Universidad {Complutense, Politécnica, Autónoma} de Madrid

Facultad de Informática, Escuela Técnica Superior de Ingenieros Informáticos, Escuela Politécnica Superior

Máster Interuniversitario en Métodos Formales en Ingeniería Informática

ABSTRACT DOMAIN FOR FLOATING-POINT PROGRAMS

Autor: Daniel Jurjo Rivas

Director: Manuel Hermenegildo Salinas

Co-director: Jose F. Morales Caballero

Madrid, 8 Julio 2021

ABSTRACT DOMAIN FOR FLOATING-POINT PROGRAMS

Autor: Daniel Jurjo Rivas
Director: Manuel Hermenegildo Salinas
Co-director: Jose F. Morales Caballero

Acknowledgments

In Spain we say "Es de bien nacidos ser agradecidos" [1] and, since I like to think that I am a "bien nacido," I would like to express my gratitude to my father and my mother, - and why not? - to my brothers. Also I would like to thanks to my partner Laura for all the effort she has done putting up with me, all her support whenever I felt down, and all the good times we have had and we will have. I have the great luck of having a lot of people around me and I would like to also mention Jose, Adri, Aura, Gonza, Marcos and, of course, my "cousin" Raúl; you know what is said.

Last, but not least, I would like to thank my supervisors: Manuel and Jose whom have put trust in me, have devoted time and effort in the development of this work, and have kindly shared with me their deep knowledge. I also would like to thank Isabel García the help that she has offered during this thesis, concretely whith the development of the base domain we used to implement our abstract interpretation.

[1] In English the equivalent saying is "Gratitude is the sign of noble souls" or "Manners make the person" but they do not sound as good as in Spanish.

Resumen

Resumen — Los números en coma flotante aparecen en prácticamente todos los programas actuales, sin embargo verificarlos cuando se usa esta aritmética no es en absoluto trivial. Esto se debe a la naturaleza de la aritmética en coma flotante y a la pérdida de precisión provocada al tratar de representar números reales en un ordenador. Por otro lado, la interpretación abstracta ha demostrado ser una técnica efectiva a la hora de capturar diversos comportamientos del código. Se han propuesto numerosas aproximaciones a este problema usando las técnicas de interpretación abstracta y entre todas ellas hemos decidido estudiar e implementar dentro del Sistema `Ciao` un dominio no relacional basado en intervalos, el cual captura restricciones de la forma $x = y \cdot z$. Hemos extendido este análisis para diferentes representaciones de los números reales utilizadas en la actualidad, con la esperanza de ayudar a los desarrolladores a escoger el tipo de dato numérico más apropiado a la hora de programar. Esto permitiría reducir el consumo de memoria mientras se mantiene bajo control la perdida de precisión, la cual desgraciadamente siempre va a existir. Con este dominio hemos sido capaces de analizar fragmentos de código industrial obteniendo resultados que nos animan a continuar por alguna de las múltiples líneas que puedan aparecer como extensión de este trabajo.

Palabras clave — Aritmética en Coma Flotante, Interpretación Abstracta, Programación Lógica, Análisis numérico.

Abstract

Abstract — Floating point numbers are widely used nowadays in programs but the verification of programs using this type of arithmetic is not trivial at all. This is due to the nature of floating-point arithmetic and the loss of precision when trying to represent real numbers in a computer. On the other hand, abstract interpretation has demonstrated to be effective in capturing different code behaviors and a number of different approaches have been proposed for analyzing floating point, numerical programs. Among these approaches, we decided to implement within the `Ciao` abstract interpretation framework and study a non-relational interval analysis capturing the behavior of constraints of type $x = y \cdot z$. We extended this analysis to different representations of real numbers with the hope of providing a tool for the developer in order to choose the most suitable type when coding. The objective is to be able to minimize the consumption of memory while controlling the loss of precision, which unfortunately is unavoidable in this computations. With this domain we have been able to run some experiments with industrial code obtaining some encouraging results. We also propose a number of research lines stemming from several possible extensions this work.

Key words — Floating Point Arithmetic, Abstract Interpretation, Constraint Logic Programming, Numerical Analysis.

Contents

List of Figures

List of Tables

1

Introduction

Since the beginning of time mankind has tried to answer three questions: Where do we come from? What are we? Where are we going? Since we were -and are- unable to answer any of them, we started doing mathematics, which can be seen as a terrible idea. More recently we have added Computer Science to the picture, which is probably an even worse idea. This master's thesis tries to answer a slightly easier question: Can we have some guarantees when we use floating point numbers?
Floating point computations are present in almost every program nowadays. From the calculus of a salary raise to the position of a satellite, computers perform thousands of floating point calculations. Due to the nature of floating point numbers and their arithmetic, floating point computations usually lead to small errors. Consider the following C program:

```c
int main(void)
{
  float x = 1.0/10;
  printf("The value of x is: %.20f \n", x);
  return 0;
}

```

We will obtain that the value of x is 0.10000000149011611938 wich is not exactly the 0.1 we were expecting. These errors can lead to big problems as they grow during the computations. For example in 1991 a Patriot missile failed its objetive of intercepting an enemy Iraqi Scud missile that crashed on an American building and caused 28 deaths. The error was caused by the representation of 0.1 as floating point. Repeated increments by 0.1 resulted, after running for 100 hours, in an accummulated error of 500m and the

death of 28 people.[1]
This example shows the importance of being able to analyze programs containing floating point computations. Sadly, programs using floating point numbers are difficult to reason about. Some of the factors that complicate this task are [1]:

1. compilers may transform the code not preserving the semantics of floating-point computations,
2. floating-point formats are implementation-defined in most of programming languages,
3. there are different implementations of the operations that are incompatible,
4. mathematical libraries often come with little or no guarantee about what is being computed and developers usually do not read this documentation,
5. predicting and avoiding phenomena caused by the limited range and precision of floating point numbers cost a lot of time and effort to developers. Modern devices support infinities, signed zeroes, non numerical objects ..., that help in some cases but also bring in new issues,
6. rounding is a source of confusion, since there exist different rounding modes and they can also change any time.

These difficulties are a root of evil and as a result verification of floating-point programs in industry tends to rely on informal methods, mainly testing and numerical evaluation of the numerical accuracy. The application of formal methods to verify this kind of programs requires also a solution to the previously enumerated problems. Let us review some of the progress that has been made in each point until now:

1. Currently some compilers provide options to keep the floating-point as is in code. When this option is not available or cannot be used the only possibility is to verify the generated code or the intermediate code which semantic is preserved.
2. The wide adoption of the `IEEE 754` standard has helped avoid most of the problems related to floating-point implementations. However, some other implementations, such as the fixed point, do not have a widely adopted implementation. Also, some older software still uses old implementations.
3. Again, thanks to the `IEEE 754` standard, some basic operations can be computed with some strong guarantees (e.g., that they are correctly rounded). However, no guarantees are provided for some other operations, such as the trigonometric functions.
4. An effective and formal approach to reasoning on mathematical functions has been proposed in [2]. These proposals exploit the fact that the implementation preserves the piecewise monotonicity nature of the approximated functions over $\mathbb{R}$.

[1] `http://www-users.math.umn.edu/~arnold/disasters/patriot.html`

5. The definition and formal proof of the correctness of direct and inverse algorithms for `IEEE 754` arithmetic constraints allows developers to use formal methods in a wide range of programs. These techniques can be used to prove that a wide number of unwanted phenomena do not happen and also allows generating test suites to reduce these issues.

6. Being able to handle floating point rounding modes and being able to handle the uncertainty about the rounding modes that are being used gives more precision to the analysis. Even if the round-to-nearest rounding mode is the most used, a number of issues have to be taken into account when designing an analysis:

   - The `IEEE754` standard granted the possibility of programmatically changing the rounding mode and is offered in most of its implementations,
   - this possibility is exploited by interval libraries and by numerical calculus algorithms [3, 4],
   - changing the rounding mode to something different to round-to-nearest can be done by third-parties causing unexpected behavior [5]. Also this can be exploited in an malicious way. There are also known examples of printer drivers and sound libraries that change the rounding mode and may fail to set it back.

1.1 Objectives

In this work we implement a floating-point based interval analysis using the Ciao Preprocessor. This work must be seen as a work-in-progress since we aim to evolve it into a more complex tool to use in real-world problems. Our approach is built over the Ciao System as a bundle and allows applying multiple analyses to the same program. We aim to:

1. Implement a fast analyzer which allows developers to reduce errors related with floating-point computations,

2. prove the correctness of the algorithms that are being used,

3. make it possible to analyze a program with different floating-point types and numerical representations allowing to save memory without an undesired loss of precision,

4. handle the uncertainty related with rounding modes, not only the `IEEE 754` ones but also for user-defined rounding,

5. extend the `CiaoPP` numerical analyzers,

6. develop different abstract domains within `CiaoPP`.

1.2 Structure of the document

The rest of the thesis is organized as follows: in Chapter 2 we introduce the basic notions and notation needed to develop this work, together with the current work in this field. In Chapter 3 we introduce the `Ciao` system and in particular its preprocessor, in which we implemented the artifacts of this work. Chapter 4 contains the definition of direct projections used in the implemented abstract domain. Chapter 5 shows some early results applying the developed domain. Finally Chapter 6 discuses the current limitations of the tool and the future lines of work that can be taken.

2

Background

2.1 Floating Point Representations

In this chapter we are going to define two of the most used representations of floating point numbers which we have used in this work [6].

2.1.1 Floating Point Numbers

Definition 1 (*IEEE 754 binary floating-point numbers*)**.** *The set of binary floating-points numbers $\mathbb{F}(p, e_{max})$ is defined by:*

- *The numbers of the form $(-1)^s 2^e m$, where $s \in \{0, 1\}$, $e \in [1 - e_{max}, e_{max}] \cap \mathbb{Z}$, and the mantissa*

$$m = (d_0.d_1...d_{p-1})_2 = \sum_{i=1}^{p-1} d_i 2^{-i}$$

  where $p \in \mathbb{N}$ is the precision.

- *The non-numerical values $+\infty$, $-\infty$, qNan and sNan.*

For convenience, we will denote $e_{min} = 1 - e_{max}$. The smallest positive normal floating-point number is $f^{nor}_{min} = 2^{e_{min}}$ and the largest is $f_{max} = 2^{e_{max}}(2 - 2^{1-p})$. The smallest sub-normal magnitude is $f_{min} = 2^{e_{min}+1-p}$. We will write $even(x)$ to signify that $d_{p-1} = 0$ and $sign(x)$ to obtain the value of $(-1)^s$. Moreover we define $\mathbb{F}_- = \{x \in \mathbb{F} | sign(x) = -1\}$ and $\mathbb{F}_+ = \{x \in \mathbb{F} | sign(x) = 1\}$ for a given $\mathbb{F} = \mathbb{F}(p, e_{max})$.

Observation 1. *The set of floating-point numbers is a subset of* $\mathbb{Q}$*.*

Definition 2. *Given* $\mathbb{F}$ *a set of floating-point numbers, we define the relation* $\preceq \subseteq \mathbb{F} \times \mathbb{F}$ *as:*

$$x \preceq y \Leftrightarrow x, y \notin \{qNan, sNan\} \wedge \begin{cases} x = -\infty & y \neq -\infty \\ x \neq +\infty & y = \infty \\ x = -0 & y \in \mathbb{F}_+ \cup \{+0\} \\ x \in \mathbb{F}_- \cup \{-0\} & y = +0 \\ x, y \in \mathbb{R} \cap \mathbb{F} & x < y. \end{cases} \tag{2.1}$$

Some properties arise from this definition.

Proposition 1. $\preceq \subseteq \mathbb{F} \times \ \mathbb{F}$ *defines a partial order.*

Proposition 2. $(\mathbb{F}, \preceq)$ *is a lattice*

Now we have to handle the operations, the big difference between real arithmetic and floating-point arithmetic is due to the rounding. The basic rounding is *round-to-nearest* but it is not the only one.

The following are the rounding modes defined by `IEEE 754` [6, 1].

Definition 3. *Let* $x \in \mathbb{R}$ *then:*

$$[x]_\uparrow = \begin{cases} +\infty & \textit{if } x > f_{max} \\ min\{z \in \mathbb{F}(p, e_{max}) | z \geq x\} & \textit{if } x \leq -f_{min} \vee 0 < x \leq f_{max} \\ -0 & \textit{if } f_{min} < x < 0 \end{cases} \tag{2.2}$$

$$[x]_\downarrow = \begin{cases} max\{z \in \mathbb{F}(p, e_{max}) | z \leq x\} & \textit{if } \ -f_{max} \leq x < 0 \vee f_{min} \leq x \\ +0 & \textit{if } 0 < x < f_{min} \\ -\infty & \textit{if } x < -f_{max} \end{cases} \tag{2.3}$$

$$[x]_0 = \begin{cases} [x]_\downarrow & \textit{if } x > 0 \\ [x]_\uparrow & \textit{if } x < 0 \end{cases} \tag{2.4}$$

$$[x]_n = \begin{cases} [x]_\downarrow & \textit{if} - f_{max} \leq x \leq f_{max} \wedge \\ & |[x]_\downarrow - x| < |[x]_\uparrow - x| \vee \\ & |[x]_\downarrow - x| = |[x]_\uparrow - x| \wedge even([x]_\downarrow) \\ [x]_\downarrow & \textit{iff}_{max} < x < 2^{e_{max}}(2 - 2^{-p}) \vee x \leq -2^{e_{max}}(2 - 2^{-p}) \\ [x]_\uparrow & otherwise \end{cases} \tag{2.5}$$

Proposition 3. *Let $x \in \mathbb{R}\backslash\{0\}$ then*

$$\begin{aligned}
&[x]_{\downarrow} \leq x \leq [x]_{\uparrow}\\
&[x]_{\downarrow} \leq [x]_0 \leq [x]_{\uparrow}\\
&[x]_{\downarrow} \leq [x]_n \leq [x]_{\uparrow}\\
&[x]_{\downarrow} \leq -[-x]_{\uparrow}
\end{aligned}$$

2.1.2 Fixed point arithmetic

Another possible representation of floating point numbers is the representation of floating point numbers as *fixed-point number* [7, 8]. This representation is useful to represent fractional values in base 2 or 10. Most modern languages do not explicitly support fixed-point representation; however, it is present in others such as `COBOL` or `ADA`. In any case, in 2008 the International Standards Organization proposed extending C with fixed-point data types for the benefit of programs running on embedded processors. This representation has been widely used in graphics engines such as the Sony's PlayStation, Game Boy Advance, Nintendo DS (2D and 3D), and in the Gamecube [9]. *Doom* [1] was the last first-person shooter title by id Software to use a 16.16 fixed point representation for all of its non-integer computations. It is also present in all relational databases, moreover the SQL notation supports fixed-point decimal arithmetic and storage of numbers. PostgreSQL has a special numeric type for exact storage of numbers with up to 1000 digits[10]. In quantum computing fixed point arithmetic is being used in the `Q#` programming language. It contains a numeric library for fixed point arithmetic on registers of qubits.

Definition 4. *The set of binary fixed-point numbers $\mathbb{F}ix(p, q)$,*[2] *which is defined by the $N - bit$ binary word:*

$$-1^s b_{p-1} b_{p-2} b_{p-3} \ldots b_0 . b_{-1} b_{-2} \ldots b_{-q}$$

where $1 + p + q = N$, s represents the sign, and bit b_k has a weight of 2^k. Note that q represents the number of bits used to represent decimals. Also note that the representation is an integer number in a range from $-2^p + 2^{-q}$ to $2^p - 2^{-q}$.

