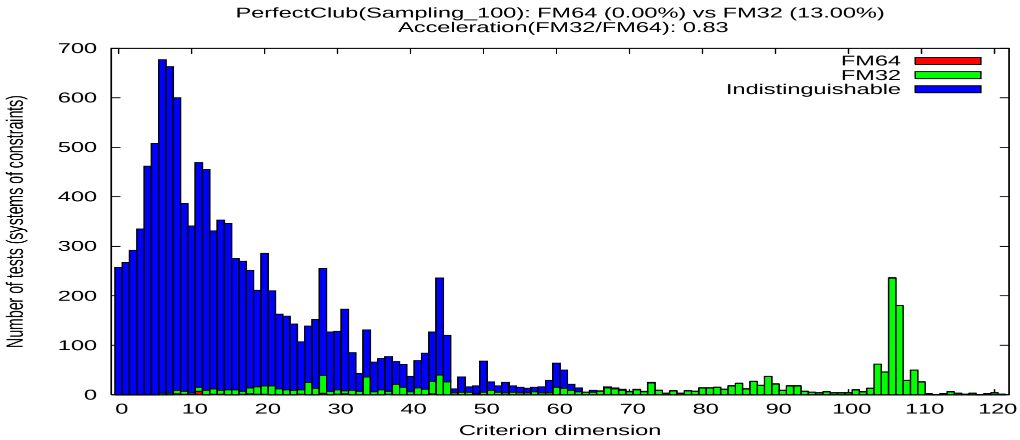


Figure 5.23: PerfectClub: Dimension C^3 Fourier-Motzkin 64-bit vs 32-bit in sampled database

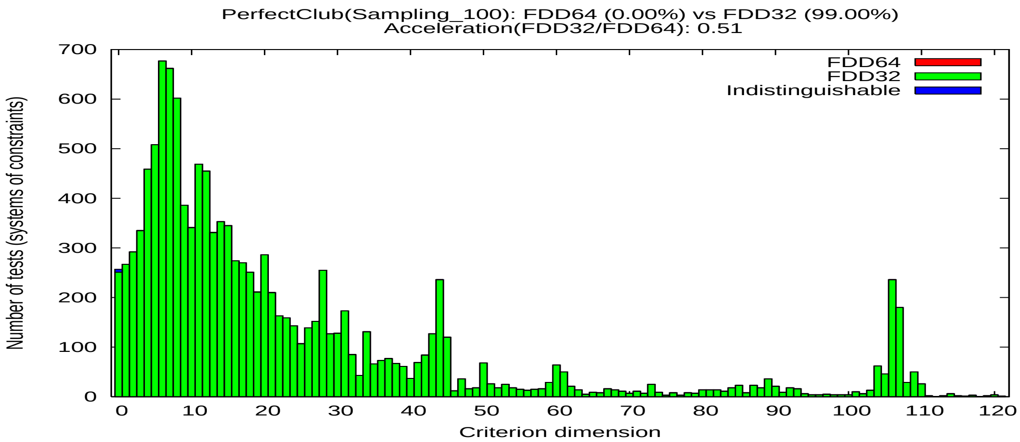


Figure 5.24: PerfectClub: Dimension C^3 Fourier-Motzkin 64-bit vs 32-bit in sampled database

	PerfectClub		
timeout = 2 minutes	**#overflows**	**#timeouts**	**#operations**
C^3 *JANUS 64-bit*	5	0	12668
C^3 *JANUS 32-bit*	8	0	12668
C^3 *Simplex 64-bit*	9	0	12668
C^3 *Simplex 32-bit*	59	0	12668
C^3 *Fourier-Motzkin 64-bit*	0	2	12668
C^3 *Fourier-Motzkin 32-bit*	1	2	12668
C^3 *Double Description 64-bit*	2	7	12668
C^3 *Double Description 32-bit*	18	7	12668

Table 5.13: PerfectClub: Numbers of exceptions in sampled database

	PerfectClub		
timeout = 2 minutes	**#overflows**	**#timeouts**	**#operations**
C^3 *JANUS 64-bit*	0	0	1310
C^3 *JANUS 32-bit*	0	0	1310
C^3 *Simplex 64-bit*	2	0	1310
C^3 *Simplex 32-bit*	92	0	1310
C^3 *Fourier-Motzkin 64-bit*	0	0	1310
C^3 *Fourier-Motzkin 32-bit*	0	0	1310
C^3 *Double Description 64-bit*	0	407	1310
C^3 *Double Description 32-bit*	50	371	1310

Table 5.14: PerfectClub: Numbers of exceptions in filtered database

Filtered databases: If we only consider execution times, the filtered databases have similar results, though not presented here, as the sampled databases. But since the constraint systems are of larger sizes, we expect to have a different picture concerning the exceptions.

In fact, Table 5.14 shows that C^3 Simplex 64-bit has reduced from 92 exceptions to 2 exceptions compared to its version 32-bit. The Double Description method 32-bit raises 50 overflow exceptions among which 36 timeout exceptions, i.e. $407 - 371$, are expected using its 64-bit version.

Table 5.15 shows that using 32-bit instead of 64-bit results in many more overflow exceptions. The Double Description method has the biggest difference: 939 to 22.

3.7 Conclusion

First of all, we notice that sometimes exceptions could heavily penalize the execution. Let us take an example with an empty system of hundreds of constraints that raises an

	SPEC95		
timeout = 2 minutes	**#overflows**	**#timeouts**	**#operations**
C^3 *JANUS 64-bit*	3050	0	4676
C^3 *JANUS 32-bit*	3109	0	4676
C^3 *Simplex 64-bit*	3174	0	4676
C^3 *Simplex 32-bit*	3972	0	4676
C^3 *Fourier-Motzkin 64-bit*	8	625	4676
C^3 *Fourier-Motzkin 32-bit*	181	495	4676
C^3 *Double Description 64-bit*	22	0	4676
C^3 *Double Description 32-bit*	939	0	4676

Table 5.15: SPEC95: Numbers of exceptions in filtered database

overflow exception for a 32-bit implementation for the satisfiability test. This constraint system may continue to expand in a sequence of much larger systems to be manipulated in our analyses. If the 64-bit implementation can solve this constraint system, i.e. without exception, then instead of manipulating systems of hundreds of constraints, we only have to deal with an empty system with no constraint, which means much faster execution.

We can see that 64-bit computation is more accurate than 32-bit computation for at most a half execution time slow-down. Experiences show that even with 64-bit computation, a very important number of overflow exceptions are encountered, thus 64-bit is preferable.

In general, JANUS 64-bit has a better performance over C^3 Simplex 64-bit, Fourier-Motzkin 64-bit and Double Description Method 64-bit. However, the fact that C^3 Fourier-Motzkin 64-bit and Double Description method can solve a number of constraint systems that JANUS 64-bit cannot (see Section 3.4), raises a question: Is it worth to use them in such rare cases?

We remark that C^3 *Fourier-Motzkin* and Double Description method have problem mostly with timeout. Then one may consider them as an ultimate choice when we absolutely want a definitive answer. However, we cannot assure the termination of these algorithms, because memory space is limited.

For the *Simplex* algorithm, we can see that magnitude is the main problem, not the explosion of system size as for Fourier-Motzkin algorithm. So a treatment of large numbers might resolve the case. For example, we might try GNU multi-precision library instead of 64-bit computing.

The degenerated polyhedra is another problem for *Simplex* method, so pre-processing phases are to be studied. We can also use approaches proposed in [Mer05], such as Cartesian factorization or dimension space mapping. For a large constraint system that none of the above implementations can deal with, we do not know whether a polyhedron decomposition can improve the situation or not ([Mer05], page 85 to 91, and Chapter 4, Section 10.2.3).

The poor performance of the satisfiability test using Double Description method with the sampled databases and PerfectClub filtered database suggests that New POLKA, POLYLIB and PPL libraries that use this approach should meet difficulties when dealing with large size constraint systems.

For the SPEC95 filtered database, unfortunately, we cannot build any heuristics to take advantages of the Double Description method. We are aware of the size estimating algorithm presented in CDD library [FUK], but it is not cheap enough to be used as heuristics.

Finally, our experimental results have shown an unexpected fact: The most relevant criterion is the number of dimensions but not the number of constraints. Criteria more predictive than the dimension space have not been found. In most of cases, the dimension space cannot provide good heuristics, i.e. we cannot provide a cut which separates the red zones and the green zones in those histograms in order to achieve the best performance out of two implementations.

4 Results for Projection

4.1 C^3 approach: Constraints versus Generators, 64-bit

In this section, we compare the 64-bit implementation that manipulates directly on constraint systems, i.e., the Fourier-Motzkin elimination method, denoted P64, and the implementation using the double description method for 64-bit, denoted PDD64, which projects the polyhedron by converting its constraint system to a generating system and then does the projection using this representation [16]. The first one was already available in C^3, and the second one was implemented by me, in order to evaluate the two approaches.

Our comparisons only make sense when we have in hand the constraint systems but not their corresponding generating systems at the same time. This is normally the case in C^3 library. For New POLKA library, the generating system of a polyhedron can be present or not, depending on the context. POLYLIB library always keep the two representations of the polyhedron in question, thus our comparison does not make sense except the fact that we can check the equivalence of the outputs of different implementations.

The polyhedral databases used here are the projection sampled databases (see Section 2.8). In Figure 5.25, the histogram legend shows three colors red, blue and green. However, in this case, we only have the red zones which present the cases where constraint manipulation P64 is faster than the double description method PDD64 [17].

This histogram has two axes representing the number of variables in constraint systems and the number of constraint systems available in PerfectClub polyhedral sampled database for the projection operator. We can see this information in the first line of the figure header, as well the percentages: P64 is faster in all tests, PDD64 is never faster.

Furthermore, total accumulated run times of all tests are compared, from which the global acceleration is derived and displayed in the second line of figure header: the total run time of PDD64 divided by the total run time of P64 which shows that P64 is approximately 1253 times faster than PDD64.

As for satisfiability test in Section 3, histograms with dimension axis are more informative than the others.

The conclusion about the run times of two algorithms is P64 is much faster than PDD64, for PerfectClub sampled databases: 450 times faster. The same results are seen for SPEC95 databases, but not displayed here.

The big difference in performance of P64 and PDD64 suggests that in case of performance problems for *Double-Description-method-based* libraries such as New POLKA and PPL [18], implementations using Fourier-Motzkin elimination method might be a solution.

---

[16] The conversion time is much more important, since the projection using generating system is very simple.

[17] See 3.1 for more explication of these histograms.

[18] These two libraries does not always compute the generating system representation like the POLYLIB. Since POLYLIB always keep the dual representations at the same time, it is not a problem.

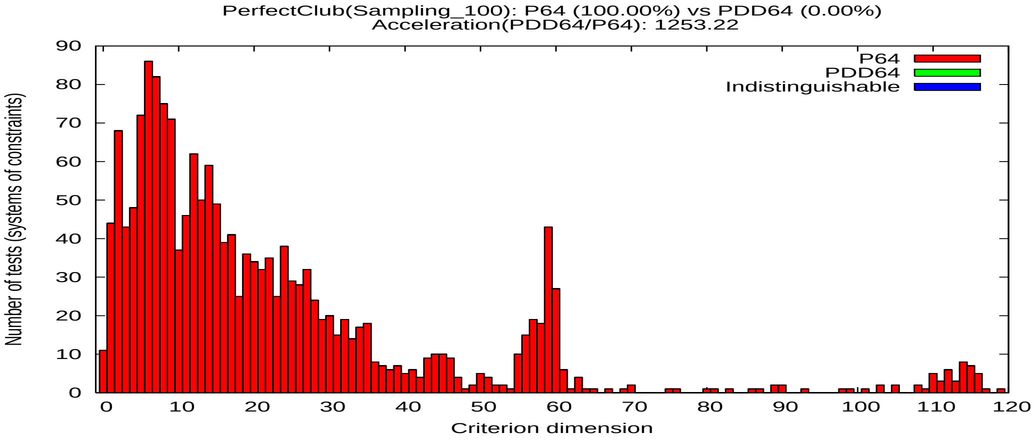


Figure 5.25: PerfectClub: Dimension P64 vs PDD64 in sampled database

	PerfectClub		
timeout = 2 minutes	**#overflows**	**#timeouts**	**#operations**
P64	0	0	1789
PDD64	19	15	1789

Table 5.16: PerfectClub: Numbers of exceptions in sampled database

4.2 Overflow and Timeout Exceptions: 64-bit

In this section, we compare the numbers of exceptions: overflow exceptions, raised when appears a number which is too large for computation, and timeout exceptions, when the execution time of a test is longer than two minutes, which is arbitrarily considered not acceptable.

Table 5.16 shows the numbers of exceptions for P64 and PDD64, and the number of all tests in PerfectClub projection sampled database. P64 is more robust than PDD64, because it has no overflow nor timeout, whereas PDD has 19 overflows and 15 timeout exceptions that may block our analysis. For the SPEC95 projection sampled database with 2583 tests, we however do not have any exception.

4.3 Arithmetic Precision: 64-bit versus 32-bit

Figure 5.26 represents the same results of P64 and P32 comparisons with PerfectClub projection sampled database, which shows P32 is faster than P64, as for the other databases[19].

Figure 5.27 also represents the same results of PDD64 and PDD32 comparisons with PerfectClub projection sampled database, which says that PDD32 is faster than PDD64, as for the other databases. We notice as well that there are cases where the 64-bit version is faster.

[19] There are cases where P64 is faster, but we do not know why.

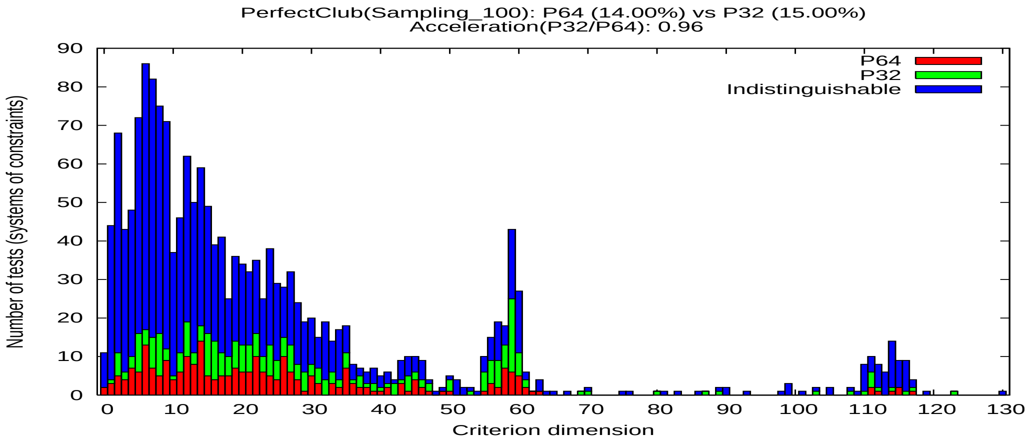


Figure 5.26: PerfectClub: Dimension P64-bit vs P32-bit in sampled database

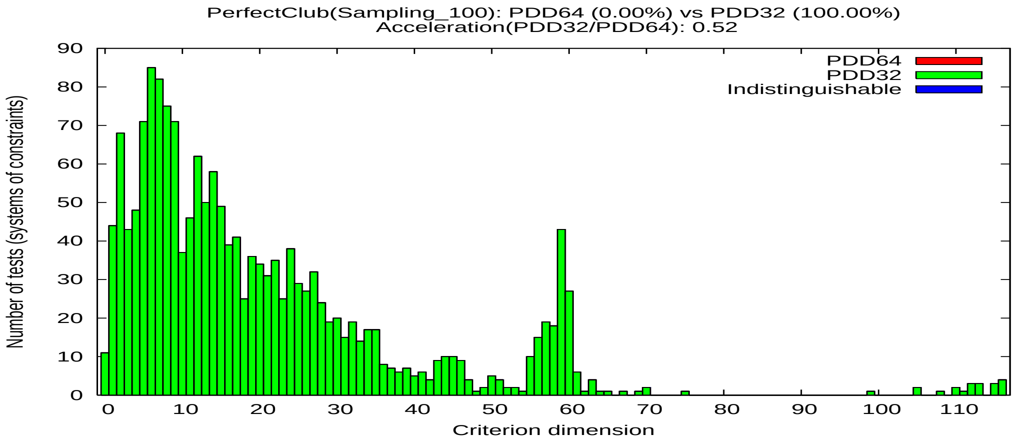


Figure 5.27: PerfectClub: Dimension PDD64-bit vs PDD32-bit in sampled database

	PerfectClub		
timeout = 2 minutes	**#overflows**	**#timeouts**	**#operations**
P64-bit	0	0	1789
C^3 *P32-bit*	0	0	1789
PDD64-bit	19	15	1789
PDD32-bit	66	14	1789

Table 5.17: PerfectClub: Numbers of exceptions in sampled database

Now we compare the run times between two executables, 32-bit and 64-bit of each algorithm. Then we compare the numbers of exceptions between 32-bit and 64-bit versions of these implementations. In Figure 5.26 and Figure 5.27, we see some different ratios of run times which vary from 0.45 to 0.98, thus the sacrifice in execution time for using $64 - bit$ (higher precision) instead of $32 - bit$ is not very important.

Table 5.17 shows that for the set of PerfectClub sampled database and the PDD implementation, numbers of exceptions are fewer using the higher precision. We notice that the number of overflows in 32-bit computation is much higher than in 64-bit (66 to 19 exceptions), compared to the difference of timeout exceptions (15 to 14 exceptions). Meanwhile, the direct constraint manipulation approach shows no difference in exceptions.

4.4 Conclusion

Our conclusion for this part consists of two main points:

- Given the constraint system representation, P64 is better than PDD64 in term of run time performance and number of exceptions;
- The 64-bit is preferable to 32-bit computation for our projection databases.

5 Results for Minimization

5.1 C^3 approach: constraints versus generators, 64-bit

In this section, we compare the 64-bit implementation of the minimization operator that manipulates directly the constraint systems, denoted N64, and the implementation using the double description method for 64-bit, denoted NDD64, which minimizes the polyhedron in question by converting its constraint system to generating system (see Chapter 4, Section 9) [20]. The first one was already available in C^3, and the second one was implemented by me, in order to compare the two approaches. The polyhedral databases used here are the minimization sampled databases (see Section 2.8).

As we have explained in Chapter 4, the semantics of this minimization operator is open enough. Thus the two above algorithms can result in constraint systems which are physically quite different, but should represent two *equivalent* polyhedra. As a matter of fact, this difference does not affect the meaning of our comparison, since we are interested in an effective implementation of the abstract minimization operator with a correct semantics. Our evaluation permits us checking the equivalence of the output of these two approaches.

In Figure 5.28, the histogram shows three colors red, blue and green. The red zones present the cases where constraint manipulation N64 is faster than double description method NDD64, the green zones imply the cases where NDD64 is faster than N64, whereas the blue zones mean that our implementation cannot tell which one is faster than the other, i.e. the run time is about equal, because of the time resolution.

This histogram has two axes representing the number of variables in the constraint system (also known as the dimension space of the polyhedron corresponding), and the number of constraint systems available in PerfectClub polyhedral sampled database. We can see this information in the first line of the figure header, as well the percentages: N64 is faster in 100 percent of all tests, NDD64 is never faster.

Furthermore, total accumulated run times of all tests are compared, from which the global acceleration is derived and displayed in the second line of figure header: the total run time of NDD64 divided by the total run time of N64 which shows that N64 is approximately 450 times faster than NDD64.

As for satisfiability test in Section 3, histograms with dimension axis are more informative than the others.

The conclusion for run times of two algorithms is N64 is much faster than NDD64, for PerfectClub sampled database: 450 times faster. The same results are seen for SPEC95.

The big difference in performance of N64 and NDD64 suggests that in case of performance problems for *Double-Description-method-based* libraries such as New POLKA, POLYLIB and PPL, implementations using direct manipulation on constraint systems can be a solution.

[20] The dual conversion time is the minimization time.

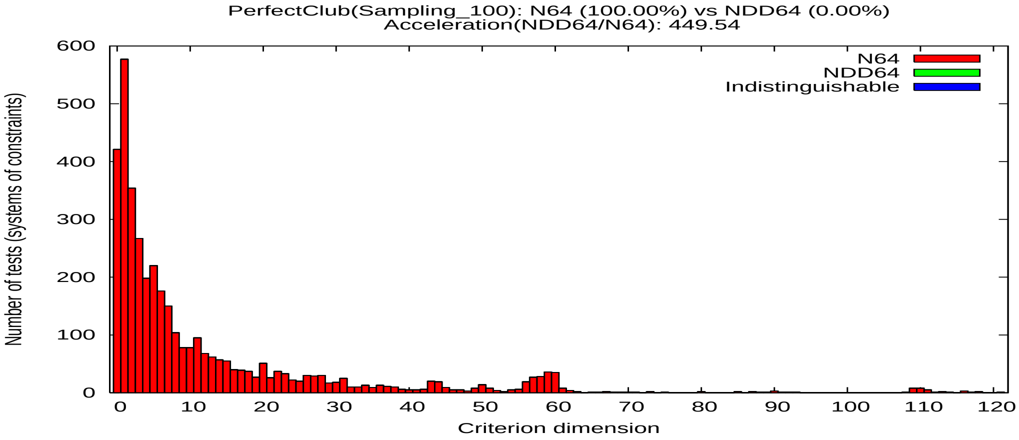


Figure 5.28: PerfectClub: Dimension N64 vs NDD64 in sampled database

	PerfectClub		
timeout = 2 minutes	**#overflows**	**#timeouts**	**#operations**
N64-bit	0	0	3894
NDD64-bit	14	9	3894

Table 5.18: PerfectClub: Numbers of exceptions in sampled database

5.2 Overflow and Timeout Exceptions: 64-bit

In this section, we compare the numbers of exceptions: overflow exception, when appears a number which is too large for computation, and timeout exception, when the execution time of a test is longer than two minutes, which is arbitrarily considered not acceptable.

Table 5.18 shows numbers of exceptions for N64 and NDD64, and the number of all tests in PerfectClub minimization sampled database. One remarks that N64 is more robust than NDD64, because it has no overflow nor timeout, whereas NDD64 has 14 overflows and 9 timeout exceptions that may block our analysis. For the SPEC95 minimization sampled database with 4608 tests, we however did not find any exception.

5.3 Arithmetic Precision: 64-bit versus 32-bit

Figure 5.29 represents the same result of N64 and N32 comparisons with PerfectClub minimization sampled database, which is the N32 is faster than N64 , as for the other databases[21].

Figure 5.30 also represents the same result of NDD64 and NDD32 comparisons with PerfectClub minimization sampled database. NDD32 is faster than NDD64, as for the other databases.

We compare the run times between two executables, 32-bit and 64-bit of each algo-

[21] There are cases where N64 is faster, but we do not know why.

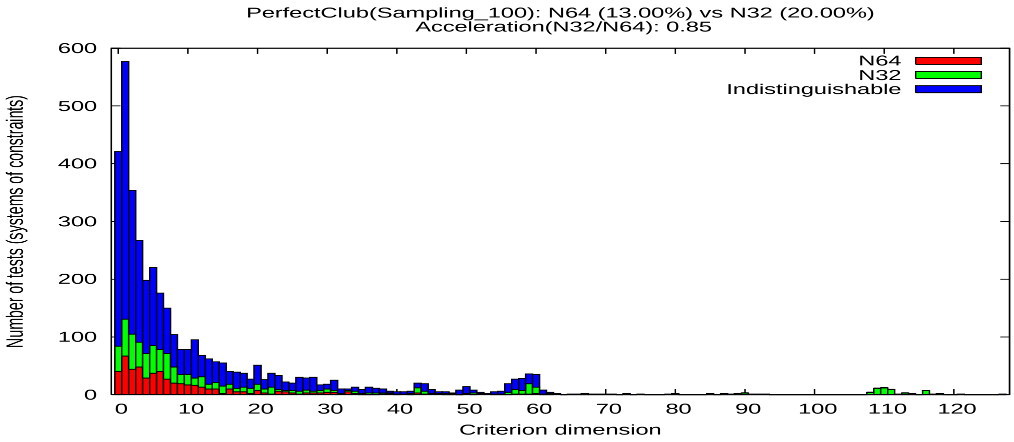


Figure 5.29: PerfectClub: Dimension N64-bit vs N32-bit in sampled database

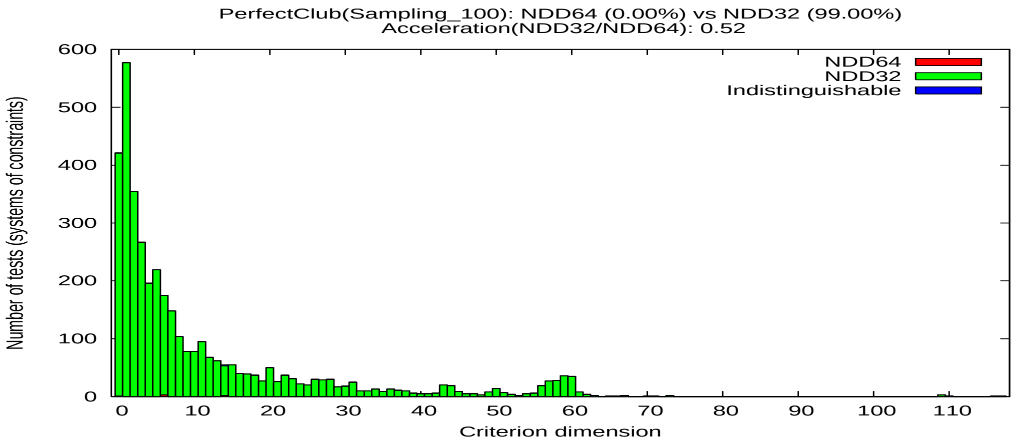


Figure 5.30: PerfectClub: Dimension NDD64-bit vs NDD32-bit in sampled database

	PerfectClub		
timeout = 2 minutes	**#overflows**	**#timeouts**	**#operations**
N64-bit	0	0	3894
N32-bit	0	0	3894
NDD64-bit	14	9	3894
NDD32-bit	74	5	3894

Table 5.19: PerfectClub: Numbers of exceptions in sampled database

rithm. Then we compare the numbers of exceptions between 32-bit and 64-bit versions of these implementations. In Figure 5.29 and Figure 5.30, we count differences of run times which varies from 0.84 to 1.07, thus the sacrifice in execution time for using $64-bit$ (higher precision) instead of $32-bit$ is not very important.

Table 5.19 shows that for the set of PerfectClub sampled database and the NDD implementation, numbers of exceptions are fewer using the higher precision. We notice that the number of overflows in 32-bit computation is much higher than in 64-bit (74 to 14 exceptions), compared to the difference of timeout exceptions (9 to 5 exceptions). Meanwhile, the direct constraint manipulation approach shows no difference in exceptions.

5.4 Conclusion

This conclusion consists of two main points:

- Given the constraint system representation, N64 is better than NDD64 in term of run time performance and number of exceptions;
- The 32-bit is preferable to 64-bit computation, for our minimization databases (for the timeout exceptions and a slightly shorter run time).

6 Results for Dual Conversion

We have several implementations for the dual conversion operator, which are mostly based on the Chernikova algorithm (see Chapter 4, Section 6). The conversion is dual, thus we focus only in converting a constraint system to its generating system. Since we do not have databases for the dual conversion operator, we have chosen to use the convex hull sampled databases, since this operator is strongly based on the dual conversion operator (see Section 2.8 for POLYBENCH's polyhedral databases).

We experienced problems[22] in adapting the LRS library (see Chapter 3, Section 3, page 53), to the POLYBENCH framework, thus the results for LRS are not yet available. New POLKA (see Chapter 3, Section 3, page 54) is not chosen since its convex hull implementation is strongly related to the dual conversion operator, as explained in Chapter 4. Therefore, in this section, we only present the comparison between two implementations: C^3 dual conversion that use POLYLIB implementation (see Chapter 3, Section 3, page 52) and CDD (see Chapter 3, Section 3, page 52).

6.1 C^3 Dual Conversion versus CDD, 64-bit

We now compare the implementation of Chernikova algorithm for 64-bit in C^3, denoted C3DD64, and the C-implementation using the double description method of the CDD library for 64-bit, denoted CDD64 (see Chapter 3, Section 3, page 52). Here we ignore the fact that CDD64 uses C-built-in double floating point or GNU multi-precision rational library which are both faster than C3DD64's integer computation.

In Figure 5.32, the histogram shows only green zones [23] which mean that CDD64 is always faster than C3DD64. In fact, CDD64 is faster in 100 percent of all tests. Finally, total run times accumulated for all tests are compared: we have CDD64 is faster than C3DD64 with a ratio of $0,14 : 1$, which means approximately 7 times faster, for the SPEC95 convex hull sampled database.

The histogram in Figure 5.31, shows green zones indicating the cases where CDD64 is faster than C3DD64 and red zones corresponding to the opposite cases. They are well separated by the dimension 80, from which a time-run-based heuristic can be constructed as follows: We use CDD64 for constraint systems with dimensions lower than 80 and C3DD64 for those with higher dimensions. We however notice that CDD64 is faster in 98 percent of all tests, thus this heuristic does not seem to be necessary according to the total run times accumulated. We have CDD64 is faster than C3DD64 with a ratio of $0,30 : 1$, which means approximately 3 times faster, for the PerfectClub convex hull sampled database.

The conclusion for run times of these two algorithms is that CDD64 is faster than C3DD64, and we have a good example of a heuristics which can be based on the exper-

[22] The very poor performance of our LRS conversion suggests that bugs might be present.

[23] See 3.1 for more explication of these histograms.