Observation 2. *The decimal fixed point representation would be straightforward just substituting 2 by 10.*

It is clear that $\mathbb{F}ix(p, q) \subset \mathbb{Z}$ for all $p, q \in \mathbb{N}$. So we just have to consider the lattice $(\mathbb{F}ix(p, q), \leq)$ with $\leq$ the usual order between integers.

The following algorithms evaluates a mapping between the binary fixed-point numbers and doubles.

```c
long long int to_fixed(double x, int prec)
{
```

[1] In almost every computer science related blog there is someone talking about Doom so here we are.

[2] We choose this notation for clarity and because of its resemblance with `fixed<n, q>` which is used when working with fixed point in C. Another, more standard notation is Q(p, q) or X(p, q) [8]

```
  double aux = x * pow(2, prec);
  long long int b = round(aux);
  if (aux < 0)
  {
    b  = llabs(b);
    b  = ~b;
    b  = b+1;
  }
  return b;
}
```

Example 1. *Consider the floating point number* 3.14 *whose representation as floating-point number is* 3.14000000000000012434497875801753252744674682617188*. If we represent it as a fixed-point with precision 3, we obtain that it is 25, which is an integer number which we can use without any rounding related problem.*

```
  double to_double(long long int fix, int prec)
{
  long long int c = llabs(fix);
  int sign = 1;
  if (fix < 0){
    c = c -1;
    c = ~c;
    sign = -1;
  }
  double f = (1.0 * c)/pow(2, prec);
  f = f * sign;
  return f;
}
```

2.2 Abstract Interpretation

Abstract interpretation [11] was developed by Radhia and Patrick Cousot in the late 1970s. This technique allows constructing sound program analysis tools which can extract properties of a program by approximating its semantics. Since our work is based in this technique let us give some basic insights.

2.2.1 Abstract Interpretation Basics

Sets and orders

Definition 5. *Let S and T be two sets then:*

- $\wp(S) = \{X \subseteq S\}$ *denotes the powerset of S,*
- $S \setminus T = \{x \in S | x \notin T\}$ *the set-difference,*

- $\overline{S} = \{x|x \notin S\}$ *the set complementation,*
- $|S|$ *denotes the cardinality of* S*,*
- $S \subset T$ *denotes the strict inclusion and* $S \subseteq T$ *the inclusion.*

We say that a set S is finite if its cardinal is finite.

Definition 6. *Let* S *a set, the binary relation* R *is an order relation if and only if:*

- $\forall x \in S, xRx$
- $\forall x, y \in S, xRy \wedge yRx \Rightarrow x = y$
- $\forall x, y, z \in S, xRy \wedge yRz \Rightarrow xRz$

Definition 7. *A set* L *with ordering relation* $\sqsubseteq$ *is a* poset*, and it is usually denoted as* $\langle L, \sqsubseteq \rangle$.

Definition 8. *A poset* $\langle L, \sqsubseteq \rangle$ *is a* lattice*, denoted* $\langle L, \sqsubseteq, \sqcup, \sqcap, \top, \bot \rangle$*, if* $\forall x, y \in L$ *we have that the* least upper bound (lub) $x \sqcup y$*, the* greatest lower bound (glb) $x \sqcap y$*, the* greatest element *(top)* $\top$*, and the* least element *(bottom)* $\bot$ *belong to* L*. It is* complete *when for every* $X \subseteq L$ *we have that* $\bigsqcup X, \sqcap X \in L$*. We use subscripts, like in* $\bot_L$*,* $\top_L$*, or* $\bigsqcup_L X$ *to disambiguate the underlying lattice when it is not evident from the context.*

Definition 9. *We say that a lattice* L *is complete if* $\forall X \subseteq L$ *both* $\sqcup X$ *and* $\sqcap X$ *exist.*

Definition 10. *A poset* S *with order* $<$ *is said to satisfy the* ascending chain condition*(ACC) if and only if does not exist any infinite strictly ascending sequence* $x_1 < x_2 < x_3 < ...$ *with* $x_i \in S \forall i$.

Functions.

Given $f : S \rightarrow T$ and $g : T \rightarrow Q$ we denote with $g \circ f : S \rightarrow Q$ their composition, i.e., $(g \circ f)x = g(f(x))$. We let $\mathtt{id}_S : S \rightarrow S$ be the identity function over S.

Definition 11. *Let* $f : l \rightarrow D$ *a function and let* L *and* D *be two lattices. We say that* f *is* additive (resp. co-additive) *if and only if:*

$$\forall X \subseteq L, X \neq \emptyset, f(\sqcup_L X) = \sqcup_D f(X)(resp. f(\sqcap_L X) = \sqcap_D f(X)). \tag{2.6}$$

Continuity is kept when f preserves *lubs* of increasing (non-empty) chains. If $f : L \rightarrow D$ we overload the notation by writing $f : \wp(L) \rightarrow \wp(D)$ for the additive extension of f to sets of values (i.e., for any non-empty $S \in \wp(L)$ we have $f(S) = \{f(v) \mid v \in S\}$). For a continuous function f, the least fixed point $lfp(f) = \sqcap\{x \mid x = f(x)\} = \sqcup_{n \in \mathbb{N}} f^n(\bot)$ where $f^0(\bot) = \bot$ and $f^{n+1}(\bot) = f(f^n(\bot))$. Note that additive functions are continuous.

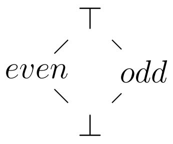

Figure 2.1: Hasse diagram of the Parity Domain lattice.

2.2.2 Abstract domains

Definition 12. *Let C (*concrete*) and A (*abstract*) be complete lattices. And let $\alpha : C \to A$ and $\gamma : A \to C$ two monotone functions. A pair (α, γ) forms a* Galois connection *between C and A if for any $c \in C$ and $a \in A$ we have $\alpha(c) \sqsubseteq_A a \Leftrightarrow c \sqsubseteq_C \gamma(a)$. The function α (resp. γ) is the* left-adjoint *(resp.* right-adjoint*) to γ (resp. α) and it is additive (resp. co-additive). A Galois connection such that $\alpha \circ \gamma = \mathtt{id}_A$ is called a* Galois insertion.

Given a Galois connection, we call $\mathsf{A} = \langle A, \sqsubseteq, \sqcup, \alpha, \gamma \rangle$ an *abstract domain*, with join operator $\sqcup$. An abstract domain satisfies the *ascending chain condition* (*ACC*) if it has no infinite ascending chain. In such cases the fixpoint of any monotone function can be effectively computed in a finite number of steps, or by the use of a *widening* operator ∇ [11].

Definition 13. *A binary operator $\diamond : A \times A \to A$ is a* widening operator *in an abstract domain $(A, \sqsubseteq)$ if*

1. *computes upper bounds: $\forall x, y \in A, x \sqsubseteq x \diamond y \wedge y \sqsubseteq x \diamond y$*
2. *$\forall (x_i)_{i \in \mathbb{N}}, (y_i)_{i \in \mathbb{N}} \in A, \exists k \geq 0 s.t. x_{k+1} = x_k$ where x is computed as $x_0 = y_0$, $x_{i+1} = x_i \diamond y_{i+1}$*

Example 2 (Trivial widening)**.** *The most trivial widening satisfying the definition is:*

$$x \diamond y = \begin{cases} x & \text{if } y \sqsubseteq x \\ \top & \text{otherwise} \end{cases} \tag{2.7}$$

Example 3 (Abstracting parity)**.** *Let the concrete domain of integers be $D_{\mathsf{Int}} = \langle \wp(\mathbb{Z}), \subseteq, \cup, \cap, \emptyset, \mathbb{Z} \rangle$, let $C_{even} = \{\bot, even, odd, \top\}$, and $\gamma : C_{even} \to \wp(\mathbb{Z})$ and $\alpha : \wp(\mathbb{Z}) \to C_{even}$ be defined as*

$$\gamma(x) = \begin{cases} \emptyset & \text{if } x = \bot \\ \{1\} & \text{if } x = odd \\ \{0\} & \text{if } x = even \\ \mathbb{Z} & \text{otherwise} \end{cases} \qquad \alpha(x) = \begin{cases} \bot & \text{if } x = \emptyset \\ odd & \text{if } \forall y \in x, \exists n \in \mathbb{N} y = 2n - 1 \\ even & \text{if } \forall y \in x, \exists n \in \mathbb{N} y = 2n \\ \top & \text{otherwise} \end{cases}$$

Let $\sqsubseteq_{even} \subseteq (C_{even} \times C_{even})$ be defined as $x \sqsubseteq_{even} y$ if and only if $\gamma(x) \subseteq \gamma(y)$. The Hasse diagram of the lattice induced by $\sqsubseteq_{even}$ is shown in 2.1. The lattice $\langle C_{even}, \sqsubseteq_{even} \rangle$,

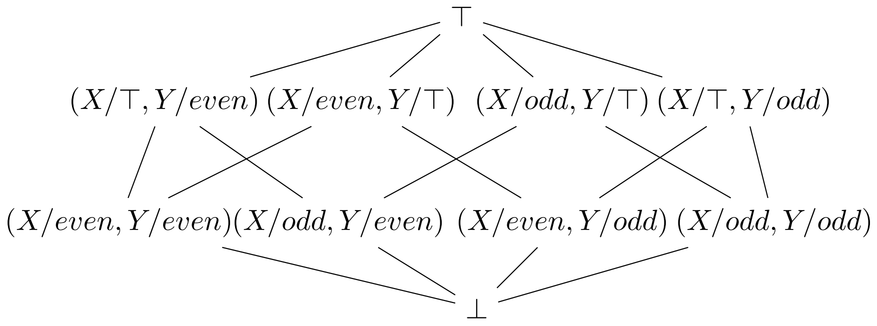

Figure 2.2: An abstract domain using program variables X and Y, and the even lattice.

together with γ can be used to capture the property "the program returns an even or odd value". For instance, assuming that a program can return any natural number, the output of the abstract interpretation process would be a value in C_{even} corresponding to an over approximation to the set of actual values of the execution.

Usually being able to only represent input/output values of a program is very limiting since abstract values are usually related to variables in the program. This is typically represented using sets of *Var/AbstractVal* pairs.

Example 4 (Abstracting parity)**.** *A domain to infer whether the variables in a program take values that are even or odd. D_{Even}, is built using the lattice $\langle C_{even}, \sqsubseteq_{even}\rangle$. Given a program P, the set of values in the abstract domain is $A_P = \{X/v \mid X \in vars(P), v \in C_{even}\}$. That is, the Cartesian product of the program variables and the values in C_{even}. The ordering $\sqsubseteq_{D_{Bits}}$ is built extending $\sqsubseteq_{Bits}$ to the pairwise (i.e., variable per variable) comparison of the values assigned to each of the variables in $dom(\lambda), \lambda \in A_P$. 2.2 shows a lattice for an abstract domain for a program with two variables X, Y using C_{even}.*

Note that, in general, in the *Var/AbstractVal* pairs, abstract values can actually be any term, including variables, thus allowing the represention of *relational* properties, i.e., properties that represent relations between variables, e.g., "$x > y$". Also note that since the number of variables in the program is known (and finite), if we have a finite lattice, one can always build a finite lattice of *Var/AbstractVal* pairs. Of course this is only computable if the terms used as abstract values are finite.

2.3 State of the Art

2.3.1 Interval Domain

The interval domain is one of the best-known abstract domains and is one of the first domains to consider when working with these problems.
ECLAIR [2] is one of the most recent tools to analyse numerical programs. The numerical

[2] https://www.bugseng.com/eclair

analysis is based on the use of intervals to work with floating point arithmetic [1]. This system is able to detect a large number of faults and to categorize possible underflows.

Interval domains normally consider the following concrete and abstraction functions over the lattice of intervals of a given numerical space, let's say N:

$$\gamma(x) = \begin{cases} \emptyset & \text{if } x = \bot \\ y & \text{if } x = [y, y] \\ \{y \in N | y \in [l, u]\} & \text{if } x = [l, u] \\ N & \text{otherwise} \end{cases}$$

$$\alpha(x) = \begin{cases} \bot & \text{if } x = \emptyset \\ [min(N), max(N)] & \text{if } x \subset N \\ \top & \text{otherwise} \end{cases}$$

Example 5. *The operations defined for the general interval domain are quite straightforward, and can be induced just considering the possibilities of a given operation. For brevity let's consider just addition and multiplication:*

$$\begin{aligned} int_1 + int_2 &= \bot \; if \; int_i = \bot \; for \; some \; i \in \{1, 2\} \\ [l1, u1] + [l2, u2] &= [l1 + l2, u1 + u2] otherwise \end{aligned}$$

$$\begin{aligned} int_1 * int_2 &= \bot \; if \; int_i = \bot \; for \; some \; i \in \{1, 2\} \\ [l1, u1] * [l2, u2] &= [min(\{l_i * u_j | i, j \in \{1, 2\}\}), max(\{l_i * u_j | i, j \in \{1, 2\}\})] otherwise \end{aligned}$$

Consider the following piece of code:

```
int main(void)
{
  uint x;
  int y;
  int z;
  int t;
  if (x <= 2) {
    y = x + x;
    z = x*y;
    w = x - x;
    t = z - 2*x-y;
  }
}
```

In this case we have the following propagation:

x = [0, 2]	y = [0, 4]
z = [0, 8]	w = [-2, 2]
t = [-4, 12]	

Notice that the result of $x - x$ in line 10 is not the interval [0, 0]. This is due to the nature of interval analysis and it is also the cause of some loss of precision.

2.3.2 Affine Domain

An abstract numerical domain based on affine arithmetic [12] is presented in [13]. This solution is an extension of Interval Arithmetic that takes linear correlations into account making this domain relational. In this domain values are represented in affine form as follows:

$$x = x_0 + \sum_{i=0}^{n} x_i \epsilon_i \tag{2.8}$$

Where each $x_i \in \mathbb{R}$ and $\epsilon_i \in [-1, 1]$. The ϵ_i are denoted as *noise symbols* and help the analysis to know which variables are *implicitly dependent.*

The Galois connection for this domain over a numerical domain N is given by:

$$\gamma(x) = \begin{cases} \emptyset & \text{if } x = \bot \\ [x_0, x_0] & \text{if } x = x_0 \\ [x_0 - \sum_{i=1}^{n} x_i, x_0 + \sum_{i=1}^{n} x_i] & \text{if } x = x_0 + \sum_{i=0}^{n} x_i \epsilon_i \\ N & \text{otherwise} \end{cases}$$

$$\alpha(x) = \begin{cases} \bot_{Aff} & \text{if } x = \bot \\ x = \frac{l+u}{2} + \epsilon_n & \text{if } x = [l, u] \\ \top & \text{otherwise} \end{cases}$$

Notice that in this case we didn't define in detail the case $x = [l, u]$. This is because the abstraction on this domain is complex and it is not the main objective of this work. Anyway a brief explanation is that α is not applied during the analysis, the abstraction is done directly over the numerical domain N so $\alpha_{aff} = \alpha(\alpha_{Int}(x))$ for $x \in N$.