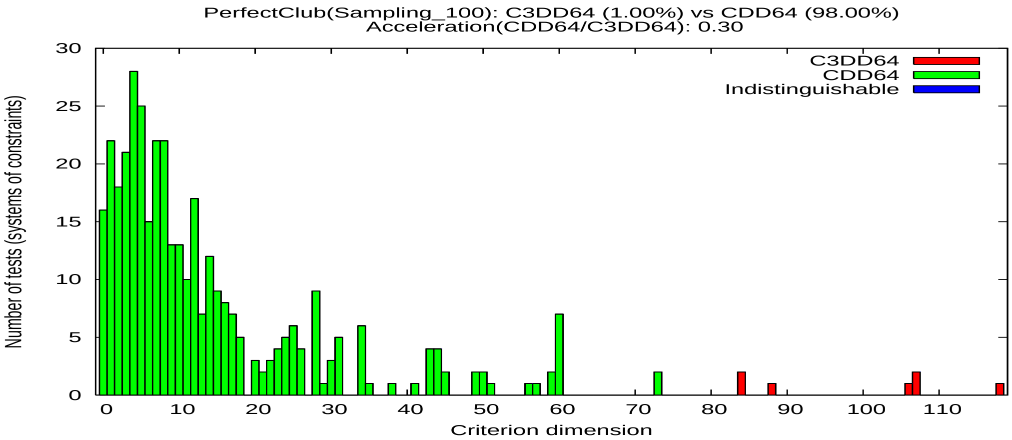


Figure 5.31: PerfectClub: Dimension C3DD64 vs CDD64 in sampled database

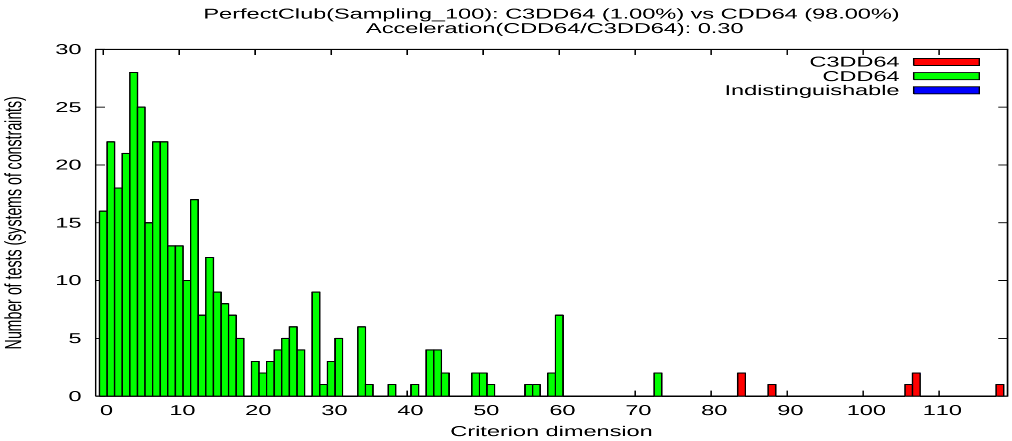


Figure 5.32: SPEC95: Dimension C3DD64 vs CDD64 in sampled database

	SPEC95		
timeout = 2 minutes	**#overflows**	**#timeouts**	**#operations**
C3DD64-bit	1	0	485
CDD64-bit	0*	0	485

Table 5.20: SPEC95: Numbers of exceptions in sampled database

imental results. If we do not consider the exception problem, we can obtain the parallel algorithm's performance with this heuristics that is much easier to implement.

6.2 Overflow and Timeout Exceptions: 64-bit

We have to emphasize here once more, that CDD does not detect overflows (see Chapter 3, Section 3, page 52). However, since POLYBENCH compares the resulting constraint systems of the two implementations, we can deduce the numbers of exceptions raised: if CDD64's output is *correct*, compared to C3DD64's output, then CDD64 has no overflow, as shown in Table 5.20. The numbers of exceptions come from C3DD64 and CDD64's executions on SPEC95 convex hull sampled database.

We notice that this example reveals a difference between the two implementations: an overflow exception from C3DD64. However, the convex hull sampled databases only contain fewer than 500 tests (PerfectClub's 393 tests do not show any difference in numbers of exceptions), thus we can conclude that the difference is not significant.

6.3 Arithmetic Precision: 64-bit versus 32-bit

Figure 5.33 illustrates the comparisons between C3DD64 and C3DD32 for PerfectClub convex hull sampled database. C3DD32 is faster than C3DD64 for all tests with two times faster for total run time.

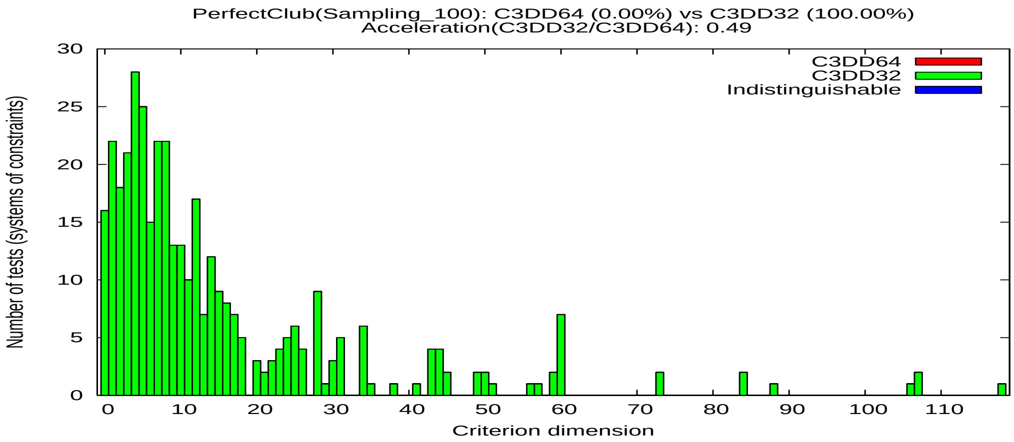


Figure 5.33: PerfectClub: Dimension C3DD64-bit vs C3DD32-bit in sampled database

	PerfectClub		
timeout = 2 minutes	**#overflows**	**#timeouts**	**#operations**
C3DD64-bit	0	0	393
C3DD32-bit	0	0	393
CDD64-bit	0	0	393
CDD32-bit	0	0	393

Table 5.21: PerfectClub: Numbers of exceptions in sampled database

As for the CDD implementation, we only have a little difference between CDD64 and CDD32, as demonstrated by Figure 5.34, for PerfectClub convex hull sampled database. There are cases where CDD64 is faster, but we do not know why.

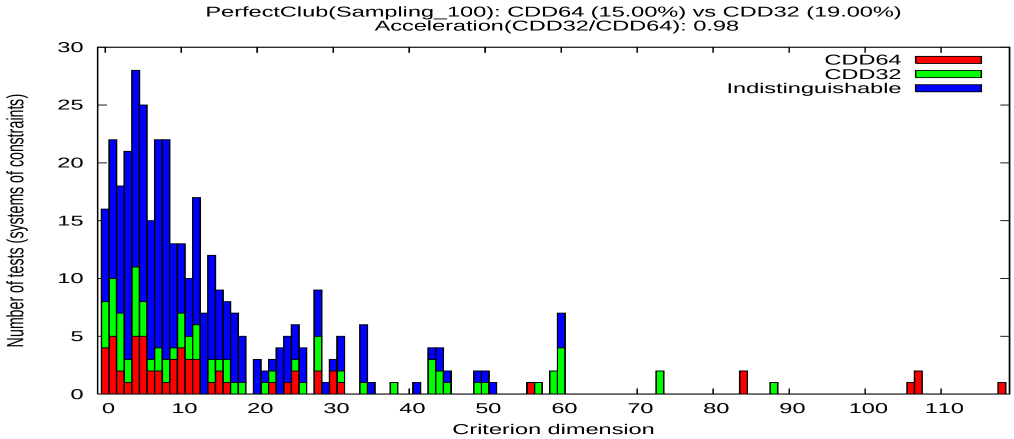


Figure 5.34: PerfectClub: Dimension CDD64-bit vs CDD32-bit in sampled database

Table 5.21 shows that the C3DD and CDD implementations yield no exceptions with PerfectClub convex hull sampled database. There are as well no exceptions for 485 tests in SPEC95 convex hull database.

6.4 Conclusion

Two points are noticeable:

- CDD64 (rational computing) is better than C3DD64 (integer computing) in term of run time performance and number of exceptions;
- The choice between 32-bit and 64-bit computation is not important for our convex hull sampled databases.

7 Results for Convex Hull

In this section, we compare three implementations for the convex hull operator. The first implementation, called C^3 's Partial Factorization, uses the POLYLIB implementation and the partial factorization described in Section 10.2.1, Chapter 4. The second implementation is the POLYLIB convex hull function, and the third one is the New POLKA convex hull function. (see Chapter 3, Section 3, page 54 and Section 3, page 54 for descriptions of POLYLIB and New POLKA libraries).

Convex hull sampled databases problem: The convex hull polyhedral databases described in Section 2.8 contain fewer tests than the other databases, thus sometimes small database histograms like Figure 5.35 do not make a good distribution. Exceptionally, the comparisons between the partial factorization and New POLKA only counts three tests in Figure 5.36, hence their evaluation is meaningless [24]. Therefore the full databases are used instead, though it requires much longer execution times since convex hull is an expensive operator.

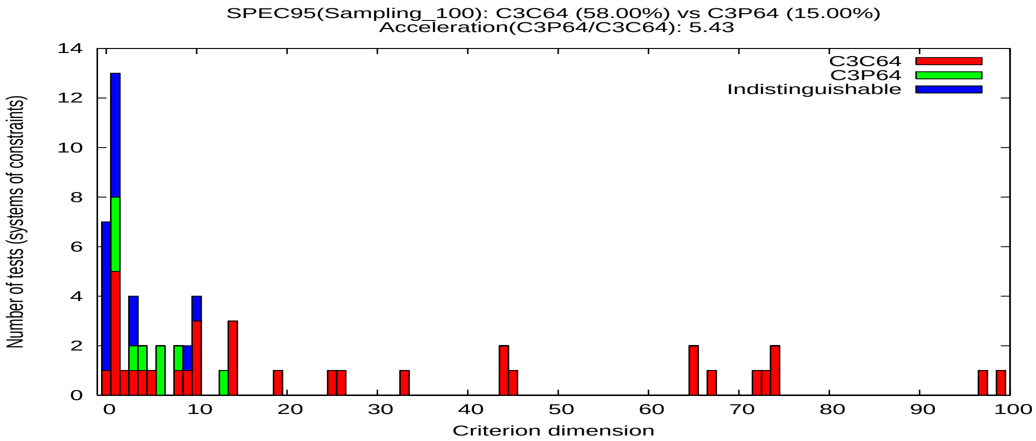


Figure 5.35: SPEC95: Dimension C3C64 vs C3P64 in sampled database - too few tests

7.1 C^3 Partial Factorization versus POLYLIB, 64-bit

We now compare C^3 convex hull implementation using partial factorization for 64-bit, denoted C3C64, and POLYLIB's convex hull implementation for 64-bit, denoted C3P64, in order to see whether the partial factorization, which can be seen as a pre-processing step, improves the convex hull computation or not.

In Figure 5.37, the histogram shows green zones meaning the cases where C3P64 is faster than C3C64 and red zones meaning the opposite cases. The blue zones mean we

[24]In our method, if one of the two considered implementations raises an exception for a test, the run times of the test will be ignored.

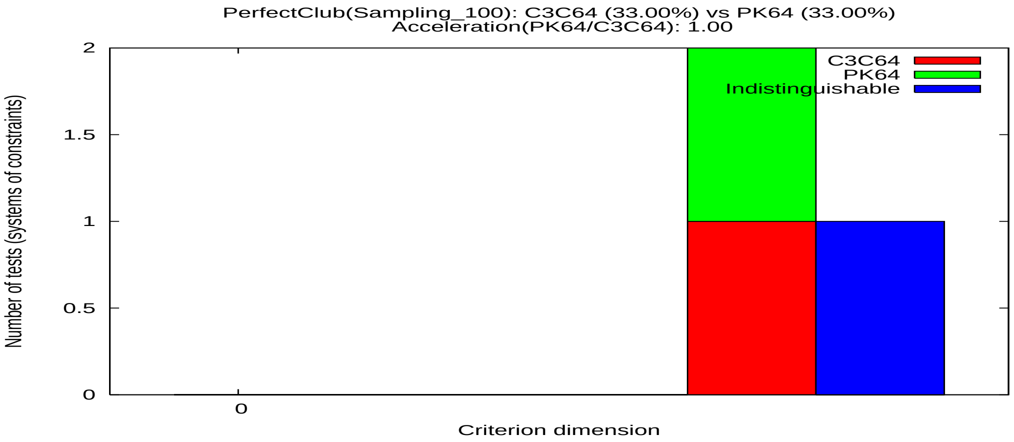


Figure 5.36: PerfectClub: Dimension C3C64 vs PK64 in sampled database - too few tests

cannot distinguish the difference because of the time resolution [25]. We notice that C3C64 is faster in 67 percent of all tests, whereas C3P64 is faster in only 7 percent of all tests. But, the total execution time ratio for this database indicates that C3P64 is faster than C3C64 with a ratio of $0,78:1$, which means approximately $1,25$ times faster. This means in some cases of PerfectClub databases, the partial factorization does not work very well.

On the contrary, Figure 5.38 presents a much better performance of the partial factorization C3C64, with SPEC95 databases. Indeed, C3C64 is faster in 63 percent of all tests, whereas C3P64 is faster in only 7 percent of all tests, and most important, C3C64 is 5 times faster than C3P64 by the total accumulated run times.

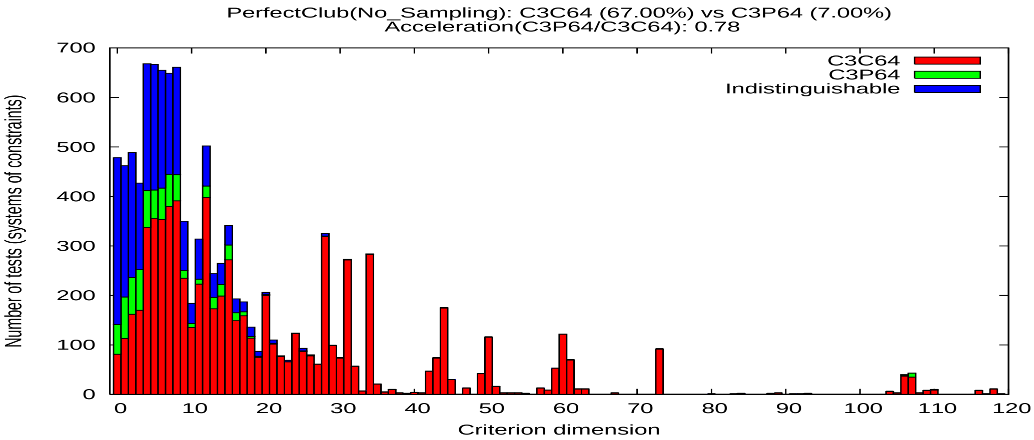


Figure 5.37: PerfectClub: Dimension C3C64 vs C3P64 in full database

A Sampling Problem for PerfectClub Convex Hull Database: An interesting remark on the results regarding our sampling implementation is that, in this unique case,

[25]See 3.1 for more explication of these histograms.

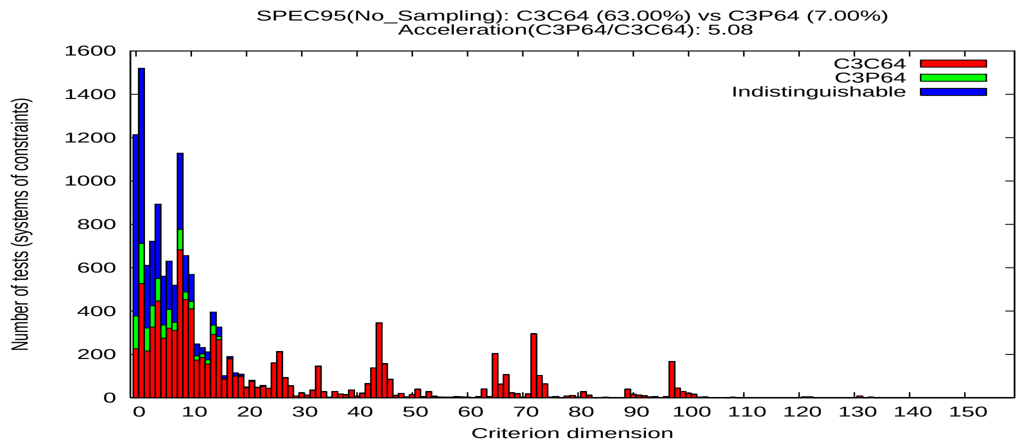


Figure 5.38: SPEC95: Dimension C3C64 vs C3P64 in full database

we have a small incoherence in conclusion for the PerfectClub sampled database and its full database: Figure 5.39 shows that C3C64 is about 3 times faster than C3P64, concerning the total run time. It is in fact the contrary to the result obtained with the full database.

This means, in our sampling of interval 100 on the PerfectClub convex hull database, where we randomly take 1 test from every 100 tests, a few tests that are not suitable for the partial factorization have been eliminated. We have not investigated this problem yet.

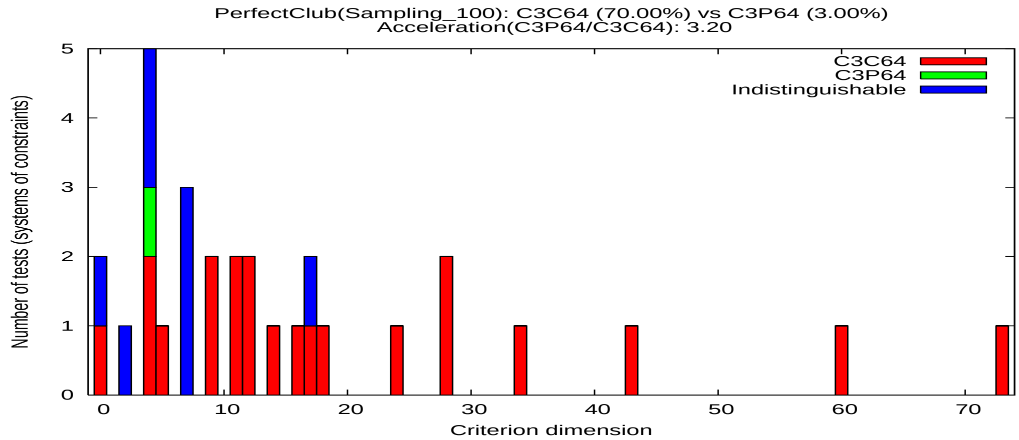


Figure 5.39: PerfectClub: Dimension C3C64 vs C3P64 in sampled database - too few tests

Our conclusion for timing performance of these two algorithms is that C3C64 is faster than C3P64, with the convex hull full databases.

7.2 C^3 Partial Factorization versus New POLKA, 64-bit

The comparison between C^3 convex hull implementation using partial factorization for 64-bit, denoted C3C64, and New POLKA's convex hull implementation for 64-bit, denoted PK64, is not totally satisfying for three reasons.

Firstly, though it seems to us that our conversion to POLYBENCH's format of New POLKA is correct, the number of exceptions raised by New POLKA is too important. When we reduce the maximal memory space allowed [26] in New POLKA with respect to those defined in POLYLIB, hence in C^3, we have far fewer exceptions.

Secondly, since New POLKA and POLYLIB both implement the same Chernikova algorithm, and since our tests are one-test-at-a-time, the memory strategy implemented in New POLKA has no effect and the performances are expected to be alike [27]. We are then interested in the Partial Factorization, which is in fact a pre-process step.

Thirdly, New POLKA's initialization and POLYBENCH's internal format conversion seem to be too expensive.

In Figure 5.40 and Figure 5.41, we can see the same result that C3C64 is faster than PK64: C3C64 in 35 and 33 percent of all tests, whereas PK64 is faster in 22 and 21 percent of all tests; C3C64 is approximately $1,5$ time faster than PK64 with respect to the total execution time ($1 : 1,48$ and $1 : 1,43$, respectively).

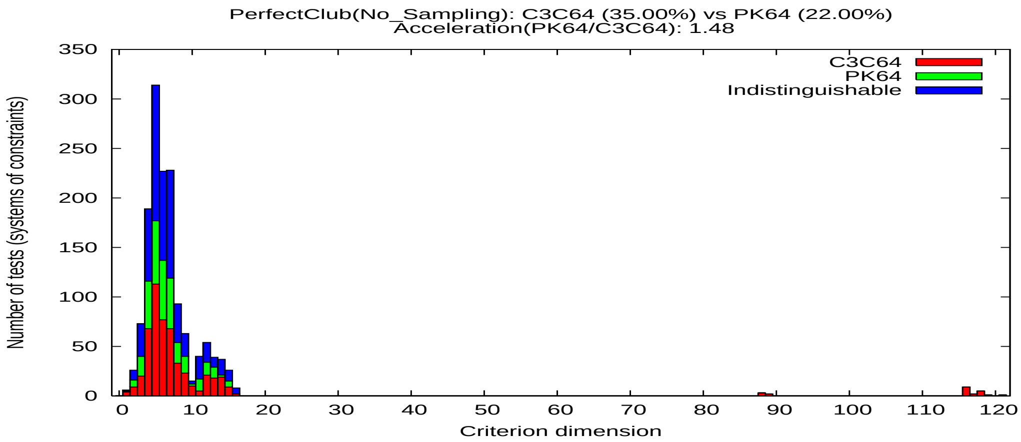


Figure 5.40: PerfectClub: Dimension C3C64 vs PK64 in full database

We remark that for the SPEC95 convex hull database, in comparison with the previous section, the number of tests is far fewer, which means that New POLKA's implementation raises many exceptions. We study this in the next section.

7.3 Overflow and Timeout Exceptions: 64-bit

The numbers of exceptions presented in Table 5.22 come from C3C64, C3P64 and PK64's executions on SPEC95 convex hull database. We can clearly see that there are too many timeout exceptions from New POLKA, and that the partial factorization not only improves the execution time but also reduces (in this case eliminates) the overflow exceptions from POLYLIB's original implementation of Chernikova algorithm. We obtained similar results

[26] MAX_NB_RAYS from 20000 to 5000.

[27] It was verified in our early experiments but not yet in POLYBENCH.

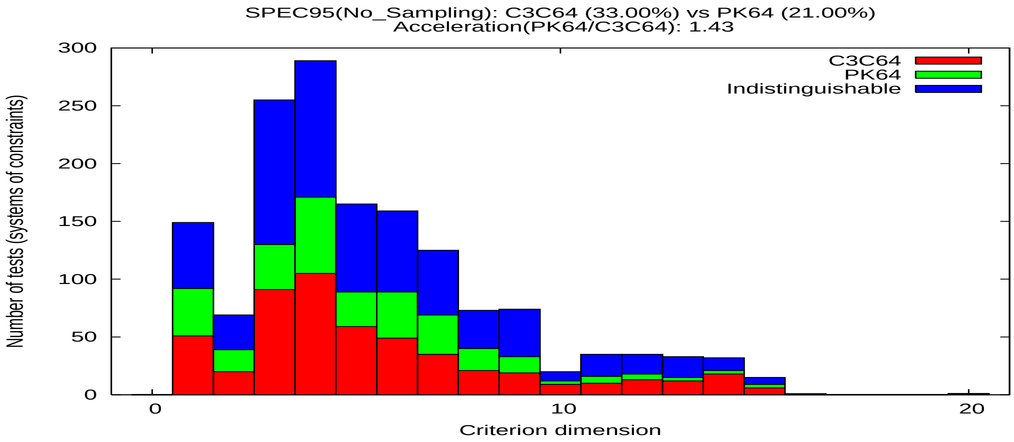


Figure 5.41: SPEC95: Dimension C3C64 vs PK64 in full database

	SPEC95		
timeout = 2 minutes	**#overflows**	**#timeouts**	**#operations**
C3C64-bit	0	0	24151
C3P64-bit	512	0	24151
PK64-bit	0	1270	24151

Table 5.22: SPEC95: Numbers of exceptions in full database

from PerfectClub's 19683 tests.

7.4 Arithmetic Precision: 64-bit versus 32-bit

Since C3C64 is clearly the winner, in this section we only compare it with its 32-bit version, denoted C3C32. Figure 5.42 illustrates these comparisons for PerfectClub convex hull database. C3C32 is faster than C3C64 for 37 percent of all tests with three times faster for total run time of all tests, whereas C3C64 is faster for only 8 percent of tests. SPEC95 convex hull database gives us similar results.

Table 5.23 shows that for the set of SPEC95 convex hull database, the C^3 partial factorization implementations, there are a few differences (0 to 32 exceptions with 24151 tests). For the PerfectClub, there are no differences with 19683 tests. Our conclusion for this part is that 64-bit computation is preferable to 32-bit computation.

	SPEC95		
timeout = 2 minutes	**#overflows**	**#timeouts**	**#operations**
C3C64-bit	0	0	24151
C3C32-bit	32	0	24151

Table 5.23: SPEC95: Numbers of exceptions in full database

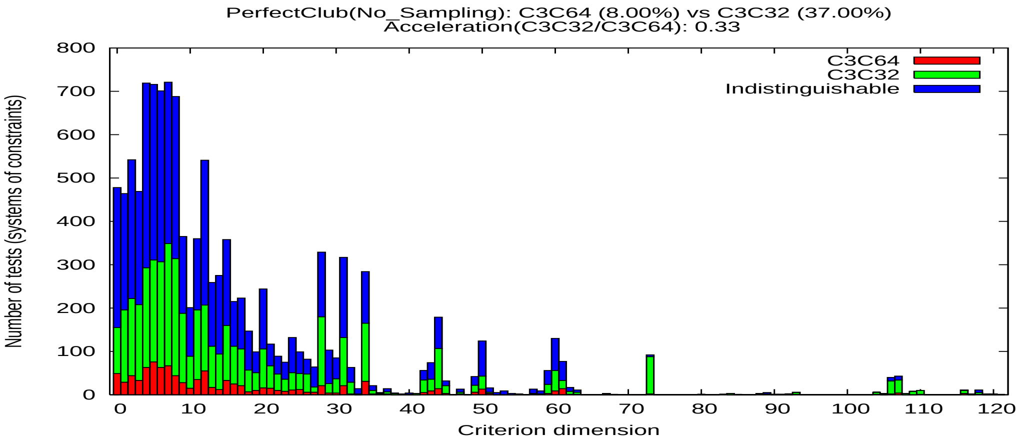


Figure 5.42: PerfectClub: Dimension C3C64-bit vs C3C32-bit in full database

7.5 Conclusion

Four points are relevant:

- The Partial Factorization designed and implemented by Corinne ANCOURT and Fabien COELHO proves to be an improvement for implementations of Chernikova algorithm, in term of run time performance and number of exceptions;
- Existence of some tests that are not suitable for the Partial Factorization, therefore the incoherence of PerfectClub sampled and full database (see the sampling problem in Page 172);
- 64-bit is preferable to 32-bit computation for this set of tests;
- We plan to work on the conversion of New POLKA's convex hull implementation for POLYBENCH.

8 Conclusion

In this chapter, we have presented the first large scale and real life experimental evaluations of key polyhedral operators. These evaluations require a lot of work because of the diversity of implementations and because of numerous missing implementations had to be coded by the author.

Our benchmark system is quite different from other benchmarks in that, besides the performance, the stability analyses, i.e., analyses of ability to cope with computational problems, we have integrated polyhedral characteristics in our evaluations.

Moreover, since our set of tests is generated from PIPS [PIPb, IJT91], an static analyzer performing on standard benchmarks with large size applications (PerfectClub and SPEC95 benchmarks), the results provide richer information than the bibliographical experimental results.

Indeed, pre-existing evaluations are not satisfying for several reasons: our bibliographic study has revealed the fact that, it doesn't exist yet a mechanism to evaluate effectively these works, especially in the domain of program analysis and transformation with real-life examples. Conducted evaluations are based on at most one hundred problems, mostly theoretical, without analyses on quantity, on criteria, on exceptions (cases where algorithms fail because of resource limits), etc...

An example of existing comparisons can be found in [SOG96], where JANUS is compared with Omega test [PUG91], whose results have discovered only performance-related issues concerning the *nightmare* problem, due to the fact that the OMEGA tool include many overhead factors that cannot be reduced by the author of [SOG96].

There is also a set of tests provided by *CDD* [FUK] and *LRS* [Avi], then used by PPL developers[28] in order to evaluate the PPL library's performance, with respect to the other libraries. These evaluations are based on the vertex/facet enumeration problem with a set of less than two hundred hand-made inputs, which are supported by the libraries *POLYLIB* [LOE, Wil], *New POLKA* [JEAb, JEA00], the LINEAR C^3 [PIPa, CAI00], *CDD* [FUK] and *LRS* [Avi] (see Chapter 3 for these libraries). Although these inputs supply varying complexities for these libraries, they are far from satisfying, given the differences among applications.

Apart from clarifying whether CPU or memory efficiency or both are the intended measures of interest, our benchmark offer problem-related analyses (such as polyhedral size parameters, their origin), stability comparisons, incoherent results checking, precision of computing comparisons, etc... Those features are not available elsewhere.

The average execution time and the ability to deal with exceptions were of high interest. Furthermore, these results gave us an idea of the impact of choices made at higher levels, or the interaction in the sequence of operations. For instance, a suitable projection approximation can reduce the polyhedral size of several satisfiability tests afterward.

[28] Available on PPL's website.