The operations defined for the affine domain are more complex, specially the product and the division. Addition of affine numbers is computed component-wise and does not add new noise. Consider $x = x_0 + \sum_{i=1}^{n} x_i \epsilon_i, y = y_0 + \sum_{i=1}^{n} y_i \epsilon_i$

$$x + y = (x_0 + y_0) + \sum_{i=1}^{n} (x_i + y_i) \epsilon_i \tag{2.10}$$

The product is quite more complex and there is more than one definition [13, 14]. This is because a new noise term is added. Consider the following definition:

$$x * y = (x_0 y_0 + \frac{1}{2}\sum_{i=1}^{n}|x_i y_i|) + \sum_{i=1}^{n}(x_i y_0 + x_0 y_i)\epsilon_i + (\frac{1}{2}\sum_{i=1}^{n}|x_i y_i| + \sum_{i \neq j}|x_i y_j|)\epsilon_{n+1} \quad (2.11)$$

Which is, as said, more complex but also captures better the operations providing more narrow intervals. This approach is implemented in FLUCTUAT[3] Considering the previous example we get the propagation and its interval equivalence:

$x = 1 + \epsilon_1 = [0, 2]$	$y = 2 + \epsilon_1 + \epsilon_2 = [0, 4]$
$z = \frac{5}{2} + 3\epsilon_1 + \epsilon_2 + \frac{3}{2}\epsilon_3 = [-3, 8]$	$w = 0 = [0, 0]$
$t = \frac{-3}{2} + \frac{3}{2}\epsilon_3 = [-3, 0]$	

This domain offers more tight intervals and better control of error and where it appears. Anyway it requires a larger effort and the complexity of some basic operations grows faster.

2.3.3 Polyhedra Domain

Another well-known domain is the Polyhedra Domain [15]. In this case the values are presented as linear inequalities. This domain has been widely used, some implementations are the Parma Polyhedra Library [16] and a ELINA which includes a fast polyhedra domain [17] together with others as zones or octagons.

Again from the example presented in 5 we can infer the following linear inequalities.

$0 \leq x \leq 2$	$0 \leq y - x \leq 2$
$0 \leq z \leq 8$	$-2 \leq w \leq 4$
$-8 \leq t \leq 8$	

The polyhedra domain has been studied for quite some time and there exist some variations that are able to obtain good approximations of the polyhedra in less time. Some of them are:

- Octagon domain. This domain, presented by A.Miné [18], introduces a relational, numerical abstract domain well-suited for use in static analysis by abstract interpretation. It allows representing conjunctions of constraints of the form

[3] https://www.lix.polytechnique.fr/Labo/Sylvie.Putot/fluctuat.html

$\pm X \pm Y \leq c$ where X and Y are program variables and c is a constant in $\mathbb{Z}, \mathbb{Q}$, or $\mathbb{R}$ automatically inferred. It achieves quadratic memory cost per abstract element and a cubic worst-case time cost per abstract operation, with respect to the number of program variables.

- Zone domain. The Zone abstract domain [19] is able to represent and manipulate invariants of the form $X - Y \leq c$ and $\pm X \leq c$ with a time-cost similar to the Octagon domain.

2.3.4 Other Domains

There are many other domains that have been proposed to solve the problems related with floating point numbers using abstract interpretation. It is worth mentioning the work implemented by A. Miné [20], where the precision of floating point is improves symbolically on-the-fly. The main idea is to simplify numerical expressions before they are fed to abstract transfer functions. This method was implemented within the **ASTRÉE** static analyzer that has been used in embedded critical avionics software. There are also applications of abstract domains in order to analyze model-based tools such as Matlab/Simulink [21].

3
The Ciao System

Ciao [22] is a modern, multiparadigm programming language built up from a logic-based kernel supporting constraint logic programming, different levels of modularity, an assertion system, multiparadigm programming, and interfacing with foreign code. The main motivation behind the system is to develop a combination of programming language and development tools that together help programmers produce better code in less time and with less effort. To do this there are two main approaches: *verification* and *testing*. The first one uses formal methods to prove some specifications of the code while the testing consists in executing concrete inputs and checking that the program input-output relations are the expected.

The Ciao language introduced a development workflow [23, 24, 25] that integrates the two approaches above. In this model, program *assertions* (see 3.1.2) are fully integrated in the language, and serve both as specifications for static analysis and as run-time check generators, unifying run-time verification and unit testing with static verification and static debugging. Assertions are optional and the model admits from the start that some parts of assertions may not be checkable at compile-time and will then generate run-time tests for them when possible. This model represents an alternative approach for writing safe programs without relying on full static typing, which is specially useful for dynamic languages like Prolog. The intention is to combine the best elements from static and dynamic language approaches [26] and is an antecedent to the now popular *gradual-* and *hybrid-typing* approaches [27, 28]

A high-level view of the Ciao System is shown in 3.1. Blue-colored boxes represent user-written code; green boxes represent different tools within the system: the compiler, LPdoc and the CiaoPP Program Processor; and the red box represents the execution environment of the system, i.e., its run-time abstract machine and libraries. In this thesis, only some of them are detailed, as not all of them are used.

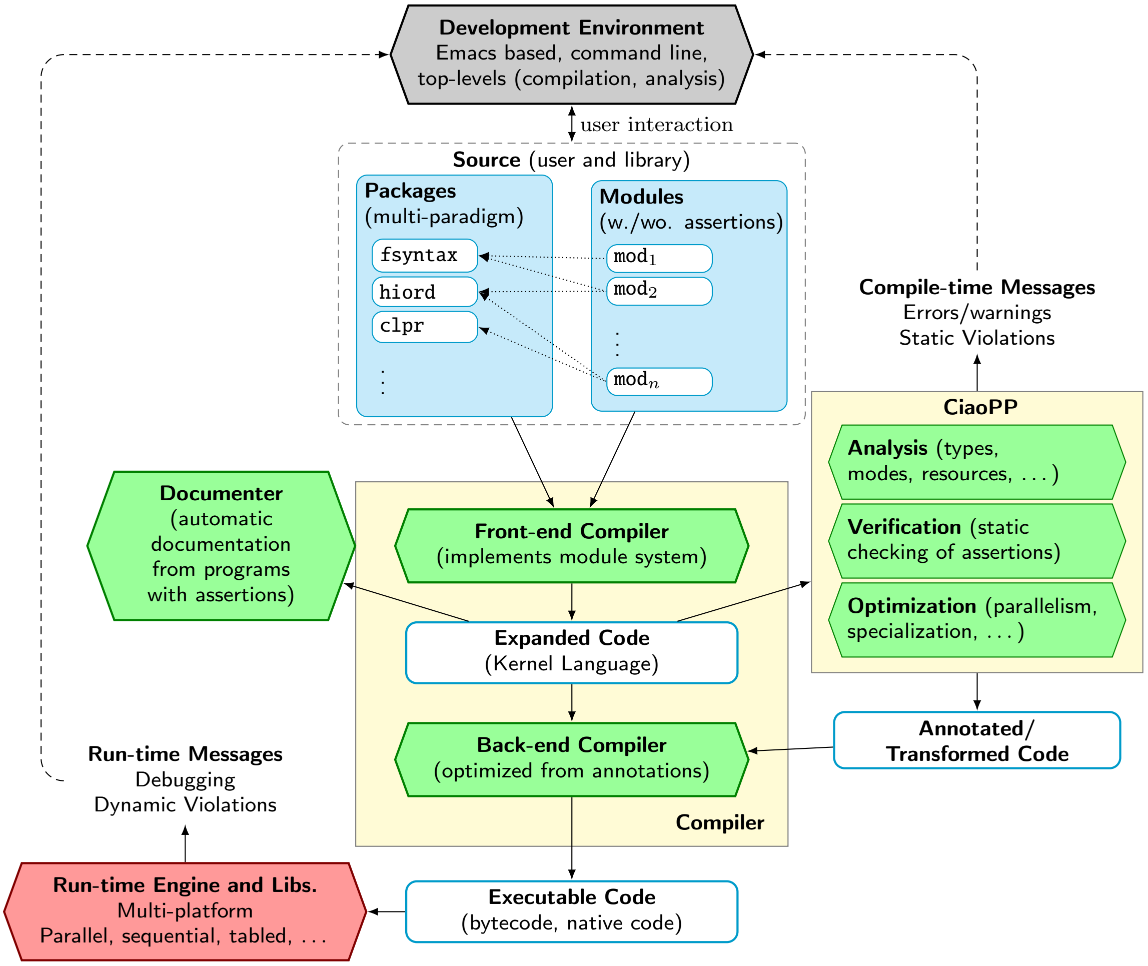

Figure 3.1: A high-level view of the Ciao system [29].

Most of the work in this work is based on the Ciao Program Preprocessor which allowed us to develop static analyses via abstract interpretation.

3.1 The CiaoPP Program Processor

CiaoPP[1] (see the right part of 3.1, and 3.2) is the abstract interpretation-based preprocessor of the Ciao multi-paradigm program development environment.CiaoPP can perform a number of program debugging, analysis, and source-to-source transformation tasks on (Ciao) Prolog programs. The tasks performed by CiaoPP include:

- Inference of properties at the level of predicates and literals of the program, including types, modes and other variable instantiation properties, non-failure, determinacy, bounds on computational cost, bounds on sizes of terms in the program, etc.

[1] https://cliplab.org/~clip/Software/Ciao/ciaopp-1.2.0.html/ciaopp_ref_man.html

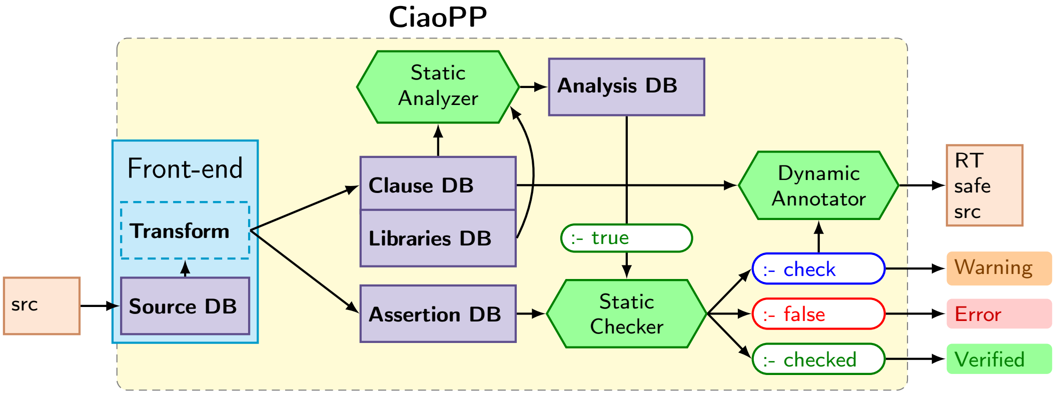

Figure 3.2: Architecture of the CiaoPP verification framework.

- Static debugging and verification. This includes checking how programs call system library predicates and also checking the assertions present in the program or in other modules used by the program.

- Source to source program transformations such as program specialization, slicing, partial evaluation, and program parallelization (with granularity control). It also produces run-time test annotations for assertions which cannot be checked completely at compile-time, so that the program can be run safely by dynamically checking properties.

- Producing abstract models of programs that act as certificates of the correctness of the code. The system is used to certify that code is safe with respect to the given policy (i.e., an abstraction-carrying code approach to mobile code safety [30]).

All the aforementioned features rely on the statically inferred properties based on fixpoint computation. Figure 3.2 provides an overview of the components in CiaoPP. CiaoPP has a *Front-end* that transforms programs (possibly written in a different language) to extract clauses and assertions (specification of the program). The *Static Analyzer* component has several fixpoint computation algorithms that are used to produce analysis graphs (stored in the *Analysis DB*). The information in the Analysis DB (true assertions) is used to statically check the assertions in the *Static Checker*. For each assertion originally with status check, the result of this process can be: that it is verified (the new status is checked), that a violation is detected (the new status is false), or that it is not possible to decide either way, in which case the assertion status remains as check, as detailed in 3.1.2. In such cases, optionally, a warning may be displayed and/or a run-time test generated by the *Dynamic Annotator* component for the (the part of) the assertion that could not be discharged at compile-time, test cases generated, etc.

3.1.1 Supporting multiple languages

A basic technique used by the CiaoPP framework, in order to support different input languages, is to translate input programs, possibly containing assertions, to a languageindependent intermediate representation, which in this case is (constrained) Horn clause-based [31] –an approach used nowadays in many analysis and verification tools [32, 33, 31, 34, 35, 36, 37, 38, 39].

As mentioned before, such translations are performed by the "Front-end" (Fig. 3.2). Techniques such as partial evaluation and program specialization offer powerful methods to obtain such translations with provable correctness. Using this approach, CiaoPP has been applied to the analysis, verification, and optimization of a number of languages (besides `Ciao`) ranging from very high-level ones to bytecode and machine code, such as Java, XC (C like) [40], Java bytecode [41, 42], ISA [43], LLVM IR [44], Michelson [45], . . .), and properties ranging from pointer aliasing and heap data structure shapes to execution time, energy, or smart contract "gas" consumption [46, 47].

We refer to the Horn clause-based intermediate representation as the "HC-IR." The key point, that is directly relevant to our work, is that this HC-IR is handled uniformly by the analyzers, independently of the input language . This means that we can develop our analyses within this Horn clause-based framework, and they will then be applicable to any input language for which a translation is written.

In our examples we will use both imperative and logic programs, sometimes providing the logic program (i.e., the HC-IR) equivalent to the imperative program and others using logic programs directly. More concretely, the examples will be written in `Ciao`'s [29] logic programming subset, with Prolog-style syntax and operational semantics and using `Ciao`'s assertion language, (introduced in the following section). The framework itself and the abstract domains developed are also written in `Ciao`.

The main implication of this approach in our work is that we do not need to review the different abstract interpretation techniques used for different languages, but rather it suffices to understand and use the techniques developed for analysis of the HC-IR, i.e., for analysis of logic programs.

3.1.2 Assertions

Assertions are linguistic constructions that allow stating properties of a program, such as, e.g., conditions on the state (current substitution or constraint) that hold or must hold at certain points of program execution. During this work we use the Ciao assertion language [24, 23, 48, 25], and since the results of our analysis are represented as assertions it is useful to understand them. These assertions are instrumental for many purposes, such as expressing the results of analysis, providing specifications, guiding the analysis, and documenting. Such assertions can express a wide range of properties, including functional (state) properties (e.g., shapes, modes, sharing, aliasing, . . .) as well as non-functional (i.e., global, computational) properties such as resource usage (energy, time, memory, . . .), determinacy, non-failure, or cardinality. The set of properties that can be used in assertions is extensible and new abstract domains can be defined as "plug-ins" to support them. Without loss of generality, we use for concreteness a subset of the syntax of the **`pred`** assertions of [49, 24, 48], which allows describing sets of *preconditions* and *conditional postconditions* on the state for a given predicate as well as global properties. A `pred` assertion is of the form:

:- [*Status*] pred *Head* [: *Pre*] [=> *Post*] [+ *Comp*].

where *Head* is a predicate descriptor (i.e., a normalized atom) that denotes the predicate that the assertion applies to, and *Pre* and *Post* are conjunctions of *property literals*, i.e., literals corresponding to predicates meeting certain conditions which make them amenable to checking [48]. *Pre* expresses properties that hold when *Head* is called, namely, at least one *Pre* must hold for each call to *Head*. *Post* states properties that hold if *Head* is called in a state compatible with *Pre* and the call succeeds. *Comp*describes properties of the whole computation such as resource usage, termination, determinism, non-failure, etc., and they apply to calls to the predicate that meet *Pre*. *Pre*, *Post*, and *Comp*can be empty conjunctions (meaning true), and in that case they can be omitted. *Status* is a qualifier of the meaning of the assertion. The following statuses are intended to be specified by the programmer:

- `check`: the assertion expresses properties that must hold at run-time, i.e., that the analyzer should prove (or else generate run-time checks for). `check` is the *default* status, and can be omitted.
- `trust`: the assertion represents an actual behavior of the predicate that the analyzer assumes to be correct although it may not be able to infer it automatically.

The following statuses are intended to be used as communication between the different components and providing information to the user, as part of the analysis/verification process (corresponding to the ovals in 3.2):

- `checked`: the analyzer proved that the property holds in all executions.
- `true`: the analyzer inferred the assertion.
- `false`: the analyzer proved that the property does not hold in some execution.

As mentioned before, *parts* of assertions that cannot be discharged statically will remain in `check` status and run-time tests will be generated for them if necessary.

Example 6. *The following assertions describe different behaviors of the* `pow` *predicate that computes* $\mathtt{P} = \mathtt{X}^{\mathtt{N}}$*:* `(1)` *is stating that if the exponent of a power is an even number, the result (*`P`*) is non-negative,* `(2)` *states that if the base is a non-negative number and the exponent is a natural number the result* `P` *also is non-negative:*

```
:- pred pow(X,N,P) : (int(X), even(N)) => P ≥ 0.  % (1)
:- pred pow(X,N,P) : ( X ≥ 0,  nat(N)) => P ≥ 0.  % (2)
pow(_, 0, 1).
pow(X, N, P) :-
    N > 0,
    N1 is N - 1,
    pow(X, N1, P0),
    P is X * P0.

:- prop even/1.
even(N) :-
    0 is N mod 2.
```

Here, the **`even/1`** *property is defined by the user, while* **`int/1`** *and* **`nat/1`** *are assumed to be understood by the abstract domain. The* predicate *defining the property is analyzed using the abstract domain, thus inferring the abstract meaning of the user-defined property, and that meaning is used. Different treatment is required when the assertion is used for analysis or for verification.*

In addition to **`pred`** assertions we also consider *program-point assertions.* They are expressed as regular literals using as predicate name their *Status*, i.e., `trust`(*Cond*) and `check`(*Cond*). They imply that whenever the execution reaches a state originated at the program point in which the assertion appears, *Cond* (should) hold. Without loss of generality we limit the discussion to **`pred`** assertions. Program-point assertions can be translated to **`pred`** assertions,

Definition 14 (Meaning of a Set of Assertions for a Predicate)**.** *Given a predicate represented by a normalized atom Head, and a corresponding set of assertions* $\{a_1 \ldots a_n\}$*, with* a_i = "**`:- pred`** *Head* **`:`** *Pre*$_i$ **`=>`** *Post*$_i$**`.`**" *the set of* assertion conditions *for Head is* $\{C_0, C_1, \ldots, C_n\}$*, with:*

$$C_i = \begin{cases} \mathsf{calls}\ (\mathit{Head}, \bigvee_{j=1}^{n} \mathit{Pre}_j) & i = 0 \\ \mathsf{success}(\mathit{Head}, \mathit{Pre}_i, \mathit{Post}_i) & i = 1 \ldots n \end{cases}$$

where calls (*Head*, *Pre*)[1] states conditions on all concrete calls to the predicate described by *Head*, and success(*Head*, *Pre*$_i$, *Post*$_i$) describes conditions on the success constraints produced by calls to *Head* if *Pre*$_i$ is satisfied. These allow representing behaviors for the same predicate for different call substitutions (multivariance). If the assertions a_i above, $i = 1, \ldots, n$, include a + *Comp* field, then the set of *assertion conditions* also include conditions of the form comp(*Head*, *Pre*$_i$, *Comp*$_i$), for $i = 1, \ldots, n$, that express properties of the whole computation for calls to *Head* if *Pre*$_i$ is satisfied.

Assertions also provide the user with information that it is not available to the analyzer for different reasons. In our concrete case we are going to receive a file with multiple assertions explaining the result of that analysis.

Example 7. *Consider the following implementation of* **`append`**

```
:- module(_, [app/3], [assertions]).

:- entry app(A,B,C) : (list(A), list(B)).
:- pred app(A,B,C) : (list(A), list(B)) => var(C).
app([],Y,Y).
app([X|Xs], Yss, [X|Zs]) :-
  app(Xs,Ys,Zs).
```

Running the **`CiaoPP`** *analyzer we obtain the following file:*

[1] We denote the calling conditions with calls (plural) for historic reasons, and to avoid confusion with the higher order predicate in Prolog `call/2`.

```
1   :- module(_1,[app/3],[assertions,nativeprops,regtypes]).
2
3   :- entry app(A,B,C)
4   : ( list(A), list(B) ).
5
6   :- checked calls app(A,B,C)
7   : ( list(A), list(B) ).
8
9   :- false success app(A,B,C)
10  : ( list(A), list(B) )
11  => var(C).
12
13  :- true pred app(A,B,C)
14  : mshare([[A],[A,B],[A,B,C],[A,C],[B],[B,C],[C]])
15  => mshare([[A,B,C],[A,C],[B,C]]).
16
17  :- true pred app(A,B,C)
18  : ( list(A), list(B), term(C) )
19  => ( list(A), list(B), list(C) ).
20
21   app([],Y,Y).
22   app([X|Xs], Yss, [X|Zs]) :-
23   app(Xs,Ys,Zs).
```

Notice that in line 9 we have a false, that means (quite obvious) that the assertion is wrong. The correct one is obtained changing var(C) to list(C).

3.2 Abstract Interpretation with Ciao

In order to explain how abstract interpretation is done in Ciao we need to briefly explain how resolution works on a CLP program introducing the notion of AND trees and generalized AND trees in which the abstract interpretation is based. We follow [50, 51] and [52] for the concrete semantics and [53, 54, 55, 56] for the analysis framework.

Definition 15 (*substitution*)**.** *A* substitution *is a set* $\theta = \{V_1/t_1, \ldots, V_n/t_n\}$ *with* V_i *distinct variables and* t_i *terms* $i \in \{1 \ldots n\}$.

We say that t_i is the *value* of V_i in θ. The set $\{V_1, \ldots, V_n\}$ is the *domain* of θ; the *range* is the set of variables appearing in $t_1, \ldots, t_n$. By *vars*(t) we denote the set of variables occurring in t, by *vars*(θ) the union of the domain and the range of θ. The composition of two substitutions η and θ is denoted by $\eta\theta$.

A *resolvent* is represented by $\leftarrow (A_1, \ldots, A_n)\eta_i$ where $\eta_i = \theta_1 \ldots \theta_i$ is the composition of the substitutions applied so far. For the initial resolvent (the query) we have $\eta_0 = \epsilon$, which denotes the empty substitution.

In order to perform a logical inference the computation rule select the leftmost literal $A_1\eta_i$. The search rule then selects a clause $H \leftarrow (B_1, \ldots, B_m)$, renames it, and unifies the head H with $A_1\eta_i$. If the unification is successful with most general unifier θ_{i+1}, then the resolvent $\leftarrow (B_1, \ldots, B_m, A_2, \ldots, A_n)\eta_{i+1}$ is derived with $\eta_{i+1} = \eta_i\theta_{i+1}$. Resolutions are usually presented as proof trees usually, also referred to as AND trees, where literals in the resolvent are the leaves of the tree.

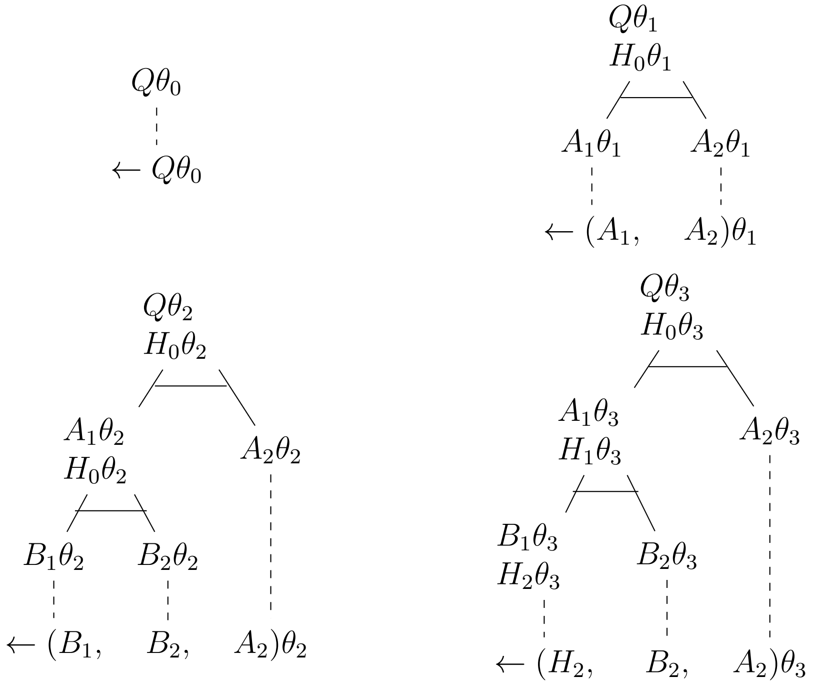

Figure 3.3: Some successive AND trees. Figure adapted from [52].

The set of all AND trees which can originate from a given set of queries specifies completely the procedural behavior of a program for that set of queries.Figure 3.3 illustrates this representation of proof states (i.e., resolvents) as AND trees, and program execution as a sequence of such states.

Generalized AND trees. An AND tree contains more information than needed for analysis purposes; it shows the whole state of the computation (all variables). To characterize the procedural behavior, it suffices to know the instance of procedure calls immediately before their execution and immediately after their completion. This is why [52] suggests representing a sequence of successive AND trees using a *generalized* AND *tree*. These trees are built starting with an initial node, which is the query Q, adorned on the left with a substitution θ, and with the domain of θ a subset of $vars(Q)$. θ is then called the *call substitution* of Q, and the rest of the tree is built by expanding the leaf nodes in the following way:

- If L is a leaf adorned on the left with the call substitution θ_i, then:
  - If C is a properly renamed clause, $H \leftarrow B_1, \ldots, B_n$, defining L, then the tree is expanded by pairing L with H and adding the calls $B_1, \ldots, B_n$ as children of L. B_1 is adorned on the left with the call substitution θ_{i+1}. The domain of θ_{i+1} is $vars(H \leftarrow B_1, \ldots, B_n)$. If the clause C is a fact, θ_{i+1} adorns an empty body and is also the *success substitution* of the body.
  - If L is a built-in, then the tree is expanded by adorning L on the right with a substitution θ_{i+1}. The domain of θ_{i+1} is $vars$(clause of L). θ_{i+1} is the success

substitution of L. With L the last call of its clause, θ_{i+1} is also the success substitution of the body; otherwise it is the call substitution of the next call.

- If a node (call) L is adorned on the left with a call substitution but not adorned with a substitution on the right, such that L is the parent of a clause body with success substitution θ_i. The tree is expanded by adorning L on the right with a substitution θ_{i+1}. The domain of θ_{i+1} is $vars$(clause of L). θ_{i+1} is the success substitution of L. With L the last call of its clause (the query), θ_{i+1} is also the success substitution of the body (the query); otherwise it is the call substitution of the next call.

To obtain the whole tree of a successful SLD derivation of a program the previous steps must be repeated until the root node of the tree is adorned on the right with the success substitution. Note that for a given query a number of trees may exist, considering the different clauses that may be unifiable.

Generalized AND trees are specially convenient for analysis. Accumulated substitutions in a concrete domain, can result in having an unbounded number of variables. Also, it becomes easy to compare substitutions adorning different instances of the same clause, something which is essential for the treatment of recursive clauses. The only additional requirement is that procedure exit must recover a different restriction of the same accumulating substitution.

It is also often interesting to consider trees with also OR nodes, i.e., AND-OR *trees*, rather than considering sets of AND trees. AND-OR *trees* capture the semantics in a useful way for analyses such as determinacy [57, 58], cardinality [59] and non-failure [60], among others.

Analysis Graphs. The abstract interpretation performed by Ciao is *query-dependant* and its result is an *abstraction* of the generalized AND tree semantics. The Ciao analyzer is based on the PLAI algorithm [53, 54, 55, 56]. The purpose of this process is to obtain a finite object (an *analysis graph*) that abstracts (safely approximates) the (possibly infinite) set of (possibly infinite) generalized AND trees in the execution of a CLP program. The input to the abstract interpretation process is the program P, an abstract domain D_α , and a set of initial *abstract queries* $Q_\alpha = \{\langle A_i, \lambda_i^c \rangle\}$, where each A_i is a normalized atom, and $\lambda_i^c \in D_\alpha$. Q_α defines the (typically infinite) set of concrete queries Q such that the analysis must be correct for, $[\![P]\!]_Q = [\![P]\!]_{\gamma(Q_\alpha)}$.

4

Floating Point Interval Domain

Definition 16 (Floating point interval). *Let $\mathbb{F} = \mathbb{F}(p, e_{max})$. The set of floating-point intervals over $\mathbb{F}$ is:*

$$\mathscr{I}_{\mathbb{F}} = \{\emptyset\} \cup \{[l, u] \,|\, l, u \in \mathbb{F},\, l \preceq u\} \tag{4.1}$$

Where $[l, u]$ denotes the set $\{x \in \mathbb{F} \,|\, l \preceq x \preceq u\}$.

Proposition 4. *$\mathscr{I}_{\mathbb{F}}$ is a bounded meet-semmilatice with the set intersection.*

<u>Proof:</u>. *Since $\mathbb{F}$ is a partially ordered set by $\preceq$, defined in Definition 2, and the intersection of two sets always exists if $\emptyset$ is in the space, $\mathscr{I}_{\mathbb{F}}$ is a semmilatice. And it is bounded by $\emptyset$ and $[-\infty, +\infty]$.*

Working with floating-point intervals allows us to capture numbers that are not in $\mathbb{F}$.

4.1 Rounding modes

As we saw in 3 the IEEE 754 standard introduces different rounding operators and each user can choose the preffered one at each moment. The rounding mode used affects the results of computations and must be taken into account during the analysis. In this section we are going to present the rounding mode selector used in our analysis. We also prove that our selection of lower and upper round modes satisfies $x \circ_{r_l} y \preceq x \circ y \preceq x \circ_{r_u} y$.