The results obtained in the satisfiability part have proved to be useful, where an external algorithm, JANUS, was extended to 64-bit precision and integrated by me in our analysis and transformation tool PIPS, because of its demonstrated better performance.

The dual conversion or double description approach is interesting since this operator can be used to implement projection, minimization and, mostly, convex hull computation. We have implemented the satisfiability test, projection and minimization using this approach, in order to compare with C^3 approach.

The decomposition of polyhedra by Corinne ANCOURT and Fabien COELHO (see Chapter 4, Section 10.2.1), can be used in order to speed-up the computation as shown in the convex hull part. This technique, or the strongly related one in [Mer05], can be applied for the other operators, including the satisfiability test.

Our evaluations first confirm the excellence of JANUS compared to all other implementations and algorithms as suggested in [SOG96] for integer satisfiability.

They also contain new comparisons of implementations that were not considered before, such as the Fourier-Motzkin method, the minimization operator implementation using the direct constraint manipulation in C^3 library, etc.

Then, they provide information on the connection between polyhedral problems, which are inputs for polyhedral libraries, and static analyzer's analyses on programs.

Furthermore, our framework studies the direct impact when using the 64-bit instead of 32-bit computation in polyhedral libraries, which is not possible from previous work.

Our experimental results show unexpectedly that the best predictive criterion is the number of dimensions and not the number of constraints. No criterion more predictive has been found. In case no clear winner is obtained, new heuristics can be constructed from polyhedral characteristics as described in the dual conversion operator part.

We have observed an unexpected sensibility for experimental data sets, for example in the comparisons of number of exceptions between JANUS and C^3 Simplex.

We also have seen the limit interest for hard cases, where the filtered databases provide poor information, except for the satisfiability test.

Bug detection and non-regression testing can be used to detect, for example, the difference between integer and rational answer. Such differences have been detected between JANUS and C^3 Simplex, which is rarely present, using our benchmark.

Our system of benchmarking is of course not complete and there are many things left to be done. For example, there are no experimental results related to backup algorithms dealing with exceptions, and making approximations which are outlined in Chapter 4.

We intend to add into our evaluations more operators and more libraries. Designs and implementations of new algorithms such as approximate algorithms for polyhedral operators or improvements using the decomposition of polyhedra are of interest.

Indeed, the Cartesian factorization proposed by [HMPV03], as well as the decomposition by inclusion test (Chapter 4, Section 10.2.3) are prime candidates for evaluation.

About the POLYBENCH framework, we need to improve the sampling as well as the repeating rule for small execution time.

We also plan to add memory usage comparisons into our comparisons, and try out other criteria in order to use heuristic-based approach.

A memorization scheme to avoid several executions of the same algorithm, as well as the resume capability are to be implemented.

Chapter 6

Conclusion

Abstract domain libraries used in current analyzers are dealing with problems limiting the effectiveness of checking statically safety and security properties of programs written in different languages, and identifying and locating origins of failures. In this dissertation we have studied the problems raised from the applications of static program analyzers in the industrial context, and then described two approaches to those problems.

One of the two main goals of this dissertation is to help designing a common interface for abstract domain libraries used in five static analyzers PIPS [PIPb, IJT91], NBAC [NBA, JEA00], ASTREE [AST, BCC$^+$03], the OMEGA framework [Ome, PUG91] and CHINA [CHI, PAR, BRZaH02]. The other goal is to provide a case study with the polyhedra-based libraries: benchmarking available polyhedral implementations. These two goals aim at answering the two questions:

- There are several abstract domains, which one is appropriate in a given context?
- For each abstract domain, how to decide which implementations are appropriate?

For the first goal, our starting point is to analyze problems existing in a particular analyzer, named PIPS [PIPb, IJT91], when using abstract domain libraries, and then extend to other analyzers such as ASTREE [AST, BCC$^+$03] and NBAC [NBA, JEA00].

For the second goal, benchmarking polyhedral libraries is interesting since this most used abstract domain counts several different implementations. They are varying, with several algorithmic discoveries that make them very robust, and complicated to the point that without a benchmarking system, we cannot determine which one is the most efficient. Benchmarking helps deciding when and where to use which (appropriate) implementations. It also helps regression testing, bug detection, performance and stability evaluations, etc.

The problems are identified and handled at several levels in actual static program analysis projects. They can be computing limits such as execution time and memory space, or technology limits that harm the accuracy of analyses, such as algorithmic unavailability. All static analyzers using abstract domains suffer from the very same problems, thus they develop different techniques whose objectives are same as ours.

Nevertheless, our approaches are different and complete theirs: instead of developing new abstract domains to totally replace existing abstract domains (e.g. the work of ASTREE group), we propose using both old and new ones, given that each one has its own advantages and disadvantages; and, given the poor pre-existing evaluations, we decide to take a step further by constructing a complex framework for evaluations, then propose a case study with polyhedra-based implementations.

The Chapter 1 introduced the context of static analyses and identified the main problems for static analyzers. Then Chapter 2 went into more details with basic concepts and some examples, where some related books and articles that we deem important are cited. The next chapters described our work, divided into two parts.

The first part, Chapter 3 and Chapter 4, deals with an adaptive abstract domain, which in fact leads to the construction of a common interface. This interface tries to combine and use efficiently existing abstract domains implementations.

The second part, Chapter 4 and Chapter 5, describes a framework permitting evaluations of equivalent implementations for the polyhedral domain. This framework later can be used with other abstract domains. We notice here that these two approaches, though both are quite important, have yet been worked on before.

In addition, we have Appendix page 189 explaining some important algorithms which are briefly described in Chapter 4.

Common Interface: In respond to the first question mentioned above, we decide that we can indeed use all the abstract domains since each one has its own advantages and disadvantages at the same time.

We have analyzed existing interfaces of available implementations, in Chapter 3, with different degrees of interest: *POLYLIB* [LOE, Wil], *New POLKA* [JEAb, JEA00], the LINEAR C^3 [PIPa, CAI00], PPL [PAR, BRZaH02], OCTAGON [MIN, MIN01b] and OMEGA [Ome, PUG91] libraries.

Our proposition for a common interface for abstract set manipulation engines, called HQ interface and presented in Chapter 4, helps to identify the problems, and presents our very first solution. It consists of a prototype for a common interface, and practical issue-related documents that reveal different approaches in existing implementations, from which concrete decisions can be made.

Also in Chapter 4, we have introduced the state of the art of a related project, the APRON project, and compare it with our approach. In fact, the HQ interface was presented in APRON meetings, and helped starting deeper discussions on the subject. Our proposition is divided into two parts.

The first part directly addresses the interface with imperative signatures, which should describe what we need, how to present, introduce and expose it as clearly as possible. We discuss the name of operators, what this function does, why we need other versions of it, when we apply approximation, whether we should have a list of arguments instead of only one argument, the level of this function, etc. These problems are already complex, thus they should be separated from the second part.

The second part, which is as important as the first part, is where we discuss implementation issues such as how we handle exceptions, how we manage the memory, etc. In fact, there are many ways to deal with this kind of problems, so finally we just have to pick one that is the most appropriate. For example, from the initial HQ's signatures whose prototype is implemented in Java, we can use the JNI (Java Native Interface) tool to generates its C signatures. However, this approach is not satisfying since the generated code is not easy to understand, so we can consider building a set of rules for this conversion. Other problems such as memory management, destructive functions, destructive arguments are also documented in order to be decided later.

During APRON meetings, more problems were identified and discussed, and another prototype was constructed. Recent developments presented at the VMCAI 2005 and NSAD 2005 workshops show that current polyhedral libraries such as *POLYLIB* [LOE, Wil] and PPL [PAR, BRZaH02], as well as polyhedra-related libraries such as Octagon library, are being worked on their interfaces. Likewise, new abstract domains are introduced, e.g. in [FER05b], which may have an important impact on our common interface.

Some complex problems such as product of domains or the Presburger domain are not dealt with. At first, the implementation of the common interface will help the three static analyzers described in Chapter 3 to profit from those abstract domains. Then, given the compatibility among the libraries, other analyzers using abstract interpretation can use them, too.

Benchmarking Polyhedral Implementations: The second question mentioned above is reformulated as follows: we have considered several abstract domains, but did we have the best use of each domain? To answer this question, we have analyzed in detail the most common polyhedral operators in Chapter 4, and conducted several experiments with the most used polyhedral libraries, presented in Chapter 5.

Indeed, Chapter 4 presented an operator by operator view of the common polyhedral API, surveyed algorithms and existing implementations, and discussed problems concerning each operator. Problem of different operator names is clarified. Used convex polyhedra related definitions and the list of these implementations were introduced in Chapter 3.

In Section 6, we have presented the history of Chernikova related algorithms for the dual conversion (double description) operator.

Then, in Section 7, we have detailed four algorithms that are implemented for the test of satisfiability: the Fourier-Motzkin's and Simplex method for rational test that were already available in C^3 library, the JANUS for integer test that was ported to 64-bit by me, and the algorithm based on dual conversion operator, implemented by me.

In Section 8, we have introduced two main algorithms for the projection operator, one uses the dual conversion, implemented by me, and the other uses the Fourier-Motzkin elimination method.

Similar to the projection operator, the minimization operator, which has two main algorithms, is discussed in Section 9. One algorithm is based on dual conversion, implemented by me, and the other algorithm directly manipulates the polyhedron's H-representation form.

The convex hull operator is exposed along with three recent improvements: the partial factorization (designed and implemented by Corinne ANCOURT and Fabien COELHO, see Section 10.2.1), the Cartesian factorization [HMPV03] (see Section 10.2.2) and our decomposition using inclusion test, inspired by the Cartesian factorization (Section 10.2.3).

Other operators such as the intersection, difference, widening, narrowing, etc. were then briefly presented.

In this chapter, for each operator, differences among existing libraries at the interface level were analyzed, in order to study the possibility of an integration among nearly-equivalent works. Being mostly an experimental work, practical issues such as incompatibilities among polyhedral libraries are relevant.

Our contributions to improve some of these operators such as the port to 64-bit of JANUS and its integration in C^3 (Section 7.2.3, Page 98) are also presented in this chapter.

Propositions for improvement, (e.g. the decomposition using inclusion test, Section 10.2.3, Page 113) as well as properties (e.g. projection using double description Page 102) are introduced.

Backup algorithms for approximations to deal with polyhedral high complexity are briefly discussed.

The question of precision versus approximation is raised throughout the chapter, as well as computational issues like 32-bit, 64-bit, 128-bit or GNU multi-precision modes.

The experimental results are presented in Chapter 5. In this chapter, we introduced our POLYBENCH framework and its benchmarking results for polyhedral libraries.

In fact, it is the first large scale and real life experimental evaluations of key polyhedral operators. These evaluations require a lot of work because of the diversity of implementations and because of numerous missing implementations had to be coded by the author.

Our benchmark system is quite different from other benchmarks in that, besides the performance, the stability analyses, i.e., analyses of ability to cope with computational problems, we have integrated polyhedral characteristics in our evaluations.

Moreover, since our set of tests is generated from PIPS [PIPb, IJT91], a static analyzer performing on standard benchmarks with large size applications (PerfectClub and SPEC95 benchmarks), the results provide richer information than previous experimental work.

Indeed, pre-existing evaluations are not satisfying for several reasons: our bibliographic study has revealed the fact that, it doesn't exist yet a mechanism to evaluate effectively these works, especially in the domain of program analysis and transformation with real-life examples. Conducted evaluations are based on at most one hundred problems, mostly theoretical, without analyses on quantity, on criteria, on exceptions (cases where algorithms fail because of resource limits), etc...

An example of existing comparisons can be found in [SOG96], where JANUS is compared with Omega test [PUG91], whose results have discovered only performance-related issues concerning the *nightmare* problem, due to the fact that the OMEGA tool include many overhead factors that cannot be reduced by the author of [SOG96].

There is also a set of tests provided by *CDD* [FUK] and *LRS* [Avi], then used by PPL developers in order to evaluate the PPL library's performance [Ba], with respect to the other libraries. These evaluations are based on the vertex/facet enumeration problem with a set of less than two hundred hand-made inputs, which are supported by the libraries *POLYLIB* [LOE, Wil], *New POLKA* [JEAb, JEA00], the LINEAR C^3 [PIPa, CAI00], *CDD* [FUK] and *LRS* [Avi] (see Chapter 3 for these libraries). Although these inputs supply varying complexities for these libraries, they are far from satisfying, given the differences among applications.

In POLYBENCH framework, the average execution time and ability to deal with exceptions are the most interested factors. However, apart from clarifying whether CPU or memory efficiency or both are the intended measures of interest, our benchmark offer problem-related analyses (such as polyhedral size parameters, their origin), stability comparisons, incoherent results checking, precision of computing comparisons, etc... Those features are not available elsewhere.

Furthermore, these results gave us an idea of the impact of choices made at higher levels, or the interaction in the sequence of operations. For instance, a suitable projection approximation can reduce the polyhedral size of several satisfiability tests afterward.

The dual conversion or double description approach is interesting since this operator can be used to implement projection, minimization and, mostly, convex hull computations. We have implemented the satisfiability test, projection and minimization using this approach, in order to compare with C^3 approach.

The results obtained in the satisfiability part (Section 3) have proved to be useful, where an external algorithm, JANUS, was extended to 64-bit precision and integrated by me in our analysis and transformation tool PIPS, because of its demonstrated better performance. Our evaluations first confirm the excellence of JANUS compared to all other implementations and algorithms as suggested in [SOG96] for integer satisfiability.

It is important to note that, JANUS has better performance than double description based method, which is largely used by *POLYLIB* [LOE, Wil], *New POLKA* [JEAb, JEA00].

The decomposition of polyhedra by Corinne ANCOURT and Fabien COELHO (see Chapter 4, Section 10.2.1), can be used in order to speed-up the computation as shown in the convex hull part (Section 7). This technique, or the strongly related one in [Mer05], can be applied for the other operators, including the satisfiability test.

Our evaluations also contain new comparisons of implementations that were not considered before, such as the Fourier-Motzkin method, the minimization operator implementation using the direct constraint manipulation in C^3 library, etc.

They also provide information on the connection between polyhedral problems, which are inputs for polyhedral libraries, and static analyzer analyses on programs.

Furthermore, our framework studies the direct impact when using the 64-bit instead of 32-bit computation in polyhedral libraries, which is not possible from previous work.

Our experimental results show unexpectedly that the best predictive criterion is the number of dimensions and not the number of constraints. No criterion more predictive has been found. In case no clear winner is obtained, new heuristics can be constructed from polyhedral characteristics as described in the dual conversion operator part.

We have observed an unexpected sensibility for experimental data sets, for example in the comparisons of number of exceptions between JANUS and C^3 Simplex (Section 3).

We also have seen the limit interest for hard cases, where the filtered databases provide poor information, except for the satisfiability test.

Bug detection and non-regression testing can be used to detect, for example, the difference between integer and rational answer. Such differences have been detected between JANUS and C^3 Simplex, which is rarely present, using our benchmark.

In conclusion, the main results are recapitulated as follows:

- First experimental evaluations for polyhedral operators at large scale and for real-life applications;
- No absolute winner: need of new heuristics;
- Unexpected sensibility for different test sets;
- Error detection and non-regression testing.

Our contributions in this dissertation are the evaluations that are published in NSAD 2005 workshop and HQ, the generic multi-domain, multi-implementation interface, which has been represented and used in the APRON project.

Since a common interface is a collective work, our proposition has only been served to break the surface of the whole problem, which requires more time and work.

We intend to continue working on the HQ interface, where the compatibility problems are documented, given that APRON does not supply this kind of documentation. Some adaptations from the APRON's prototype are also encouraged.

We also plan to complete our HQ interface and compare it to APRON approach, knowing that there are some key differences: while APRON members prefer to minimize the number of changes required to adopt the new interface, we are free of that obligation; HQ approach is in fact PIPS-oriented, and APRON approach needs to take into account other analyzers. Thus we can have more clarity and possible simplification of the problems.

Our system of benchmarking is of course not complete and there are many things left to be done. For example, there are no experimental results related to backup algorithms dealing with exceptions, and making approximations which are outlined in Chapter 4.

We intend to add into our evaluations more operators and more libraries. Designs and implementations of new algorithms such as approximate algorithms for polyhedral operators or improvements using the decomposition of polyhedra are of interest.

Indeed, the Cartesian factorization proposed by [HMPV03], as well as the decomposition by inclusion test (Chapter 4, Section 10.2.3) are prime candidates for evaluation.

About the POLYBENCH framework, we need to improve the sampling as well as the repeating rule for small execution times.

We also plan to add memory usage comparisons into our comparisons, and try out other criteria in order to use heuristic-based approach.

A memorization scheme to avoid several executions of the same algorithm on the same arguments, as well as the resume capability, are to be implemented.

Chapter 7

Appendix algorithms

In this appendix, we present some bibliographical research on important algorithms that are mentioned in our dissertation. We also describe our algorithms of the approximation for the projection operator.

1 Dual Conversion

1.1 Available Algorithms

The dual conversion is fundamental and has been extensively studied in computational geometry under the name *the problem of convex hull of a set of points*, which is the smallest convex set that contains all the points. The result is a generating system of the set. Its applications spread over an impressive number of fields: analysis of spectrometry data, file searching, cluster analysis, collision detection, crystallography, pattern recognition, image processing, numerical integration, statistics, metallurgy, cartography, etc., and of course, program analysis. Other problems that can be reduced to the convex hull are the Delaunay triangulation and Voronoi diagrams.

Classification - In essence, there are only two main classes of algorithms for the problem: *pivoting* and *incremental* algorithms. The pivoting algorithms implement the ideal of simplex like in linear programming, where we start from a vertex, go to another adjacent vertex, then by some way travel all the set of vertex. They are also called *graph traversal* algorithms, if we consider each vertex is a node of a graph, or *gift-wrapping* algorithms, in the context of facet enumeration going from one facet to a neighboring can be viewed as rotating a supporting hyper-plane about the common ridge, in analogy to a 3-dimensional physical realization.

The incremental algorithms compute the vertex description by intersecting the defining half-spaces sequentially. An initial simplex is constructed from a subset of $n + 1$ half-spaces and its vertices. Additional half-spaces are introduced sequentially and the vertex description are updated at each stage. Essentially such an update amounts to identifying and removing all vertices that are not contained in the new half-space, introducing new vertices for all intersections between edges and the bounding hyper-plane of the new half-space, and generating the new edges between these new vertices.

Besides, depending on some specific points, one also can divide algorithms by class of *deterministic* algorithms, or class of *output sensitive* and *randomized* algorithms.

1.1.1 Algorithms in R^2

To walk through the list of important algorithms in R^2, we firstly cite the deterministic algorithms for a finite set of n points, while being conscious that adaptation for infinite set, with rays and lines may exist. There is a strong analogy between the convex hull problem and the sorting problem: Graham's algorithm uses sorting explicitly [Gra72]; Jarvis's is similar to selection sort [Jar73]; the divide-and-conquer algorithms are the geometric equivalent of merge sort; the dynamic algorithms are similar to the insertion sort; quick-hull is analogous to quick-sort, [BDH96]). The quick-hull can be used for R^2, R^3 as well as higher dimensions. The papers [BAY, PS85] contain surveys of these algorithms.

Graham's Scan , [Gra72], takes time $O(n \ log \ n)$ and uses $O(n)$ space, for n is the number of points as input. It cannot be generalized to higher dimensions than two, because the theorem it is based on holds only in the plane R^2: *Consecutive vertices of a convex polygon occur in sorted angular order about any interior point*, where a *polygon* is a region of the plane R^2 bounded by a cycle of line segments, called *edges*, joint end-to-end in a cycle. Points where two successive edges meet are *vertices*. Graham's scan starts with one point p known to be on the convex hull (say the point with lowest horizontal coordinate) and sorts the rest of the points in angular order around p. Starting with a hull consisting of p and the point v with the smallest angle, we proceed counterclockwise around v adding vertices. If the angle formed by the new point and the last hull edge is less than 180 degrees, we add this new point to the hull. If the angle formed by the new point and the last "hull" edge is greater than 180 degrees, then a chain of vertices starting from the last hull edge must be deleted to maintain convexity. The total time is $O(n \ log \ n)$, since the bottleneck is sorting the vertices around v.

Jarvis' March , [Jar73], is based on the fact that, *the line segment defined by two vertices of a set is an edge of the convex hull if and only if all the other vertices of the set lie on the same side of the edge, or on the edge*. This algorithm is applied in two dimensions space R^2, whose running time depends on the output size, $O(nh)$ where h is the number of points on the convex hull, i.e., number of vertices. It is then called an *output sensitive* algorithm. The worst case complexity however is $O(n^2)$. In fact, it is a simple version of the aforementioned gift-wrapping algorithm in two dimensions, since each facet becomes an edge, each edge becomes a vertex of the polygon, and the breadth-first search simply walks around the hull in a clockwise or counterclockwise order.

Quick Hull , [BDH96], starts by finding the line passing through the points l and r in R^2 with smallest and largest abscissae. This line divides the points into two subsets S_1 above the line and S_2 below the line. The concatenation of the polygonal chains of S_1 and S_2 gives the convex hull needed. The algorithm find the point $o \in S_1$ that has the maximum area among all triangles formed by a point $p \in S_1$ and l, r. In case of ties, the left-most point is chosen, and o belongs to the convex hull. All the points inside the triangle olr can be eliminated. S_1 is then partitioned into subsets S_1^1 and S_1^2, where S_1^1 contains the points to the left of the ray $\overrightarrow{lo}$ and to the left of the ray $\overrightarrow{ro}$, and S_1^2 contains the points to the right of the ray $\overrightarrow{lo}$ and to the right of the ray $\overrightarrow{ro}$. The Quick-hull is then recursively applied. It has $O(n^2)$ worst case running time, and if the partition is approximately balanced, it runs in $O(n\ log\ n)$. It can be used for general dimensions, however the actual implementation of Quick-hull [Qhu] suggests in particular practical dimensions from 2 to 8.

Divide and Conquer aims to avoid the worst case running time of $O(n^2)$ by dividing the set of points S in R^2 into two subsets S_1 and S_2, each containing half of the points to obtain almost balanced partitions. The complexity obtained is then $O(n\ log\ n)$.

Dynamic Algorithms or *on-line* algorithms for convex hull maintenance are data structures that permit inserting and deleting arbitrary points while always representing the current convex hull. The first such dynamic data structure supported insertions and deletions in $O(log\ n)$ time in R^2 is described in [OvL80].

Randomized Algorithms Go to Section 1.1.3 Page 191.

1.1.2 Algorithms in R^3

Generalized Divide and Conquer - Preparata and Hong, in [PS85], devised an algorithm for three dimensional case based on Divide and Conquer algorithm in R^2, which runs in time $O(n\ log\ n)$.

Randomized Algorithms See in Section 1.1.3 Page 191.

1.1.3 Algorithms in General Dimensions

Gift Wrapping, proposed by Chand and Kapur ([CK70]), is a basic pivoting algorithm for constructing higher-dimensional convex hulls. In order to illustrate the idea, we observe a simple case with a three-dimensional convex polyhedron that is composed of facets of two dimensions, which are connected by one-dimensional lines, or edges. Each edge joins exactly two facets together. Gift-wrapping starts by finding an initial facet associated with the *lowest* vertex and then conducting a breadth-first search from this facet to discover new, additional facets. Each edge e defining the boundary of a facet must be shared with one other facet, so by running through each of the n points we can identify which point defines the next facet with e.

Essentially, we *wrap* the points one facet at a time by bending the wrapping paper around an edge until it hits the first point. The key to efficiency is making sure that each edge is explored only once. Implemented properly in d dimensions, gift-wrapping takes $O(n\phi_{d-1} + \phi_{d-2}log\phi_{d-2})$, where ϕ_{d-1} is the number of facets and ϕ_{d-2} is the number of edges in the convex hull. Thus gift-wrapping can be very efficient when there is only a constant number of facets on the hull. However, the worst case complexity is $O(n^{\lfloor d/2 \rfloor +1})$. A special case of this technique is Jarvis's March in R^2.

Beneath and Beyond Algorithm is an incremental approach that constructs the convex hull by adding one point at a time. It has the worst case complexity $O(n^{\lfloor (d+1)/2 \rfloor})$ and is optimal for even dimensions (see [Sei81, Ede87]). Chazelle in [Cha93] improved the running time to $O(n\ log\ n + n^{d/2})$ by *de-randomizing* the Clarkson and Shor's randomized incremental algorithm, and the obtained algorithm is worst case optimal in all dimensions.

Kirkpatrick and Seidel then constructed an output sensitive algorithm which runs in $O(n\ log\ h)$ time ([KS86]), where h is the number of points in the convex hull. It constructs separately the upper and the lower hulls. For the upper hull, the points in $S \in R^2$ are split into two equal size subsets, separated by a vertical line. Then it computes the merge of the two subsets (the *upper bridge*), before recursively solving the sub problems. The points of S are paired into couples (p_a, p_b) such that $x_a < x_b$. These ordered pairs define $\lfloor n/2 \rfloor$ straight lines. The median m of the set of lines' slopes is found, then a straight line of slope m that contains at least a point of S but has no point of S above it is determined. This straight line either contains the *upper bridge*, or can be used to remove some internal points in S, until the *upper bridge* is found. Actually, the bridge can be defined as the solution to a linear programming problem that can be solved in linear time.

Randomized Algorithms are firstly presented by Clarkson and Shor ([Cla88a, Cla88b, CS89]). These algorithms always produce the correct solution but their running time depends upon random choices made during the execution of the algorithm. Clarkson in [Cla88a] gave randomized incremental algorithms for constructing convex hulls in various dimensions which run in $O(n\ log\ n)$ expected time in R^2 and R^3. He also provided an $O(n\ log\ h)$ expected time algorithm in R^3 which uses Kirkpatrick and Seidel's $O(n\ log\ h)$ algorithm for finding convex hull in R^2, where h is the number of vertices.

Seidel in [Sei92] described a simpler randomized incremental algorithm for points in the plane R^2 which also runs in $O(n\ log\ n)$ expected time. A randomized variation of quick-hull of $O(n\ log\ h)$ time is introduced by Wenger in [Wen97], and a similar algorithm proposed by Shafer and Steiger in [SS93].

Chan and al. proposed a deterministic variation of quick-hull which runs in $O(n\ log\ h)$ time, and can be generalized to higher dimensions ([CSY95, CSY97]), where h is the number of vertices. This algorithm finds the median slope and is faster than the Kirkpatrick and Seidel's algorithm by a constant factor. The median finding is the most costly operation in both algorithms. Let cn be the time to find the median of n numbers, where c is a constant. Then Kirkpatrick and Seidel's algorithm spends $3cn\ log\ h$ time in the worst case, though Chan and al's algorithm spends approximately $1.2n\ log\ h$ time. In [Cha95], Chan gave an output sensitive convex hull algorithm based on ray shooting, which runs in optimal $O(n\ log\ h$ time in R^2 and R^3.

Bhattachrya and Sen offer in [BS97] an algorithm similar to Chan and al's, but the median finding step is replaced by an approximate median. The algorithm is a randomized one and runs in expected time $O(n\ log\ h)$, where h is the number of vertices. Unlike Wenger's analysis in [Wen97], the constant associated with the expected running time is quite small. In one strategy, the median element of a given set is replaced by a random element of the set. The algorithm terminates with probability $1 - O(h^{-1})$. In the other strategy, the median element is replaced by the exact median of some randomly selected elements. In this case, the algorithm terminates with probability $1 - O(2^{-\Omega(n^{1/4})})$.

Double Description Method, pioneered by Motzkin and al [TMT53], is an incremental algorithm that can be described briefly. Let's A^k be the sub-matrix of matrix A consisting of the first k rows of A. It is easy to find a generating matrix R^k for $P(A^k)$ when k is small, say $k = 1$. In the general iteration k, the algorithm constructs a double description pair (A^{k+1}, R^{k+1}) from the pair (A^k, R^k) already computed. This algorithm was rediscovered repeatedly and given different names, including several incremental algorithms [AB95].

All incremental algorithms can employ different insertion orders. One can distinguish between *static* insertion orders, which are determined from the inputs before the actual incremental algorithm is started, and *dynamic* insertion orders, where the next half-space (vertex) to be considered is a function of the current state. Typical static orderings are *min-index* (process in order as the input happens to be), *lexicographic* (process in lexicographic order of normalized coefficient vectors) and *random*. Typical dynamic orderings are *max-cutoff* and *min-cutoff* (as next half-space choose the one whose complement contains the most or least vertices of the current polyhedron). The order chosen is very important that leads to different running times of algorithms. Seidel's algorithm [Sei92] relies on the use of a lexicographic order, whereas Chazelle's [Cha93] is a dynamic insertion order.

One of the main obstacles to achieving polynomial complexity in size of the polyhedron is *degeneracy*. In the case of vertex enumeration, this means more than n facets contain a vertex and hence vertices do not have a unique basis. For facet enumeration, degeneracy dualizes to having more than n vertices contained in a facet. There are two general methods to deal with degeneracy for both pivoting and incremental algorithms.

The first method is to generalize the algorithm so that it generates the entire face lattice of the polyhedron (*lattice producing* approach). The second method employs perturbations in order to simulate non-degeneracy. Since in the case of facet enumeration, perturbing the input vertices results in *triangulating* the facets, we call it *triangulation producing*.

An implementation that deals with the double description problem, is CDD (which stands for C-implementation for Double Description method, [FUK]).