Definition 17. *Let $\mathbb{F}(p, e_{max}) = \mathbb{F}$ any floating-point format and let $S \subseteq \{\uparrow, \downarrow, n, 0\}$ the set of rounding modes. Let $y, z \in \mathbb{F}$ and $\circ \in \{+, -, /, \cdot\}$ such that $\circ \neq /$ or $z \neq 0$. Then* *we define the lower and upper rounding mode as:*

$$r_l(S, y, \circ, z) = \begin{cases} \downarrow & \mathit{if} \ \downarrow \in S \\ \downarrow & \mathit{if}\ 0 \in S \ \mathit{and}\ y \circ z > 0 \\ n & \mathit{if}\ n \in S \\ \uparrow & \mathit{otherwise} \end{cases}$$

$$r_u(S, y, \circ, z) = \begin{cases} \uparrow & \mathit{if} \ \uparrow \in S \\ \uparrow & \mathit{if}\ 0 \in S \ \mathit{and}\ y \circ z \leq 0 \\ n & \mathit{if}\ n \in S \\ \downarrow & \mathit{otherwise} \end{cases}$$

The lower rounding mode selector is implemented as follows in our domain.

```
low_round(S, _, _, _, '$down') :- in_list('$down', S), !.
low_round(S, Y, OP, Z, '$down') :-
    in_list('$\$$zero', S), op_calc(Y,OP,Z , X), X > 0, !.
low_round(S, _, _, _, '$n') :-
    in_list('$n',S), !.  low_round(_, _, _, _, '$up') :- !.
```

Theorem 1. *Let $\mathbb{F}(p, e_{max}) = \mathbb{F}$ any floating-point format and let $S \subseteq \{\uparrow, \downarrow, n, 0\}$ the set of rounding modes, $y, z \in \mathbb{F}$ and $\circ \in \{+, -, /, \cdot\}$ such that $\circ \neq /$ or $z \neq 0$. Let also $r_l = r_l(S, y, \circ, z)$ and $r_u = r_u(S, y, \circ, z)$. Then for all $r \in S$:*

$$y \circ_{r_l} z \preceq y \circ z \preceq y \circ_{r_u} z$$

There also exist $r', r'' \in S$ such that $y \circ_{r_l} z = y \circ_{r'} z$ and $y \circ_{r_u} z = y \circ_{r''} z$

<u>Proof:</u>. *First notice that for each $x, y, z \in \mathbb{F}$ we have that $[y \circ z]_n = [y \circ z]_\downarrow$ or we have that $[y \circ z]_n = [y \circ z]_\uparrow$ So let us prove that $y \circ_\downarrow z \preceq y \circ_n z \preceq y \circ_\uparrow z$ considering the possible cases:*

$y \circ_\mathbb{R} z = \pm\infty$ in this case $y \circ_\downarrow z = y \circ_n z = y \circ_\uparrow z$ and $y \circ_\downarrow z \preceq y \circ_n z \preceq y \circ_\uparrow z$.

$y \circ_\mathbb{R} z \leq -f_{min} \vee y \circ_\mathbb{R} z \geq f_{min}$ in this case by Proposition 3 we have that $[y \circ_\mathbb{R} z]_\downarrow \leq [y \circ_\mathbb{R} z]_n \leq [y \circ_\mathbb{R} z]_\uparrow$. And since $y \circ_r z \neq 0$ and $[y \circ_\mathbb{R} z]_r = y \circ_r z$ for $r \in S$ it holds that $y \circ_\downarrow z \preceq y \circ_n z \preceq y \circ_\uparrow z$.

$-f_{min} < y \circ_\mathbb{R} z < 0$. In this case $y \circ_\downarrow z = -f_{min}$ and $y \circ_\downarrow z = -0$ and $y \circ_n z \in \{-f_{min}, -0\}$ so $y \circ_\downarrow z \preceq y \circ_n z \preceq y \circ_\uparrow z$.

$0 < y \circ_\mathbb{R} z < f_{min}$. In this case $y \circ_\downarrow z = +0$ and $y \circ_\downarrow z = f_{min}$ and $y \circ_n z \in \{-f_{min}, -0\}$ so $y \circ_\downarrow z \preceq y \circ_n z \preceq y \circ_\uparrow z$.

$y \circ_\mathbb{R} z = 0$ In this case if $\circ_\mathbb{R} \in \{\cdot_\mathbb{R} / _\mathbb{R}\}$ we have that for all rounding modes the result will be ± 0. If $\circ_\mathbb{R} \in \{+_\mathbb{R}, -_\mathbb{R}\}$ we have that $y \circ_\downarrow z \neq -0$ and $y \circ_n z = y \circ_\uparrow = +0$. So for all cases we have that $y \circ_\downarrow z \preceq y \circ_n z \preceq y \circ_\uparrow z$.

Now let us consider what will happen with $\circ_0$. By definition if $y \circ_{\mathbb{R}} z > 0$ we have that $y \circ_0 z = y \circ_{\downarrow} z$. In that case we have: $y \circ_{\downarrow} z = y \circ_0 z \preceq y \circ_n z \preceq y \circ_{\uparrow} z$. And if $y \circ_{\mathbb{R}} z < 0$ we have that $y \circ_0 z = y \circ_{\uparrow} z$ and we have $y \circ_{\downarrow} z \preceq y \circ_n z \preceq y \circ_0 z = y \circ_{\uparrow} z$. Moreover if $y \circ_{\mathbb{R}} z = 0$ and $\circ_{\mathbb{R}} \in \{\cdot_{\mathbb{R}}, /_{\mathbb{R}}\}$ we have that $y \circ_{\downarrow} z = y \circ_0 z = y \circ_n z = y \circ_{\uparrow} z$ and if $\circ \in \{+_{\mathbb{R}}, -_{\mathbb{R}}\}$ we have that $y \circ_{\downarrow} z \preceq y \circ_0 z = y \circ_n z = y \circ_{\uparrow} z$ So this first part of the theorem is proved considering all the possible non empty sets $S \subset \{\uparrow, \downarrow, 0, n\}$

Thanks to this definition we do not have to be concerned about the set of rounding modes as we can always choose a pair of worst-case rounding modes.

4.2 Managing expressions

In this section we are going to show how to manage arithmetic expressions while working with floating point arithmetic. Let $\mathbb{F}(p, e_{max})$, consider the domain of the arithmetical expressions over $\mathbb{F}$:

$$\mathbb{E}_{\mathbb{F}} = x \mid x + y \mid x \cdot y \mid x - y \mid x/y \text{ where } x, y \in \mathbb{F} \tag{4.2}$$

In order to correctly approximate these expressions we consider the following evaluation functions:

Definition 18. *Let $[\![\cdot]\!]_{\downarrow} : \mathbb{E}_{\mathbb{F}} \to \mathbb{F}$ and $[\![\cdot]\!]_{\uparrow} : \mathbb{E}_{\mathbb{F}} \to \mathbb{F}$ two partial functions that for each $x \in \mathbb{F}$ that evaluates on $\mathbb{R}$ to a nonzero value:*

$$[\![x]\!]_{\downarrow} \preceq [x]_{\downarrow}$$

$$[\![x]\!]_{\uparrow} \preceq [x]_{\uparrow}$$

These functions provide an abstraction of evaluation algorithms such that:

- the indicated approximation direction is respected;
- it is precise and practical. This is not trivial: real arithmetic is the most precise but the least practical.

Now our task is, given $x = y \circ z$ where $\circ \in \{+, \cdot, -, /\}$ such that $x \in X$ $y \in Y$ and $z \in Z$, with $X, Y, Z \in \mathscr{I}_{\mathbb{F}}$. Notice that given Y, Z we always know that $X = \top$ is a valid interval and our implementation always considers this case. Our aim is to infer an interval $X' = [x_l, x_u]$ narrower than X. If the values of x are restricted, we intersect the obtained interval with the constraints $x \geq x_l$ and $x \leq x_u$.

Definition 19 (Direct propagation correctness)**.**

$$\forall r \in S, y \in Y, z \in Z : x = y \circ_r z \Rightarrow x \in X' \tag{4.3}$$

Notice that this means that $X' \subseteq X$, which in our case is trivial since $X = \top$. We also want to achieve optimality of the direct propagation, that means that:

$$\forall X'' \subset X' : \exists r \in S, y \in Y, z \in Z \text{ s.t. } y \circ z \notin X'' \tag{4.4}$$

On the other hand we can also define the equivalent properties for the inverse propagation. We aim to develop the presented tool further including this kind of propagation in order to be able to, given the constraint $x = y \circ z$, infer a narrower interval for y or z. This property is formalized as follows:

$$\forall r \in S, x \in X, z \in Z : x = y \circ_r z \Rightarrow y \in Y' \tag{4.5}$$

In this case we can hope to determine some smallest interval, achieving some similar optimality property:

$$\forall Y'' \subset Y : \exists r \in S, y \in Y' \ Y'', z \in Z \text{ s.t. } y \circ_r z \notin X \tag{4.6}$$

4.3 Algorithms for Arithmetic Constraints

In this section we are going to present algorithms that are able to obtain and *optimal* direct projection. This algorithms are based in the work of [1]. The direct projection has been developed within `CiaoPP` and has been used to analyze some programs. These algorithms are able to deal with any set of rounding modes. As we stated before, we are going to only consider the addition, the product, and the division for simplicity.

4.3.1 Addition

In this case our target is to find a narrower interval $X = [x_l, x_u]$ such that $\forall y \in Y, z \in Z, x = y + z \Rightarrow x \in X$. Where $Y = [y_l, y_u], Z = [z_l, z_u]$. The algorithm is straightforward, we select both the lower rounding mode and the upper rounding mode and we compute the addition rounding up and the addition rounding down. This is done using the predicates da_l and da_u which are obtained by inspection of 4.1 and 4.2. Most of the values of this table can be derived from the definition of the addition operation in [6].
The following algorithm implements the direct projection addition.

```
In this case the implementation is straightforward:
add_float_intervals_(S, i(Yl,Yu), i(Zl, Zu), Res) :-
    low_round(S,Yl,'$+',Zl,Rl),
    up_round(S,Yu,'$+',Zu,Ru),
    dal(Yl, Zl, Rl, Xl_),
    dau(Yu, Zu, Ru, Xu_),
    norm_interval(i(Xl_, Xu_), Res).
```

Theorem 2. *The Direct projection Algorithm 3 is correct and optimal.*

Algorithm 1 Direct projection for addition

```
Require: $x = y + z, y \in [y_l, y_u], z \in [z_l, z_u]$
Ensure: $X' = [x_l, x_u]$ s.t $x = y + z \Rightarrow x \in X'$
Ensure: $\forall X'' \subset X' : \exists r \in S, y \in Y, z \in Z$ s.t. $y +_r z \notin X'$
  $r_l := r_l(S, y_l, +, z_l); r_u := r_u(S, y_u, +, z_u)$
  $x'_l := da_l(y_l, z_l, r_l); x'_u := da_u(y_u, z_u, r_u)$
  $X' := [x'_l, x'_u]$
```

$da_l(y_l, z_l, r_l)$	$-\infty$	$\mathbb{R}_-$	-0	$+0$	$\mathbb{R}_+$	$+\infty$
$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$+\infty$
$\mathbb{R}_-$	$-\infty$	$y_l +_{r_l} z_l$	y_l	y_l	$y_l +_{r_l} z_l$	$+\infty$
-0	$-\infty$	z_l	-0	a_{da_l}	z_l	$+\infty$
$+0$	$-\infty$	z_l	a_{da_l}	$+0$	z_l	$+\infty$
$\mathbb{R}_+$	$-\infty$	$y_l +_{r_l} z_l$	y_l	y_l	$y_l +_{r_l} z_l$	$+\infty$
$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$

Table 4.1: Direct lower projection for addition.

Where

$$a_{da_l} = \begin{cases} -0 & \text{if } r_l = \downarrow \\ +0 & \text{otherwise} \end{cases} \tag{4.7}$$

Proof:. *Given the constraint $x = y + z$ we know that $\forall y \in Y, z \in Z$ and $r \in S$ $y +_{r_l} z \leq y +_r z \leq y +_{r_u} z$. Since the addition is a monotonic application we have that:*

$$y_l +_{r_l} z_l \leq y +_{r_l} z \leq y +_r z \leq y +_{r_u} z \leq y_u +_{r_u} z_u$$

From tables 4.1 and 4.2 we can obtain a lower bound for $y_l +_{r_l} z_l$ and an upper bound for $y_u +_{r_u} z_u$ that correspond with da_u and da_l.
Now we have to prove that the addition is optimal, this is $\forall X'' \subset X', \exists r \in S, y \in Y, z \in Z s.t. y +_r z \notin X''$.
Let us consider the lower bound x'_l, we aim to prove that $\exists r \in S$ such that $x'_l = y_l +_r z_l$. Consider following cases:

> *$y_l \notin \mathbb{R}_- \cap \mathbb{R}_+$ or $z_l \notin \mathbb{R}_- \cap \mathbb{R}_+$ In this case can be verify (by brute force) that $da_l(y_l, z_l, r_l) = y_l +_r z_l$.*
>
> *When $y_l \in \mathbb{R}_- \cap \mathbb{R}_+$ and $z_l \in \mathbb{R}_- \cap \mathbb{R}_+$ we have by definition of $da_l(y_l, z_l, r_l)$ that $x'_l = y_l +_{r_l} z_l$*

Since $y_l \in Y$ and $z_l \in Z$ we can conclude that $\forall X'' \subset X', x'_l \notin X''$ implies that $y_l +_r z_l$ for some $r \in S$. The same reasoning can be done for x_u concluding this proof.

$da_u(y_u, z_u, r_u)$	$-\infty$	$\mathbb{R}_-$	-0	$+0$	$\mathbb{R}_+$ $+\infty$	
$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
$\mathbb{R}_-$	$-\infty$	$y_u +_{r_u} z_u$	y_u	y_u	$y_u +_{r_u} z_u$	$+\infty$
-0	$-\infty$	z_u	-0	a_{da_u}	z_u	$+\infty$
$+0$	$-\infty$	z_u	a_{da_u}	$+0$	z_u	$+\infty$
$\mathbb{R}_+$	$-\infty$	$y_u +_{r_u} z_u$	y_u	y_u	$y_u +_{r_u} z_u$	$+\infty$
$+\infty$	$-\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$

Table 4.2: Direct upper projection for addition.

Where

$$a_{da_u} = \begin{cases} -0 & \text{if } r_u = \downarrow \\ +0 & \text{otherwise} \end{cases} \tag{4.8}$$

4.3.2 Product

Here we want to find a narrower interval $X = [x_l, x_u]$ such that $\forall y \in Y, z \in Z, x = y \cdot z \Rightarrow x \in X$. Where $Y = [y_l, y_u], Z = [z_l, z_u]$. In this case deriving an optimal and correct algorithm is not straightforward as in the case of the addition. First when the sign of y_l is not equal to the sign of y_u and neither the sign of z_l is equal to the sign z_u there is not a unique choice for the extrema, that we will denote by (y_L, y_U, z_L, z_U), so we have to calculate all the options and select among them. In the rest of the cases the selection of the extrema can be done using the σ function that it is defined as follows:

$$\sigma(y_l, y_u, w_l, w_u) = \begin{cases} (y_l, y_u, w_l, w_u) & \text{if } sgn(w_l) = sgn(y_l) = 1 \\ (y_u, y_l, w_l, w_u) & \text{if } sgn(y_l) = -1 \wedge sgn(w_u) = 1 \\ (y_u, y_u, w_l, w_u) & \text{if } sgn(y_l) = sgn(w_u) = 1 \wedge sgn(w_l) = -1 \\ (y_u, y_l, w_u, w_l) & \text{if } sgn(y_u) = sgn(w_u) = -1 \\ (y_l, y_u, w_u, w_l) & \text{if } sgn(y_u) = -1 \wedge sgn(w_l) = 1 \\ (y_l, y_l, w_u, w_l) & \text{if } sgn(y_u) = sgn(w_l) = -1 \wedge sgn(w_u) = 1 \end{cases}$$

assuming that $sgn(y_l) = sgn(y_u)$

Theorem 3. *The Direct projection Algorithm 4.3.2 is correct and optimal.*

Proof:. *Given $x = y \cdot z$ with $y \in [y_l, y_u]$ and $z \in [z_l, z_u]$.*

First note that the function σ4.3.2 choose correctly the extrema (y_L, y_U, z_L, z_U) when invoked. In the first case since $sgn(y_l) = sgn(y_u)$ the precondition of the function holds. In the second case $sgn(z_l) = sgn(z_u)$ and the extrema is obtained changing the role of y and z. When $sgn(y_l) \neq sgn(y_u)$ and $sgn(z_l) \neq sgn(z_u)$ we have that $sgn(y_l) = sgn(z_l) = -1$ and $sgn(y_u) = sgn(z_u) = 1$. In this case to consider the left side of the interval X' we have to take into account that $sgn(x'_l) = -1$ so in order to keep it correct we need to choose the smallest one between the product of y_L and z_l and z_u. We must have same consideration when computing x'_u.