Chernikova's algorithm , ([Che64, Che65, Che68]), is basically the same method as Double Description method described by Motzkin and al ([TMT53]), the *beneath and beyond* method ([Sei81, Sei87, Ede87]), Clarkson and Shor's *randomized* algorithm and Chazelle's *de-randomized* algorithm. Le Verge [Ver92] proposed an optimization to detect redundant elements in Chernikova's algorithm. This improved algorithm is implemented in POLYLIB library (Chapter 3, Section 3, page 54), and used in C^3 library (Chapter 3, Section 3, page 52).

Lexicographical Reverse Search, developed by David Avis and Komei Fukuka ([AF92]), is a pivoting algorithm using lexicographic ratio test to resolve degeneracy. Suppose we have a system of m linear inequalities defining a d-dimensional polyhedron in R^d and a vertex of that polyhedron given by the indices of d inequalities whose bounding hyper-planes intersect at the vertex. These indices define a *co-basis* for the vertex. The complementary set of $m - d$ indices are called a *basis*.

For any given linear objective function, the simplex method generates a path between adjacent bases (or equivalently *co-bases*) which are those differing in one index. The path is terminated when a basis of a vertex maximizing this objective function is found. The path is found by pivoting, which involves interchanging one of the hyper-planes defining the current co-basis with one in the basis. The path chosen from the initial given basis depends on the pivot rule used, which must be finite to avoid cycling. If we look at the set of all such paths from all bases of the polyhedron, we get a spanning forest of the graph of adjacent bases of the polyhedron. The root of each subtree of the forest is a basis of an optimum vertex. The reverse search algorithm starts at each root and traces out its subtree in depth first order by reversing the pivot rule. This algorithm's implementation is called LRS (which stands for Lexicographical Reverse Search, [Avi]).

1.2 Approximations

When exception happens, we need backup algorithms that permit approximation of the result. However, the dual conversion operator is a specific one in the static program analysis using the polyhedral domain, since it is often used as a hidden operator. Precisely, this operator is used by the other operators such as the convex hull, projection, or minimization operators. Thus, approximations are currently made for the other polyhedral operators, but not itself. We do not know of any approximation for this operator.

2 Projection

In this section, we present the proof of Theorem 8.1 which is required for the implementation of the projection using the Dual Conversion. In fact, this operator is implemented in Double Description method-based libraries such as New POLKA, POLYLIB, PPL, but no such proof has been available.

2.1 Projection using Dual Conversion

The basis containing all the variables of the space dimension is supposed linearly independent. So, given a polyhedron P in D^n with basis $\{e_1, ..., e_n\}$, we have:

$$P = \{x|P(x), x \in R, x = a_1e_1 + ... + a_ne_n\}$$

Let $W = w_1, ...w_k$ be a subset of the basis, then the projection of P from D^n to W is the projection x' of all elements x in P from D^n to W such that:

$$\begin{aligned} x' &= projection_W(x) \\ &= \textstyle\sum_{i=1}^{k} < x, w_i > w_i \end{aligned}$$

Given Definition 3.2 (Chapter 3, Page 31), the dual conversion permits an easy implementation of projection operators, thanks to Theorem 2.1:

THEOREM 2.1

$$\begin{aligned} P' &= projection_W(P) \\ &= projection_W(v_1, \ldots, v_\alpha) + projection_W(r_1, \ldots, r_\beta) + projection_W(l_1, \ldots, r_\gamma) \end{aligned}$$

PROOF. □

2.2 Approximations

When exception happens, we need backup algorithms that permit approximation of the result. Unfortunately, this problem is not yet systematically treated, so a common way to do after an exception is to remove all the constraints that contain the dimension to project. This approach has advantages like simplicity, rapidity but it gives very low precision. When the variable to be projected appears in all the constraints of the system, the result is then an empty polyhedron.

We now propose some deterministic algorithms in projection approximation for Fourier Motzkin method. Let's call v number of variables, n number of constraints of the system of constraint sc. Then the matrix size of sc is vn. For the sake of simplicity, we introduce following hypotheses without lost of genericity:

- Variable x to be projected appears in all constraints of sc
- Coefficients of $x = +(-)1$.
- l constraints of left side: $L_i <= x$, $\forall i \in [1, l]$
- r constraints of right side: $x <= R_j$, $\forall j \in [1, r]$
- $l \leq r$

Fourier Motzkin combination method has $O(vn^2)$, and in worst case $O(vn^2/2)$. The first approximation is to remove all the constraints that contain the variable to project. The complexity is then constant. Second approximation is to take only the constraint corresponding to the fact that, the average of left side L_i is less or equal to the average of the right side R_j:

$r \sum_{i=1}^{l} L_i \leq rlx \leq l \sum_{j=1}^{r} R_j$

This algorithm reduces matrix size and it is deterministic, because from one system of constraints sc and a variable x, we can derive a unique list of L_i and R_j. The operator *sum* also gives a unique result. Complexity in memory space is $O(v)$, however, the density of non-null coefficients is multiplied by r or l.

The same way, for each L_i of l constraints on the left side, we can combine it with the average of the right side:

$rL_i \leq rx \leq \sum_{j=1}^{r} R_j, \forall i \in [1, l]$

Complexity in this case is $O(vn)$. This algorithm is deterministic. We can as well combine each R_i of r constraints on the right side with the average of the left side:

$\sum_{i=1}^{l} L_i \leq lx \leq lR_j \forall j \in [1, r]$

It becomes more general, if we don't take average of all constraints of one side, but only subsets of them. Let's choose $k = [1, min(r, l)]$. For each L_i of the left side, we combine it with each average of r/k constraints of the right side.

Let $R_t = \sum_{j=t}^{tr/k} (kR_j/r), \forall t \in [1, k]$. We combine L_i (l constraints on the left side) with R_t (k constraints on the right side). The complexity in memory space is then $O(vnk)$. If $k = 1$ then we have the two approximations above. If $k = r$ (or $k = l$), then we have no approximation at all. However, this algorithm is not deterministic, unless we sort the constraints.

Finally, to reduce density of coefficient, we can choose *max* and *min* constraint by a kind of sorting: If exists only one coefficient min non-null of the first dimension, then we choose it. If exist more than one coefficient that share the same value *min* non null, then we compare coefficients of second dimension: we select only one constraint corresponding to the minimal and unique coefficient found, or we continue with other dimensions. At the end, we obtain only one constraint, which makes the algorithm deterministic. This algorithm avoid the explosion in number of constraints after the projection, but it requires some computation, hence we can use it only when it is necessary.

Bibliography

[AB95] David Avis and David Bremner. How good are convex hull algorithms? In *Symposium on Computational Geometry*, pages 20–28, 1995. citeseer.nj.nec.com/avis95how.html.

[AF92] David AVIS and Komei FUKUDA. A pivoting algorithm for convex hulls and vertex enumeration of arrangements and polyhedra. *Discrete and Computational Geometry*, 8(3):295–313, 1992. ACM Symposium on Computational Geometry (North Conway, NH, 1991).

[AF96] David AVIS and Komei FUKUDA. Reverse search for enumeration. *Discrete Applied Mathematics*, 65(1-3):21–46, 1996.

[AGR94] Nancy M. Amato, Michael T. Goodrich, and Edgar A. Ramos. Parallel algorithms for higher-dimensional convex hulls. In *IEEE Symposium on Foundations of Computer Science*, pages 683–694, 1994.

[AI91] Corinne Ancourt and François Irigoin. Scanning polyhedra with DO loops. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 39–50, April 1991. inproceeding.

[ALV] ALV. Action language verifier. http://www.cs.ucsb.edu/ bultan/composite. Project.

[APRa] APRON. Apron compte rendu. http://apron.cri.ensmp.fr. APRON site.

[APRb] APRON. Numerical program analysis. http://www.cri.ensmp.fr/apron. Project APRON.

[AST] ASTREE. Analyseur statique de logiciels temps-reel embarques. www.astree.ens.fr/. Project ASTREE.

[Avi] David Avis. Lexicographical reverse search. Web site:http://cgm.cs.mcgill.ca/avis/C/lrs.htm. Project.

[Ba] R. Bagnara and al. Performance evaluation.

[BAY] Valentina BAYER. Survey of algorithms for the convex hull problem.

[BCC+03] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI*. ACM, 2003.

[BCK+89] M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, L. Pointer, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Scheider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, R. Goodrum, and J. Martin. The PERFECT club benchmarks: Effective performance evaluation of supercomputers. *The International Journal of Supercomputer Applications*, 3(3):5–40, 1989. article.

[BDH96] C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software*, 22(4):469–483, 1996. citeseer.ist.psu.edu/article/barber95quickhull.html.

[BGP97] Tevfik Bultan, Richard Gerber, and William Pugh. Symbolic model checking of infinite state systems using presburger arithmetic. In *Computer Aided Verification*, pages 400–411, 1997.

[BHRZ03] R. Bagnara, P. Hill, E. Ricci, and E. Za. Precise widening operators for convex polyhedra, 2003.

[BHZ02] R. Bagnara, P. M. Hill, and E. Zaffanella. Set-sharing is redundant for pair-sharing. *Theoretical Computer Science*, 2002. To appear.

[BHZ04] R. Bagnara, P. Hill, and E. Za. Widening operators for powerset domains, 2004.

[Bir67] Garrett Birkhoff. *Lattice Theory*. American Mathematical Society, 1967.

[Bou02] Youcef Bouchebaba. *Optimisation des transferts de donnes pour le traitement du signal: pavage, fusion et rallocation des tableaux*. PhD thesis, cole Nationale Suprieur des Mines de PARIS, November 2002.

[BRZaH02] R. Bagnara, E. Ricci, E. Za, and a Hill. Possibly not closed convex polyhedra and the parma polyhedra library, 2002. citeseer.nj.nec.com/bagnara02possibly.html.

[BS97] Binay K. Bhattacharya and Sandeep Sen. On a simple, practical, optimal, output-sensitive randomized planar convex hull algorithm. *J. Algorithms*, 25(1):177–193, 1997.

[BS99] R. Bagnara and P. Schachte. Factorizing equivalent variable pairs in ROBDD-based implementations of *Pos*. In A. M. Haeberer, editor, *Proceedings of the "Seventh International Conference on Algebraic Methodology and Software*

Technology (AMAST'98)", volume 1548 of *Lecture Notes in Computer Science*, pages 471–485, Amazonia, Brazil, 1999. Springer-Verlag, Berlin.

[BYK01] T. Bultan and T. Yavuz-Kahveci. Action language verifier, 2001.

[CAI00] Fabien COELHO Corinne ANCOURT and François IRIGOIN. Linear algebra as a proof engine. Technical report, Centre de Recherche en Informatique, cole des Mines de PARIS, November 2000. tech.

[Cam] Caml. The caml language. http://caml.inria.fr/. The Caml language.

[CC76] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *2nd International Symposium on Programming*, pages 106–130, April 1976.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM Symposium on Principles of Programming Languages*, pages 238–252, January 1977.

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *ACM Symposium on Principles of Programming Languages*, pages 269–282, January 1979.

[CC91] P. Cousot and R. Cousot. Comparison of the Galois connection and widening/narrowing approaches to abstract interpretation. JTASPEFL '91, Bordeaux. *BIGRE*, 74:107–110, October 1991.

[CC92] Patrick Cousot and Radhia Cousot. Comparing the galois connection and widening/narrowing approaches to abstract interpretation. In *PLILP '92: Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming*, pages 269–295, London, UK, 1992. Springer-Verlag.

[CC00] Patrick Cousot and Radhia Cousot. Temporal abstract interpretation. In *ACM Symposium on Principles of Programming Languages*, pages 12–25, January 2000.

[C.E49] Shannon C.E. The synthesis of two-terminal switching circuits. volume 28 of *The Bell Systems Technical Journal*, pages 59–98, 1949.

[CH78] Patrick COUSOT and Nicolas HALBWACHS. Automatic discovery of linear restraints among variables of a program. In *ACM Symposium on Principles of Programming Languages*, pages 84–96, January 1978.

[Cha93] Bernard Chazelle. An optimal convex hull algorithm in any fixed dimension. *Discrete & Computational Geometry*, 10:377–409, 1993.

[Cha95] Timothy M. Chan. Output-sensitive results on convex hulls, extreme points, and related problems. In *SCG '95: Proceedings of the eleventh annual symposium on Computational geometry*, pages 10–19, New York, NY, USA, 1995. ACM Press.

[Che64] N.V. Chernikova. Algorithm for finding a general formula for the non-negative solutions of a system of linear equations. *USSR Computational Mathematics and Mathematical Physics*, 4(4):151–158, 1964.

[Che65] N.V. Chernikova. Algorithm for finding a general formula for the non-negative solutions of a system of linear inequalities. *USSR Computational Mathematics and Mathematical Physics*, 5(2):228–233, 1965.

[Che68] N.V. Chernikova. Algorithm for discovering the set of all the solutions of a linear programming problem. *USSR Computational Mathematics and Mathematical Physics*, 8(6):282–293, 1968.

[CHI] CHINA. A data-flow analyzer for clp languages. http://www.cs.unipr.it/China/. Project.

[Chv83] Vasek Chvata. *Linear Programming (Series of Books in the Mathematical Sciences)*. W.H Freeman and Company, New York/San Francisco, 1983. book.

[CI95] Béatrice Creusillet and François Irigoin. Interprocedural array region analyses. In *International Workshop on Languages and Compilers for Parallel Computing*, volume 1033 of *Lecture Notes in Computer Science*, pages 46–60. Springer-Verlag, 1995.

[CI96] Béatrice Creusillet and François Irigoin. Exact vs. approximate array region analyses. In *International Workshop on Languages and Compilers for Parallel Computing*, volume 1239 of *Lecture Notes in Computer Science*, pages 86–100. Springer-Verlag, 1996.

[CK70] Donald R. Chand and Sham S. Kapur. An algorithm for convex polytopes. *J. ACM*, 17(1):78–86, 1970.

[Cla88a] K. L. Clarkson. Applications of random sampling in computational geometry, ii. In *SCG '88: Proceedings of the fourth annual symposium on Computational geometry*, pages 1–11, New York, NY, USA, 1988. ACM Press.

[Cla88b] Kenneth L. Clarkson. A randomized algorithm for closest-point queries. *SIAM J. Comput.*, 17(4):830–847, 1988.

[Cla96] Philippe Clauss. Counting solutions to linear and nonlinear constraints through ehrhart polynomials: Applications to analyze and transform scientific programs. In *International Conference on Supercomputing*, pages 278–285, 1996.

[Cou97] Patrick Cousot. Program analysis: The abstract interpretation perspective. *ACM SIGPLAN Notices*, 32(1):73–76, 1997.

[Cre95] Béatrice Creusillet. IN and OUT array region analyses. In *Workshop on Compilers for Parallel Computers*, pages 233–246, June 1995.

[Cre96] Béatrice Creusillet. *Analyses de régions de tableaux et applications*. PhD thesis, Centre de Recherche en Informatique, cole des Mines de PARIS, December 1996.

[CS89] K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, ii. *Discrete Comput. Geom.*, 4(5):387–421, 1989.