Algorithm 2 Direct projection for multiplication

```
Require: $x = y \cdot z, y \in [y_l, y_u], z \in [z_l, z_u]$
Ensure: $X' = [x'_l, x'_u] \wedge x = y \cdot z \Rightarrow x \in X'$
Ensure: $\forall X'' \subset X' \exists r \in S$ s.t. $y \cdot_r z \notin X''$
  if $sgn(y_l) \neq sgn(y_u) \wedge sgn(z_l) \neq sgn(z_u)$ then
    $(y_L, y_U, z_L, z_U) := (y_l, y_l, z_u, z_l)$
    $r_l := r_l(S, y_L, \cdot, y_L) : r_u := r_u(S, y_U, \cdot, z_U)$
    $v_l := dm_l(y_L, z_L, r_l); v_u := dm_u(y_U, z_U, r_U)$
    $(y_L, y_U, z_L, z_U) := (y_u, y_u, z_l, z_u)$
    $w_l := dm_l(y_L, z_L, r_l); w_u := dm_u(y_U, z_U, r_U)$
    $x'_l := min\{v_l, w_l\}; x'_u := min\{v_u, w_u\}$
  else
    if $sgn(y_l) = sgn(y_u)$ then
      $(y_L, y_U, z_L, z_U) := \sigma(y_l, y_u, z_l, z_u)$
    else
      $(y_L, y_U, z_L, z_U) := \sigma(z_l, z_u, y_l, y_u)$
    end if
    $r_l := r_l(S, y_L, \cdot, y_L) : r_u := r_u(S, y_U, \cdot, z_U)$
    $x'_l := dm_l(y_L, z_L, r_l); x'_u := da_u(y_U, z_U, r_u)$
  end if
  $X' := [x'_l, x'_u]$
```

By 1 and as we made in the proof of 2 we have to find a lower bound for $y_L \cdot_{r_l} z_L$ and an upper one for $y_U \cdot_{r_U} z_U$. As in the previous proofs this bounds are given by the tables 4.3 and 4.4. Let us analyze some of the cases in the table. Consider the case $y_L = -\infty$ and $z_L = 0$. From a first Calculus course it is known that $\infty \cdot 0$ is an indeterminate form, moreover the IEEE754 standard considers this product as an invalid operation. But we can not return NaN since that would not be a proper interval and we cannot return $\perp$ since that could not allow us to perform a proper analysis. First note that by 4.3.2 $y_L = -\infty \Rightarrow y_l = -\infty$. Since $y_l = -\infty$ we have two cases concerning y_u:

1. *$y_u \geq -f_{max}$, in which case we have that $z \cdot 0 = -0 \; \forall z \in \mathbb{F}_-$ and would be 0 in the positive case.*

2. *$y_u = -\infty$, in this case by 4.3.2 $z_L = z_u$ and $\forall z < 0, z \cdot -\infty = \infty$.*

So we have that -0 is a correct lower bound.

Let us prove that the obtained interval X is optimal ($\forall X'' \subset X', \exists r \in S, y \in Y, z \in Zs.t.y \cdot_r z \notin X''$). Consider the lower bound x_l, let us proof that there exists $r \in S, y \in Y, z \in Z$ such that $x'_l = y \cdot_r z$. By 4.3.2 we know that $dm_l(y_L, z_L, r_l) = x'_l$ for some y_L, z_L, r_l whose existence is verified by the algorithm. Since $y_L \in Y, z_L \in Z$ if $x'_l \notin X''$ we know by 1 that $\exists r' \in S, s.t.y \cdot_r z = y \cdot_{r'} z$ so we can conclude that $\exists r \in S$ such that $y_l \cdot_r z_l \notin X''$. The same can be proven in the case of x'_u concluding the proof.

$dm_l(y_l, z_l, r_l)$	$-\infty$	$\mathbb{R}_-$	-0	$+0$	$\mathbb{R}_+$	$+\infty$
$-\infty$	$+\infty$	$+\infty$	$+\infty$	-0	$-\infty$	$-\infty$
$\mathbb{R}_-$	$+\infty$	$y_l \cdot_{r_l} z_l$	$+0$	-0	$y_l \cdot_{r_l} z_l$	$-\infty$
-0	$+\infty$	$+0$	$+0$	-0	-0	-0
$+0$	-0	-0	-0	$+0$	$+0$	$+\infty$
$\mathbb{R}_+$	$-\infty$	$y_l \cdot_{r_l} z_l$	-0	$+0$	$y_l \cdot_{r_l} z_l$	$+\infty$
$+\infty$	$-\infty$	$-\infty$	-0	$+\infty$	$+\infty$	$+\infty$

Table 4.3: Direct lower projection for multiplication.

$dm_u(y_u, z_u, r_u)$	$-\infty$	$\mathbb{R}_-$	-0	$+0$	$\mathbb{R}_+$	$+\infty$
$-\infty$	$+\infty$	$+\infty$	$+0$	$-\infty$	$-\infty$	$-\infty$
$\mathbb{R}_-$	$+\infty$	$y_u \cdot_{r_u} z_u$	$+0$	-0	$y_u \cdot_{r_u} z_u$	$-\infty$
-0	$+0$	$+0$	$+0$	-0	-0	$-\infty$
$+0$	$-\infty$	-0	-0	$+0$	$+0$	$+0$
$\mathbb{R}_+$	$-\infty$	$y_u \cdot_{r_u} z_u$	-0	$+0$	$y_u \cdot_{r_u} z_u$	$+\infty$
$+\infty$	$-\infty$	$-\infty$	$-\infty$	$+0$	$+\infty$	$+\infty$

Table 4.4: Direct upper projection for multiplication.

The product of floating point intervals is implemented as follows:

```
product_floats_intervals(S, i(Yl_, Yu_), i(Zl_, Zu_), i(Xl, Xu)) :-
  sgn(Yl_) \= sgn(Yu_), sgn(Zl_) \= Zu_, !,
  Yl = Yl_,
  Yu = Yl_,
  Zl = Zu_,
  Zu = Zl_,
  low_round(S,Yl,'$*',Zl,Rl),
  up_round(S,Yu,'$*',Zu,Ru),
  dml(Yl, Zl, Rl, Vl),
  dmu(Yu, Zu, Ru, Vu),
  Yl__ = Yu_,
  Yu__ = Yu_,
  Zl__ = Zl_,
  Zu__ = Zu_,
  dml(Yl__, Zl__, Rl, Wl),
  dmu(Yu__, Zu__, Ru, Wu ),
  minf(Vl, Wl, Xl),
  maxf(Vu, Wu, Xu).
product_floats_intervals(S, i(Yl, Yu), i(Zl, Zu), i(Xl, Xu)) :-
  sgn(Yl) == sgn(Yu), !,
  sigma(Yl, Yu, Zl, Zu, Yl_, Yu_, Zl_, Zu_),
  low_round(S,Yu_,'$*',Zu_,Rl),
  up_round(S,Yl_,'$*',Zl_,Ru),
  dml(Yl_, Zl_, Ru, Xl),
  dmu(Yu_, Zu_, Rl, Xu).
product_floats_intervals(S, i(Yl, Yu), i(Zl, Zu), i(Xl, Xu)) :-
  sgn(Zl) == sgn(Zu), !,
  sigma(Zl, Zu, Yl, Yu, Zl_, Zu_, Yl_, Yu_),
  low_round(S,Yl_,'$*',Zl_,Rl),
  up_round(S,Yu_,'$*',Zu_,Ru),
  dml(Yl_, Zl_, Rl, Xl),
  dmu(Yu_, Zu_, Ru, Xu).
```

Example 8. *Consider the intervals* $Y = [-0.0, 2.0], Z = [+0.0, 1.0]$ *and the rounding set* $\{\uparrow, \downarrow\}$*. Let us consider* $x = y \cdot z$ *In this case* $sgn(-0.0) \neq sgn(2.0) \wedge sgn(+0.0) = sgn(-0.0))$ *so we are in the third case and* $\sigma(+0.0, 1.0, -0.0, 2.0) = (1.0, 1.0, -0.0, 2.0)$ *In this case we have* $r_l = \downarrow, r_u = \uparrow$ *and the direct projections are* $dm_l(1.0, -0.0, \downarrow) = -0.0, dm_u(1.0, 2.0, \uparrow) = 2.0$*. If we consider our implementation we get:*

```
ciaopp ?- product_floats_intervals(['up', 'down'], i(-0.0, 2.0), i(0.0, 1.0), X).

X = i(-0.0,2.0) ?

yes
```

4.3.3 Division

Here we want to find a narrower interval $X = [x_l, x_u]$ such that $\forall y \in Y, z \in Z, x = y/z \Rightarrow x \in X$. Where $Y = [y_l, y_u], Z = [z_l, z_u]$. In this case the algorithm is much more complex than before. In this case we need to consider the positive and the negative part of the intervals. Also it is important to consider if it is possible to have zeros in the intervals. Once the division is considered for each positive and negative part the resulting interval is the convex union of both intervals. In this case the selection of the extrema can be done using the τ function that it is defined as follows:

$$\tau(y_l, y_u, w_l, w_u) = \begin{cases} (y_u, y_l, w_l, w_u) & \text{if } sgn(w_u) = sgn(y_u) = -1 \\ (y_u, y_l, w_u, w_l) & \text{if } sgn(w_u) = -1 \wedge sgn(y_l) = 1 \\ (y_u, y_l, w_u, w_u) & \text{if } sgn(w_u) = sgn(y_l) = -1 \wedge sgn(y_u) = 1 \\ (y_l, y_u, w_l, w_u) & \text{if } sgn(y_u) = -1 \wedge sgn(w_l) = 1 \\ (y_l, y_u, w_u, w_l) & \text{if } sgn(w_l) = sgn(y_l) = 1 \\ (y_l, y_u, w_l, w_l) & \text{if } sgn(y_u) = sgn(w_l) = 1 \wedge sgn(y_l) = -1 \end{cases}$$

assuming that $sgn(w_l) = sgn(w_u))$

Theorem 4. *The Direct projection Algorithm 4.3.3 is correct and optimal.*

__Proof:__. *As was done in 3 it can be proved that* y_L, y_U *and* w_L, w_U *computed via* τ *4.3.3 are correct bounds for the intervals* Y *and* W*. Notice that* $\forall w_1, w_2 \in W, sgn(w_1) = sgn(w_2)$ *by its definition in 4.3.3. Again by 1 and as we have done in previous proofs we can focus on finding a lower bound for* $y_L/_{r_l}w_L$*and an upper one for* $y_U/_{r_u}z_U$*. These bounds are computed by* dd_l *4.5 and* dd_u *4.6 respectively.*
As we have done in 3 with dm_l *let us comment some of the indeterminate forms that can happen. Consider* $y_L = w_L = -\infty$*. Again is well known that* $-\infty/-\infty$ *it is an indeterminate form which, once again, it is not allowed by the IEEE 754 standard. In this case by* τ *4.3.3 we have that* $y_L = y_u = -\infty$ *and* $w_L = w_L$*. By the IEEE 754 standard dividing* $-\infty/-\infty$ *it is not allowed but dividing* $-\infty$ *by any negative (and finite) number leads to* ∞ *so we can conclude that* $x_l = \infty$ *is a correct lower bound.*
Now let us prove the optimality of our algorithm. Let us consider the lower bound computed for the positive part, x_l^+*. In a similar way than before we aim to prove that*

Algorithm 3 Direct projection for division

```
Require: $x = y/z, y \in [y_l, y_u], z \in [z_l, z_u]$
Ensure: $X' = [x'_l, x'_u] \wedge x = y/z \Rightarrow x \in X'$
Ensure: $\forall X'' \subset X' \exists r \in S$ s.t. $y/_r z \notin X''$
  $Z_- = [z_l^-, z_u^+] := Z \cap [-\infty, -0]$
  if $Z_- \neq \emptyset$ then
    $W := Z_-$
    $(y_L, y_U, w_L, w_U) := \tau(y_l, y_u, w_l, w_u)$
    $r_l := r_l(S, y_L, /, w_L), r_u := r_u(S, y_U, /, w_U)$
    $x_l^- := dd_l(y_L, w_L, r_l); x_u^- := dd_u(y_U, w_U, r_u)$
  else
    $X_- = [x_l^-, x_u^-] := \emptyset$
  end if
  $Z_+ = [z_l^+, z_u^+] := Z \cap [+0, \infty]$
  if $Z_+ \neq \emptyset$ then
    $W := Z_+$
    $(y_L, y_U, w_L, w_U) := \tau(y_l, y_u, w_l, w_u)$
    $r_l := r_l(S, y_L, /, w_L), r_u := r_u(S, y_U, /, w_U)$
    $x_l^+ := dd_l(y_L, w_L, r_l); x_u^+ := dd_u(y_U, w_U, r_u)$
  else
    $X_+ = [x_l^+, x_u^+] := \emptyset$
  end if
  $X' := X_+ \uplus X_-$
```

$dd_l(y_l, z_l, r_l)$	$-\infty$	$\mathbb{R}_-$	-0	$+0$	$\mathbb{R}_+$	$+\infty$
$-\infty$	$+\infty$	$+\infty$	$+\infty$	$-\infty$	$-\infty$	-0
$\mathbb{R}_-$	$+0$	$y_l/_{r_l}z_l$	$+\infty$	$-\infty$	$y_l/_{r_l}z_l$	-0
-0	$+0$	$+0$	$+\infty$	-0	-0	-0
$+0$	-0	-0	-0	$+\infty$	$+0$	$+0$
$\mathbb{R}_+$	-0	$y_l/_{r_l}z_l$	$-\infty$	$+\infty$	$y_l/_{r_l}z_l$	$+0$
$+\infty$	-0	$-\infty$	$-\infty$	$+\infty$	$+\infty$	$+0$

Table 4.5: Direct lower projection for division.

if $[x_l^+, x_u^+] \neq \emptyset$ then there exists some $r \in S, y \in Y, z \in Z$ such that $y/_r z = x_l^+$. By 4.3.3 we know that there exist $y_L, w_L = z_L$ and r_l such that $dd_l(y_L, w_L, r_l) = x_l^+$ and their existence is guaranteed by the algorithm and by Theorem 1 we know that $\exists r \in S$ such that $y_L/_{r_l}w_L = y_L/_r w_L$. With this we can conclude that $x_l^+ \notin X'' \Rightarrow y_L/_r w_L \notin X''$. The same result holds for x_u^+, x_l^- and x_u^- so we can conclude that the algorithm is optimal.

Example 9. *Consider the intervals $Y = [-0.0, 3.0], Z = [-1.0, 1.0]$ and any set of rounding modes. First we have to consider both intervals $Z_- = [-1.0, -0.0]$ and*

$dd_u(y_u, z_u, r_u)$	$-\infty$	$\mathbb{R}_-$	-0	$+0$	$\mathbb{R}_+$	$+\infty$
$-\infty$	$+0$	$+\infty$	$+\infty$	$-\infty$	$-\infty$	-0
$\mathbb{R}_-$	$+0$	$y_u/_{r_u}z_u$	$+\infty$	$-\infty 0$	$y_u/_{r_u}z_u$	-0
-0	$+0$	$+0$	$+\infty$	-0	-0	-0
$+0$	-0	-0	-0	$+\infty$	$+0$	$+0$
$\mathbb{R}_+$	-0	$y_u/_{r_u}z_u$	$-\infty$	$+\infty$	$y_u/_{r_u}z_u$	$+0$
$+\infty$	$-\infty$	$-\infty$	$-\infty$	$+\infty$	$+\infty$	$+0$

Table 4.6: Direct upper projection for division.