[CSY95] Timothy M. Y. Chan, Jack Snoeyink, and Chee-Keng Yap. Output-sensitive construction of polytopes in four dimensions and clipped voronoi diagrams in three. In *SODA '95: Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*, pages 282–291, Philadelphia, PA, USA, 1995. Society for Industrial and Applied Mathematics.

[CSY97] Timothy M. Chan, Jack Snoeyink, and Chee-Keng Yap. Primal dividing and dual pruning: Output-sensitive construction of four-dimensional polytopes and three-dimensional voronoi diagrams. *Discrete & Computational Geometry*, 18(4):433–454, 1997.

[CTI] CTI. constraint based termination inference. http://www.cs.unipr.it/cTI/. Project.

[Da] Nurit Dor and al. Cssv: Towards a realistic tool for statically detecting all buffer overflows in c. citeseer.ist.psu.edu/606728.html.

[Ede87] Herbert Edelsbrunner. *Algorithms in combinatorial geometry*. Springer-Verlag New York, Inc., New York, NY, USA, 1987.

[FEA] Paul FEAUTRIER. Parametric integer programming. Project PIP.

[Fea88] Paul Feautrier. Parametric integer programming. *Operationnelle/Operations Research*, 22(3):243–268, 1988. article.

[Fea96] Paul Feautrier. Automatic parallelization in the polytope model. In *The Data Parallel Programming Model*, pages 79–103, 1996. citeseer.ist.psu.edu/feautrier96automatic.html.

[FER04] Jrme FERET. Static analysis of digital filters. In *European Symposium on Programming (ESOP'04)*, number 2986 in LNCS. Springer-Verlag, 2004. Springer-Verlag.

[FER05a] Jrme FERET. The arithmetic-geometric progression abstract domain. In *Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, number 3385 in LNCS, pages 42–58. Springer-Verlag, 2005. To appear, ? Springer-Verlag.

[FER05b] Jrme FERET. Numerical abstract domains for digital filters. In *International workshop on Numerical & Symbolic Abstract Domains (NSAD 2005)*, number ?? in ENTCS, page ?? Elsevier, 2005. To appear, ? Elsevier.

[FLL01] Komei FUKUDA, Th. M. LIEBLING, and Christine LUTOLF. Extended convex hull. *Computational Geometry*, 20(1-2):13–23, 2001. article.

[FP95] Komei FUKUDA and Alain PRODON. Double description method revisited. In *Combinatorics and Computer Science*, pages 91–111, 1995. citeseer.nj.nec.com/fukuda96double.html.

[FQ88] Felipe Fernandez and Patrice Quinton. Extension of chernikova's algorithm for solving general mixed linear programming problem. Technical report, Institut National de Recherche en Informatique et en Automatique Rennes, December 1988. tech.

[Fra] Matthias Franz. Convex - a maple package for convex geometry. http://www-fourier.ujf-grenoble.fr/franz/convex/. Project.

[FUK] Komei FUKUDA. C implementation of double desciption method. Web site: http://www.ifor.math.ethz.ch/fukuda/cdd_home/cdd.html. Project.

[GDD+04] D. Gopan, F. DiMaio, N. Dor, T. Reps, and M. Sagiv. Numeric domains with summarized dimensions, 2004. citeseer.ist.psu.edu/gopan04numeric.html.

[GJ00] Ewgenij Gawrilow and Michael Joswig. polymake: a framework for analyzing convex polytopes. In Gil Kalai and Günter M. Ziegler, editors, *Polytopes — Combinatorics and Computation*, pages 43–74. Birkhäuser, 2000.

[GJ01] Ewgenij Gawrilow and Michael Joswig. polymake: an approach to modular software design in computational geometry. In *Proceedings of the 17th Annual Symposium on Computational Geometry*, pages 222–231. ACM, 2001. June 3-5, 2001, Medford, MA.

[Gra72] R. L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters*, 1(4):132–133, June 1972.

[HAL79] Nicolas HALBWACHS. *Determination automatique de relations lineares verifiees par les variables d'un programe*. PhD thesis, Universite Scientifique et Medicale de Grenoble, France, Mars 1979. thesis.

[HK91] Paul Havlak and Ken Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, 1991.

[HLV94] Doran K. Wilde Herv Le Verge, Vincent Van Dongen. Loop nest synthesis using the polyhedral library. Technical report, Institut National de Recherche en Informatique et en Automatique, May 1994. tech.

[HMPV03] N. Halbwachs, D. Merchat, and C. Parent-Vigouroux. Cartesian factoring of polyhedra in linear relation analysis. In *Static Analysis Symposium, SAS'03*, San Diego, June 2003. LNCS 2694, Springer Verlag.

[HZB01] P. M. Hill, E. Zaffanella, and R. Bagnara. A correct, precise and efficient integration of set-sharing, freeness and linearity for the analysis of finite and rational tree languages. Quaderno 273, Dipartimento di Matematica, Università di Parma, Italy, 2001. www.cs.unipr.it/Publications/.

[IJT91] Franois Irigoin, Pierre Jouvelot, and Rémi Triolet. Semantical interprocedural parallelization: an overview of the pips project. In *ICS*, pages 144–151, June 1991.

[IRI05] François IRIGOIN. Detecting affine loop invariants using a modular static analysis. Technical report, Centre de Recherche en Informatique, cole des Mines de PARIS, September 2005. Version prliminaire–A/368/CRI.

[Jar73] R. A. Jarvis. On the identification of the convex hull of a finite set of points in the plane. *Information Processing Letters*, 2(1):18–21, March 1973.

[JEAa] Bertrand JEANNET. The cuddaux library. http://www.irisa.fr/prive/bjeannet/cuddaux.html. Extension of CUDD.

[JEAb] Bertrand JEANNET. New polka. www.irisa.fr/prive/bjeannet/newpolka.html. Library Polka.

[JEA00] Bertrand JEANNET. *Partitionnement Dynamique dans l'Analyse de Relations Linaires et Application la Vrification de Programmes Synchrones*. PhD thesis, Institut National Polytechnique de Grenoble, September 2000.

[KS86] David G Kirkpatrick and Raimund Seidel. The ultimate planar convex hull algorithm. *SIAM J. Comput.*, 15(1):287–299, 1986.

[LES96] Arnauld LESERVOT. *Analyse interprocdurale du flot des donnes*. PhD thesis, Universit PARIS VI, March 1996.

[LOE] Vincent LOECHNER. Polylib. Web site: icps.u-strasbg.fr/polylib/. Project.

[Loe99] V. Loechner. Polylib: A library for manipulating parameterized polyhedra, 1999. citeseer.nj.nec.com/loechner99polylib.html.

[LUS] LUSTRE. Lustre. http://www-verimag.imag.fr/SYNCHRONE. Project SYNCHRONE.

[MAS92] François MASDUPUY. Array abstractions using semantic analysis of trapezoid congruences. In *ICS*, pages 226–235, 1992.

[MAS93] François MASDUPUY. Semantic analysis of interval congruences. In *Formal Methods in Programming and Their Applications*, pages 142–155, 1993.

[Mer05] David Merchat. *Rduction du nombre de variables en analyse de relations linaires*. PhD thesis, Universit Joseph Fourier Grenoble I, May 2005.

[Mey90] Bertran Meyer. *Introduction to the Theory of Programming Languages*. Prentice Hall, Englewood Cliffs, 1990.

[MIN] Antoine MIN. The octagon abstract domain library. www.di.ens.fr/m̃ine/oct. Library Octagon.

[MIN01a] Antoine MIN. A new numerical abstract domain based on difference-bound matrices. In *PADO II*, volume 2053 of *LNCS*, pages 155–172. Springer-Verlag, May 2001. www.di.ens.fr/ mine/publi/article-mine-padoII.pdf.

[MIN01b] Antoine MIN. The octagon abstract domain. In *AST 2001 in WCRE 2001*, IEEE, pages 310–319. IEEE CS Press, October 2001. www.di.ens.fr/ mine/publi/article-mine-ast01.pdf.

[MIN02] Antoine MIN. A few graph-based relational numerical abstract domains. In *SAS'02*, volume 2477 of *LNCS*, pages 117–132. Springer-Verlag, 2002. www.di.ens.fr/ mine/publi/article-mine-sas02.pdf.

[MIN04a] Antoine MIN. *Domaines numriques abstraits faiblement relationnels: Weakly Relational Numerical Abstract Domains*. PhD thesis, cole Normale Suprieure de PARIS, December 2004.

[MIN04b] Antoine MIN. Relational abstract domains for the detection of floating-point run-time errors. In *ESOP'04*, volume 2986 of *LNCS*, pages 3–17. Springer, 2004. www.di.ens.fr/ mine/publi/article-mine-esop04.pdf.

[MR80] T.H. Matheiss and David S. Rubin. A survey and comparison of methods for finding all vertices of convex polyhedral sets. *Mathematics of operations research*, 5(2):167–185, 1980.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, San Francisco, 1997.

[NBA] NBAC. Nbac. www.irisa.fr/prive/bjeannet/nbac/nbac.html. Project NBAC.

[NGU02] Thi Viet Nga NGUYEN. *Efficient and effective software verifications for scientific applications using static analysis and code instrumentation*. PhD thesis, cole Nationale Suprieur des Mines de PARIS, November 2002.

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, Berlin, 1999.

[Ome] Omega. Omega. http://www.cs.umd.edu/projects/omega. Project.

[OvL80] Mark H. Overmars and Jan van Leeuwen. Dynamically maintaining configurations in the plane (detailed abstract). In *STOC '80: Proceedings of the twelfth annual ACM symposium on Theory of computing*, pages 135–145, New York, NY, USA, 1980. ACM Press.

[PAR] PARMA. Parma polyhedral library. Web site: http://www.cs.unipr.it/ppl/. Project.

[PIPa] PIPS. Linear c3. www.cri.ensmp.fr/pips/. Project LINEAR C3.

[PIPb] PIPS. Paralleliseur interprocedural de programmes scientifiques. www.cri.ensmp.fr/pips/. Project PIPS.

[PS85] F. P. PREPARATA and M. I. SHAMOS. *Computational Geometry: An Introduction*. Texts and Monographs in Computer Science. Springer-Verlag, New York, 1985.

[PUG91] William PUGH. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing*, pages 4–13, 1991.

[PUG92] William PUGH. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, August 1992.

[Qhu] Qhull. Qhull. http://www.qhull.org. Project.

[QUEa] Duong NGUYEN QUE. Hq interface. www.cri.ensmp.fr/people/duong/HQ_API. A common interface prototype.

[QUEb] Duong NGUYEN QUE. Polybench. www.cri.ensmp.fr/people/duong/polybench. Polyhedral Benchmark.

[R.E85] Bryant R.E. Symbolic manipulation of boolean functions using a graphical representation. volume 22 of *IEEE*, pages 688–694, 1985.

[R.E86] Bryant R.E. Graph-based algorithms for boolean function manipulation. volume 35 of *IEEE*, pages 677–691, 1986.

[Sch86] Alexander Schrijver. *Theory of linear and integer Programming.* Wiley-Interscience, New York, 1986. book.

[Sei81] Raimund Seidel. A convex hull algorithm optimal for point sets in even dimensions. Technical report, Vancouver, BC, Canada, Canada, 1981.

[Sei87] R. Seidel. *Output-size sensitive algorithms for constructive problems in computational geometry.* PhD thesis, Ithaca, NY, USA, 1987.

[Sei92] R. Seidel. Backwards analysis of randomized geometric algorithms. Technical Report TR-92-014, ICSI, 1992.

[SLY90] Zhiyu Shen, Zhiyuan Li, and Pen-Chung Yew. An empirical study of fortran programs for parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):356–364, 1990.

[SOG] Jean-Claude SOGNO. Logiciel janus. Restricted. Project Chlo.

[SOG96] Jean-Claude SOGNO. The janus test: a hierarchical algorithm for computing direction and distance vectors. In *Hawaii International Conference on System Sciences*, January 1996. extended version in 2001.

[Som] Fabio Somenzi. The cudd library. http://vlsi.colorado.edu/ fabio/CUDD/cuddIntro.html. The Caml language.

[SPE] SPEC. Spec cfp95 benchmark. http://www.specbench.org/cpu95/CFP95/. Benchmark.

[SS93] Larry SHAFER and William L. STEIGER. Randomizing optimal geometric algorithms. In *CCCG*, pages 133–138, 1993.

[ST01] Daniel A. Spielman and Shang-Hua Teng. Smoothed analysis of algorithms: why the simplex algorithm usually takes polynomial time. In *ACM SIGACT Symposium on Theory of Computing*, pages 296–305, July 2001. inproceeding.

[TC] Andreas Loebel Thomas Christof. Polyhedron representation transformation algorithm. http://www.iwr.uni-heidelberg.de/groups/comopt/software/PORTA. Project.

[TFI86] Remi Triolet, Paul Feautrier, and François Irigoin. Automatic parallelization of Fortran programs in the presence of procedure calls. In *European Symposium on Programming*, March 1986.

[TMT53] G. Thompson T. Motzkin, H. Raiffa and R.M. Thrall. The double description method. *Contributions to the Theory of Games II*, 8:51–73, 1953.

[VCH96] Clark Verbrugge, Phong Co, and Laurie J. Hendren. Generalized constant propagation: A study in c. In *Computational Complexity*, pages 74–90, 1996. citeseer.ist.psu.edu/verbrugge96generalized.html.

[Ver92] Herv Le Verge. A note on chernikova's algorithm. Technical report, IRISA, Feb 1992. tech.

[Ver94] Herv Le Verge. A note on chernikova's algorithm. Technical report, Institut National de Recherche en Informatique et en Automatique, July 1994. tech.

[vLa90] Jan van Leeuwen and al, editors. *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*. Elsevier and MIT Press, 1990.

[WEG00] Ingo WEGENER. *Branching Programs and Binary Decision Diagrams: Theory and Applications*. Siam, Philadelphia, 2000.

[Wen97] R. Wenger. Randomized quickhull. *Algorithmica*, 17:322–329, 1997.

[Wil] D. K. Wilde. A library for doing polyhedral operations. Technical Report RR-2157. citeseer.nj.nec.com/wilde97library.html.

[Wil97a] D. K. Wilde. A library for doing polyhedral operations. Technical report, Brigham Young University, August 1997. tech.

[WIL97b] Doran K. WILDE. A library for doing polyhedral operations. Technical Report RR-2157, Brigham Young University, August 1997.

[YAN93] Yi-Qing YANG. *Tests des Dpendances et Transformations de Programme*. PhD thesis, cole Nationale Suprieur des Mines de PARIS, November 1993.

[YAV04] Tuba YAVUZ. *Specification and Automated Verification of Concurrent Software Systems*. PhD thesis, University of California Santa Barbara, September 2004.

[YKTB01] Tuba Yavuz-Kahveci, Murat Tuncer, and Tevfik Bultan. A library for composite symbolic representations. *Lecture Notes in Computer Science*, 2031:52–??, 2001. citeseer.ist.psu.edu/402436.html.