$Z_+ = [+0.0, 1.0]$. In the case of the negative part we have that $\tau(-0.0, 3.0, -1.0, -0.0) = (3.0, -0.0, -0.0, -0.0)$ in which case $dd_l(3.0, -0.0, \_) = -\infty$ and $dd_l(-0.0, -0.0, \_) = +0.0$. For the positive part we have $\tau(-0.0, 3.0, +0.0, 1.0) = (-0.0, 3.0, +0.0, +0.0)$ the projections yield to $dd_l(-0.0, +0.0, \_) = -0.0, dd_u(3.0, +0.0, \_) = +\infty$. So the result of this division is $[-\infty, +\infty]$, which is expected since both $-0.0, +0.0 \in Z$. Our algorithm returns:

```
    ciaopp ?- division_float_intervals_(['$zero'], i(-0.0, 3.0), i(-1.0, 1.0), X).
    X = i(-0.Inf,0.Inf) ?
```

The algorithm of the direct projection for the division is implemented as follows:

```
division_float_intervals_(S, i(Yl, Yu), i(Zl, Zu), Res) :-
    div_neg_part(S, i(Yl, Yu), i(Zl, Zu), Bot),
    div_pos_part(S, i(Yl, Yu), i(Zl, Zu), Bot), nonrel_fintervals_bot(Bot), !,
    Res = Bot.
division_float_intervals_(S, i(Yl, Yu), i(Zl, Zu), Res) :-
    div_neg_part(S, i(Yl, Yu), i(Zl, Zu),  i(Zlm, Zum)),
    div_pos_part(S, i(Yl, Yu), i(Zl, Zu), Bot), nonrel_fintervals_bot(Bot), !,
    norm_interval(i(Zlm, Zum), Res).
division_float_intervals_(S, i(Yl, Yu), i(Zl, Zu), Res) :-
    div_neg_part(S, i(Yl, Yu), i(Zl, Zu), Bot), nonrel_fintervals_bot(Bot),
    div_pos_part(S, i(Yl, Yu), i(Zl, Zu),i(Zlp, Zup)) , !,
    norm_interval(i(Zlp,Zup), Res).
division_float_intervals_(S, i(Yl, Yu), i(Zl, Zu), Res) :-
    div_neg_part(S, i(Yl, Yu), i(Zl, Zu),  i(Zlm, Zum)),
    div_pos_part(S, i(Yl, Yu), i(Zl, Zu),i(Zlp, Zup)) , !,
    minf(Zlm, Zlp, Xl),
    maxf(Zum, Zup, Xu),
    norm_interval(i(Xl, Xu), Res).
```

where:

```
div_pos_part(_, _, i(Zl, Zu), Res) :-
    nonrel_finf(Inf),
    finterval_intersection(i(Zl, Zu), i(0.0, Inf), Bot), nonrel_fintervals_bot(Bot), !,
    Res = Bot.
div_pos_part(S, i(Yl, Yu), i(Zl, Zu), Res) :-
    nonrel_finf(Inf),
    finterval_intersection(i(Zl, Zu), i(0.0, Inf), i(Wl, Wu)), !,
    tau(Yl,Yu,Wl,Wu,Yl_,Yu_,Zl_,Zu_),
    low_round(S,Yl_,'$/',Zl_,Rl),
    up_round(S,Yu_,'$/',Zu_,Ru),
    ddl(Yl_, Zl_, Rl, Xlp),
    ddu(Yu_, Zu_, Ru, Xup),
    norm_interval(i(Xlp, Xup), Res).
```

The predicate **div_neg_part** is implemented as **div_pos_part**.

4.4 Widening

For our implementation we decided to use the standard widening operator for Interval Domains defined as follows:

$$X \diamond Y \begin{cases} \bot & \text{if } X = Y = \bot \\ X & \text{if } X = \bot \vee X = \top \\ Y & \text{if } Y = \bot \vee Y = \top \\ [W_0, W_1] & \text{otherwise} \end{cases} \tag{4.9}$$

where $W_0 = min\{x_l, y_l, lub_l\}, W_1 = max\{x_u, y_u, lub_u\}$ and $[lub_l, lub_u] = lub(X, Y)$ is the lower upper bound of X and Y

The implementation of the widening operator is presented in 4.5. Another implementations of the widening operator can be considered and our aim is to develop widening operators able to give better intervals for floating point computations.

4.5 Adding the domain to **CiaoPP**

In this section we are going to briefly explain how this domain is added to `CiaoPP`.

This domain is implemented as a "derived domain" of the the **`nonrel_base`** *non-relational* abstract domain. Derived domains in CiaoPP are implemented through (Ciao specific) syntactic language extensions that allow the composition of generic code, and the specification of common domain interfaces.

The generic **`nonrel_base`** domain binds the general, more complex, domain interface expected by the PLAI fixpoint (e.g., **`call_to_entry/10`** and the other operations, whose description are out of the scope of this text), with a reduced set of abstract domain definitions. Namely, the predicates that define the value lattice (floating-point intervals in our case) and the predicates that define the transfer functions for basic operations relevant for this domain, such as unification, comparisons, and arithmetic operations.

The predicates that define the lattice and their operations are the following:

- Predicates **`top/1`** and **`bot/1`** are the representation of "top" and "bot" in the abstract domain.
- **`var/1`** refers to the abstraction of a free variable in the abstract domain. In this case, we decided to consider these variables as top, so it is **`var('$top')`**,
- **`less_or_equal_elem/3`**, this predicate refers to the order relation in the domain. In this case, we consider the symbolic order relation between floating point numbers (definition 2):

```
less_or_equal_elem(_,Top) :- top(Top), !.
less_or_equal_elem(Bot,_) :- bot(Bot), !.
less_or_equal_elem(i(N0,N1),i(T0,T1)) :-
    leqf(N0, T0),
    leqf(N1, T1).

leqf(_, NInf) :- fneginf(NInf), !, fail.
leqf(NInf, _) :- fneginf(NInf), !.
leqf(Inf, _) :- finf(Inf), !, fail.
leqf(_, Inf) :- finf(Inf), !.
leqf(N1, N2) :-
    compare(C,N1, N2),
    (C = (<) ; C = (=)), !.
```

- `compute_glb_elem/4` and `compute_lub_elem/4`, they compute the greatest lower bound and the lowest upper bound respectively. The implementation is quite similar in both cases,

```
compute_glb_elem(X, Top, X) :- top(Top), !.
compute_glb_elem(Top, X, X) :- top(Top), !.
compute_glb_elem(i(N0,N1), i(T0,T1), X) :-
    maxf(N0,T0,G0), minf(N1,T1,G1), leqf(G0, G1), !,
    X=i(G0,G1).
compute_glb_elem(_, _, B) :-
    bot(B).

compute_lub_elem(Top, _, Top) :- top(Top), !.
compute_lub_elem(_, Top, Top) :- top(Top), !.
compute_lub_elem(Bot, X, X) :- top(Bot), !.
compute_lub_elem(X, Bot, X) :- top(Bot), !.
compute_lub_elem(i(N0,N1), i(T0,T1), I) :-
    minf(N0,T0,G0),
    maxf(N1,T1,G1),
    simplify_felem(i(G0,G1), I).
```

- `widen_elem/4`, this predicate implements the widening operator. Note that this domain does not consider or use any narrowing operator, this is included as future work as we would comment later. The widening is defined as follows following the definition in 4.4:

```
widen_elem(Bot, W, W) :- bot(Bot), !.
widen_elem(W, Bot, W) :- bot(Bot), !.
widen_elem(Top, _, Top) :- top(Top), !.
widen_elem(_, Top, Top) :- top(Top), !.
widen_elem(V1, V2, W) :-
    finterval_num(V1), finterval_num(V2), !,
    compute_lub_elem(V1,V2,W).
widen_elem(V1, V2, W) :-
    compute_lub_elem(V1,V2,Lub),
    finterval_avalue_get_min(Lub,MinLub),
    finterval_avalue_get_max(Lub,MaxLub),
    finterval_avalue_get_min(V1,MinV1),
    finterval_avalue_get_max(V1,MaxV1),
    finterval_avalue_get_min(V2,MinV2),
    finterval_avalue_get_max(V2,MaxV2),
    % if the lower bound lub is smaller than any of lower bounds, widen
    ( ( \+ leqf(MinV1, MinLub) ; \+ leqf(MinV2, MinLub) ) -> fneginf(W0)
    ; W0 = MinLub
    ),
    % if the upper bound lub is bigger than any of the upper bounds, widen
    ( ( \+ leqf(MaxLub, MaxV1) ; \+ leqf(MaxLub, MaxV2) ) -> finf(W1)
    ; W1 = MaxLub
    ),
    simplify_felem(i(W0, W1), W).
```

where the predicate finterval_num/1 checks if the value is a proper value of the numerical domain and simplify_felem/2 just sends the interval $[-\infty, \infty]$ to $\top$ in order to avoid errors.

The next set of predicates defines the abstract transfer functions. In this case these operations work directly with the abstract substitutions required by the general PLAI [61] domain interface, but relies on a few operations provided by nonrel_base (such as get_value_asub/3 or replace_value_asub/4, etc.) abstract the actual representation.

Some of the built-in that are implemented are the following:

- amgu/5, amgu(+AbsInt,+Term1,+Term2,+ASub0,-NASub) computes NASub which is the abstract unification of Term1 and Term2 with ASub0 an abstract substitution representing the state of both terms. In this case, we implemented the predicate amgu/4 which performs Robinson's unification algorithm [62].

```
amgu(T1,T2,ASub0,NASub) :- var(T1),var(T2), !,
    get_value_asub(ASub0,T1,Value1),
    get_value_asub(ASub0,T2,Value2),
    compute_glb_elem(Value1,Value2,Glb),
    replace_value_asub(ASub0,T1,Glb,ASub1),
    replace_value_asub(ASub1,T2,Glb,NASub).
amgu(T1,T2,ASub0,NASub) :- var(T2), !,
    amgu(T2,T1,ASub0,NASub).
amgu(T1,T2,ASub0,NASub) :- var(T1), !,
    abstract_term(T2,ASub0,NVal),
    replace_value_asub(ASub0,T1,NVal,NASub).
amgu(T1,T2,ASub0,NASub) :- functor(T1,F,A), functor(T2,F,A), !,
    T1 =.. [F|Args1],
    T2 =.. [F|Args2],
    amgu_args(Args1,Args2,ASub0, NASub).
amgu(_T1,_T2,_ASub1,ASub2) :-
    bot(ASub2).
```

  Briefly explained, the algorithm checks if any of the term is a variable, in this case it gives to that variable a value based in the abstract substitution (get_value_asub/3) and computes the greatest lower bound with the predicate compute_glb_elem/4. Once this is done the abstract unifier can be obtained. If none of the terms is a variable the abstract unifier can be obtained with the help of the predicate amgu_args/4:

```
amgu_args([],[],ASub, ASub).
amgu_args([A1|As1],[A2|As2],ASub0, NASub) :-
    amgu(A1,A2,ASub0,ASub1),
    amgu_args(As1,As2,ASub1, NASub).
```

- abuiltin/3, which declare the abstract semantics of the rest of *built-in* predicates for this domain. Some examples are the predicate =</2:

```
abuiltin((X =< Y),Call,Succ):-
    abstract_term(X,Call,ValX),
    abstract_term(Y,Call,ValY),
    compute_glb_elem(ValX,ValY,Glb),
    ( bot(Glb) -> % intervals are disjoint
        ( less_or_equal_elem(ValX, ValY) -> Succ = Call
        ; Succ = Glb
        )
    ; finterval_avalue_get_max(ValX,MaxX),
      finterval_avalue_get_max(ValY,MaxY),
      finterval_avalue_get_min(ValX,MinX),
      finterval_avalue_get_min(ValY,MinY),
      minf(MaxX,MaxY,X1),
      maxf(MinX, MinY, Y0),
      NValX0 = i(MinX, X1),
      NValY0 = i(Y0, MaxY),
      simplify_felem(NValX0, NValX),
      simplify_felem(NValY0, NValY),
      replace_value_asub(Call,X,NValX,Succ0),
      replace_value_asub(Succ0,Y,NValY,Succ)
    ).
```

Another example is the `is/2` predicate that performs arithmetic evaluation:

```
abuiltin((X is Y),Call,Succ):-
    ( is_abs_operate(Y,Call,NVal0) ->
        get_value_asub(Call,X,Val0),
        compute_glb_elem(NVal0,Val0,NVal),
        replace_value_asub(Call,X,NVal,Succ)
    ; amgu(X,Y,Call,Succ)
    ).
```

where `is_abs_operate/3` is defined as:

```
is_abs_operate(X, _Call, NVal) :- float(X), !
    NVal = i(X,X).
is_abs_operate(X, Call, Val) :- var(X), !,
    get_value_asub(Call, X, Val).
is_abs_operate(+(X,Y), Call, NVal) :-
    is_abs_operate(X,Call,NXVal),
    is_abs_operate(Y,Call,NYVal),
    add_float_intervals(NXVal, NYVal, NVal_).
is_abs_operate(-(X,Y), Call, NVal) :-
    is_abs_operate(X,Call,NXVal),
    is_abs_operate(Y,Call,NYVal),
    substract_float_intervals(NXVal, NYVal, NVal_).
is_abs_operate(*(X,Y), Call, NVal) :-
    is_abs_operate(X,Call,NXVal),
    is_abs_operate(Y,Call,NYVal),
    product_float_intervals(NXVal, NYVal, NVal_).
is_abs_operate(/(X,Y), Call, NVal) :-
    is_abs_operate(X,Call,NXVal),
    is_abs_operate(Y,Call,NYVal),
    division_float_intervals(NXVal, NYVal, NVal_).
```

Where `division_float_interval/3` calls `division_float_intervals_/4` 4.3.3 giving it the set of rounding modes that are selected via the `rounding_modes/1` predicate. It is implemented as follows:

```
  division_float_interval(Y, Z, X) :-
    rounding_modes(S),
    division_float_intervals_(S, Y, Z, X).
```

The rest of predicates related with addition, substraction and product are defined in the same way.

Other typical predicates that need to be implemented as part of the domain (not included in this description) are those required to interpret the properties in the assertions (in CiaoPP's terminology *input interface*), or represent abstract substitutions as properties (in CiaoPP's terminology *native* properties) suitable for the interaction with other analyses or the output of the analysis as user-level assertions.

5

Experimental results

We applied our implementation in different contexts in which we believe it can provide extra value by avoiding some errors relating floating point computations.

5.1 Missile Failure

The failure occurred during the Gulf War that opened this work 1 will also open this chapter. The following is extracted from the GAO report. The Patriot battery at Dhahran failed to track and intercept the Scud missile because of a software problem in the system's weapons control computer. This problem led to an inaccurate tracking calculation that became worse the longer the system operated. At the time of the incident, the battery had been operating continuously for over 100 hours. By then, the inaccuracy was serious enough to cause the system to look in the wrong place for the incoming Scud.

The following code captures the behavior of the counter that led to the failure in the Patriot. This is, running for 100 hours and updating a counter each 0.1 seconds.

```
:- module(_, _, [assertions]).

counter(X) :-
   Y is 1.0/10.0,
   % Each 0.1 seconds add 1 integer => 100*60*60*100
   Times is 100.0*60.0*60.0*10.0,
   X is Y*Times.

```

Running this experiment we obtained the interval: $i(360000.0, 360000.03125)$. If we take into account that a Scud missile travels at about 1,676 meters per second and the size of the interval (which in this case is equivalent to the error) is 0.03125 we can approximate that in that time a missile can travel up to 52, 375 meters which is a large distance when dealing with this kind of software.

5.2 ESA trigonometric functions

One of the industries which use floating point computations most is the aerospace industry. In this industry there are two critical points that must be taken into account. Since it is almost sure that there will be floating point errors, it is desirable to minimize as much as possible this error. One possible solution would be to use floating points with a huge precision. Sadly this is not always possible since it would lead to a large use of memory that would increase the prize of the components and the size. Our implementation allows us to compare different floating point sizes in order to know which one is the smallest floating point that generates the smaller error.

We have considered the implementation of the arccosin from the ESA implemented with the following algorithm and the following use of constants:

```
      % __ieee754_acos(x)
      % Method :
      % acos(x)  = pi/2 - asin(x)
      % acos(-x) = pi/2 + asin(x)
      % For |x|<=0.5
      % acos(x) = pi/2 - (x + x*x^2*R(x^2)) (see asin.c)
      % For x>0.5
      %   acos(x) = pi/2 - (pi/2 - 2asin(sqrt((1-x)/2)))
      %   = 2asin(sqrt((1-x)/2))
      %   = 2s + 2s*z*R(z)  ...z=(1-x)/2, s=sqrt(z)
      %   = 2f + (2c + 2s*z*R(z))
      %     where f=hi part of s, and c = (z-f*f)/(s+f) is the correction term
      %     for f so that f+c = sqrt(z).
      % For x<-0.5
      % acos(x) = pi - 2asin(sqrt((1-|x|)/2))
      %   = pi - 0.5*(s+s*z*R(z)), where z=(1-|x|)/2,s=sqrt(z)
      %
      %
      ...
      ...
      ...
      one=  1.00000000000000000000e+00, // 0x3FF00000, 0x00000000
      pi =  3.14159265358979311600e+00, // 0x400921FB, 0x54442D18
      pio2\_hi =  1.57079632679489655800e+00, // 0x3FF921FB, 0x54442D18
      pio2\_lo =  6.12323399573676603587e-17, // 0x3C91A626, 0x33145C07
      pS0 =  1.66666666666666657415e-01, // 0x3FC55555, 0x55555555
      pS1 = -3.25565818622400915405e-01, // 0xBFD4D612, 0x03EB6F7D
      pS2 =  2.01212532134862925881e-01, // 0x3FC9C155, 0x0E884455
      pS3 = -4.00555345006794114027e-02, // 0xBFA48228, 0xB5688F3B
      pS4 =  7.91534994289814532176e-04, // 0x3F49EFE0, 0x7501B288
      pS5 =  3.47933107596021167570e-05, // 0x3F023DE1, 0x0DFDF709
```

```
    qS1  = -2.40339491173441421878e+00, // 0xC0033A27, 0x1C8A2D4B
    qS2  =  2.02094576023350569471e+00, // 0x40002AE5, 0x9C598AC8
    qS3  = -6.88283971605453293030e-01, // 0xBFE6066C, 0x1B8D0159
    qS4  =  7.70381505559019352791e-02; // 0x3FB3B8C5, 0xB12E9282

```

First of all notice that they consider just 20 decimals once they define the constants. Moreover the given definition of pi is not exact since the 20 first decimals of pi are .14159265358979323846 and they consider .14159265358979311600. This is an error of 0.00000000000000001224. In this analysis we are going to consider the different possible branches in order to be able to analyze each case with more detail. We considered 32 and 64 bits floating points and a fixed point of 24 bits. We also tried different rounding configurations available since this is a design decision and in a computation different roundings can happen.

Since the code consider different partitions of the space let us consider the results obtained with out analysis and compare them with the actual expected values. First of all let us take into account that $\forall x \in \mathbb{R}, acos(x) \in [-\pi, \pi]$

If $x \in [-1, -0.5]$. The fragment of code which implements the behavior of the acosin in this case is:

```
      ...
      else  if (hx<0) {   // x < -0.5
  z = (one+x)*0.5;
  p = z*(pS0+z*(pS1+z*(pS2+z*(pS3+z*(pS4+z*pS5)))));
  q = one+z*(qS1+z*(qS2+z*(qS3+z*qS4)));
  s = __ieee754_sqrt(z);
  r = p/q;
  w = r*s-pio2_lo;
  return pi - 2.0*(s+w);
      }
      ...

```

Which equivalent **Ciao** code is:

```
      acos_operations_branch3(X1, Branch) :-
        One =  1.00000000000000000000e+00, % 0x3FF00000, 0x00000000
        ...
        QS4 =  7.70381505559019352791e-02, % 0x3FB3B8C5, 0xB12E9282
        X1 < -0.5, X1 > -1.0, !,
        Z is (One + X1)*0.5,
        P is Z*(PS0+Z*(PS1+Z*(PS2+Z*(PS3+Z*(PS4+Z*PS5))))),
        Q is One+Z*(QS1+Z*(QS2+Z*(QS3+Z*QS4))),
        %S is sqrt(Z),
        S > 0.0, S < 0.5, %% Z in i(0.0, 0.25)
        R is P/Q,
        W is R*S-Pio2_lo,
        Branch is Pi -2.0*(S+W).
```

In this case we expected the result to be in the interval $[\frac{2\pi}{3}, \pi]$. Our analysis obtained:

```
    :- true pred 'esa_cos:acos_operations_branch3'(A,B)
 : ( A/'$top', B/'$top' )
 => ( A/i(-1.0,-0.5), B/i(2.0372044444084167,3.141592653589793) )
 + ( complete_id(5), domain(nonrel_fintervals),
    ↪ callers([('esa_cos:acos_operations_branch3/2',0)]) ).
```

The interval that we have obtained is bigger than the interval $[\frac{2\pi}{3}, \pi]$, since $\frac{2\pi}{3} \approx 2.094395102...$.

If $x \in [-0.5, 0.5]$ the result is expected to be in the interval $[\frac{\pi}{3}, \frac{2\pi}{3}]$. In this case the implementation corresponds to:

```
    if(ix<0x3fe00000) { // |x| < 0.5
  z = x*x;
  p = z*(pS0+z*(pS1+z*(pS2+z*(pS3+z*(pS4+z*pS5)))));
  q = one+z*(qS1+z*(qS2+z*(qS3+z*qS4)));
  r = p/q;
  return pio2_hi - (x - (pio2_lo-x*r));
    }

```

That can be seen as `Ciao` code as:

```
      acos_operations_branch2(X1, Branch) :-
         One =  1.00000000000000000000e+00, % 0x3FF00000, 0x00000000
         ...
         QS4 =  7.70381505559019352791e-02, % 0x3FB3B8C5, 0xB12E9282
         X1 < 0.5, X1 > -0.5, !,% |X| < 0.5
         Z is X1*X1,
         P is Z*(PS0+Z*(PS1+Z*(PS2+Z*(PS3+Z*(PS4+Z*PS5))))),
         Q is One+Z*(QS1+Z*(QS2+Z*(QS3+Z*QS4))),
         R is P/Q,
         Branch is Pio2_hi - (X1 - (Pio2_lo-X1*R)).
```

where we obtained:

```
    :- true pred 'esa_cos:acos_operations_branch2'(A,B)
      : ( A/'$top', B/'$top' )
       => ( A/i(-0.5,0.5), B/i(0.9460467100143433,2.195545792579651) )
       + ( complete_id(4), domain(nonrel_fintervals),
           ↪ callers([('esa_cos:acos_operations_branch2/2',0)]) )
```

Here we obtained another overaproximation since $\frac{\pi}{3} \approx 1.047197551...$ and $\frac{2\pi}{3} \approx 2.094395102...$.

If $x \in [0.5, 1.0]$ the result is expected to be in the interval $[0.0, \frac{pi}{3}]$ And the implemented code is:

```
        else {        // x > 0.5
    z = (one-x)*0.5;
    s = __ieee754_sqrt(z);
    df = s;
    SET_LOW_WORD(df,0);
    c  = (z-df*df)/(s+df);
    p = z*(pS0+z*(pS1+z*(pS2+z*(pS3+z*(pS4+z*pS5)))));
    q = one+z*(qS1+z*(qS2+z*(qS3+z*qS4)));
    r = p/q;
    w = r*s+c;
    return 2.0*(df+w);
        }

```

Here the `Ciao` code is:

```
        acos_operations_branch4(X1, Branch) :-
           One =  1.00000000000000000000e+00, % 0x3FF00000, 0x00000000
           ...
           QS4 =  7.70381505559019352791e-02, % 0x3FB3B8C5, 0xB12E9282
           X1 > 0.5, X1 < 1.0, !,
           Z is (One - X1)*0.5,
           S > 0.0, S < 0.5, %% Z in i(0.0, 0.25)
           Df is 0.0,
           C is (Z - Df*Df)/(Df + S),
           P is Z*(PS0+Z*(PS1+Z*(PS2+Z*(PS3+Z*(PS4+Z*PS5))))),
           Q is One+Z*(QS1+Z*(QS2+Z*(QS3+Z*QS4))),
           R is P/Q,
           W is R*S+C,
           Branch is 2.0*(Df+W).
```

Notice that in this transformation S corresponds to the square root of Z, in this case and since we do not have a direct projection for square roots we used the information that we have from Z to give an appropriate interval for S. Same happens with Df which we set to zero because both extrema of the interval S are set to zero. In this cases results are not as good as it would be desirable. We obtain an interval

```
      :- true pred 'esa_cos:acos_operations_branch4'(A,B)
   : ( A/'$top', B/'$top' )
   => ( A/i(0.5,1.0), B/i(-0.0,0.Inf) )
   + ( complete_id(6), domain(nonrel_fintervals),
      ↪ callers([('esa_cos:acos_operations_branch4/2',0)]) ).
```

which is an interval containing the expected interval but much bigger. This is due to the `SET_LOW_WORD` behavior which we were not able to capture.

In this case we have considered floating-point numbers of 64 bits and all the possible rounding modes. This means that if the interval obtained for is $i(x_l, x_u)$ for all input and every sequence of rounding modes the result is in that interval.

We considered different configurations, since we need to measure somehow the distance between the interval obtained and the "correct" interval we have defined the measure:

$m(i(x_l, x_u), i(y_l, y_u))) = ||x_u - x_l| - |y_u - y_l||$ where $i(x_l, x_u)$ is the correct interval and $i(y_u, y_l)$ the obtained one.

	Float 64			Float32		
	B1	B2	B3	B1	B2	B3
$\{n, 0, \uparrow, \downarrow\}$	0.20230214043057648	0.057190608337404	∞	0.2023023658337368	0.05719092422149	∞
$\{n\}$	0.20230214043057603	0.057190608337400	∞	0.2023020678105129	0.05719044738433	∞
$\{0\}$	0.20230214043057559	0.057190608337403	∞	0.2023020082058681	0.05719068580291	∞
$\{\uparrow\}$	0.20230214043058580	0.057190608337403	∞	0.2023023062290920	0.05719092422149	∞
$\{\downarrow\}$	0.20230214043057603	0.057190608337403	∞	0.2023020678105129	0.05719068580291	∞

Table 5.1: Result of apply Floats to ESA arcos function

	$\mathbb{F}ix(40, 24)$		
	B1	B2	B3
$\{n, 0, \uparrow, \downarrow\}$	0.202301527606804	0.057190657	∞
$\{0\}$	0.202301527606804	0.057190657	∞
$\{n\}$	0.202301527606804	0.057190657	∞
$\{\uparrow\}$	0.202301527606804	0.057190657	∞
$\{\downarrow\}$	0.202301527606804	0.057190657	∞

Table 5.2: Result of apply 24 bits fixed-point to ESA arcos function

In this case we can see that the error does not vary too much with different configurations of floating point and different possible rounding modes. It is also surprising that the most precise mode seems to be the fixpoint with 64 bits and 24 bits of precision. This format also remains the same under changes of rounding modes.

6 Conclusions

Motivated by the importance of floating-point arithmetic in software development we aimed to provide a tool capable of analyzing different types of programs in order to help ensure some error control and making developers able to take some architectural decisions with some strong guarantees related to the type of numbers to be used. With this humble objective we summoned the powerful theory of abstract interpretation and together with the strong system that `Ciao` provides we developed a prototype of what we hope will be eventually be a complete and effective bundle that solves some of the problems that we enumerated in Chapter 1. Looking back from the beginning of this work we can enumerate the following tasks that we have accomplished:

1. We implemented an analysis based on a refinement of the Interval Domain which captures the ranges where values obtained by floating-point computations can be found, allowing developers to estimate whether their programs can behave in an unexpected way.

2. We proved that the basic operations implemented are correct and optimal.

3. We have been able to analyze the same programs with different configurations of floating-point numbers (32 and 64 bits) and we also included all possible fixed-point number configurations as they can be useful when working with legacy software.

4. We have been able to consider different rounding modes and handle the cases where knowledge about the rounding mode being used is limited.

5. We implemented a new numerical analyzer within the current version of `CiaoPP`.

6.1 Future work

We would like to end this work with some lines that we hope to be able to explore in the future:

Capturing the behavior of NaNs

In our implementation we (and most state-of-the-art analyzers) only consider the floating-point numbers. [1] proposes considering an auxiliary boolean domain where $\top$ means "can be Not a Number" and $\bot$ means "cannot be a NaN". We aim to include this domain together with the implemented domain as a product domain [63].

Implement a correct backward analysis

Even though we currently have the theory background needed to develop a correct backward analysis for this domain we did not have enough time to implement it on the current analysis. One of the first things that we aim to do is to extend our domain with backward analysis. This extension would allow the developer to ensure whether some concrete and unexpected behavior can happen in a program. If we consider the following code:

```
double weird_sum(float x){ max_range = 100; double sum = 0; for int
  i=0; i<100; i++ { sum = sum + i/x; } return sum; }
```

This code computes $\sum_{i=0}^{100} \frac{i}{x}$. It would be desirable to know whether the value that the function returns is or not NaN. A backward analysis would generate an interval $[-0.0, +0.0]$ pointing out that if some value is selected among that interval it could lead to a NaN. This would also help to create test suites.

Develop better domains for numerical analysis

Interval analysis is a good option when trying a prototype and has some valuable properties; among them simplicity is an important one. Anyway as we saw in 2.3.2 the Affine Domain which was proposed by Sylve Putot et al. obtains narrower intervals. It would be desirable to develop an analysis where both Interval and Affine analysis were executed together, using Interval Analysis where the relational power that the Affine Domain offers is not needed.

It would also be interesting to try to find newer domains able to capture narrower intervals, at least on some subsets of the language, that could be used together with currently existing domains.

Test current work in Industrial Code

We have used our tool on fragments of industrial code as we have seen in Section 5.2. We would like to try this approach in full industrial codes to help to develop test suites inside the testing cycle and detect other undesirable behaviors that could hide from a testing approach as underflows/overflows.

Bibliography

[1] Roberto Bagnara et al. *Correct Approximation of IEEE 754 Floating-Point Arithmetic for Program Verification.* 2019. arXiv: `1903.06119 [cs.PL]`.

[2] Roberto Bagnara et al. *A Practical Approach to Interval Refinement for math.h/cmath Functions.* 2020. arXiv: `1610.07390 [cs.PL]`.

[3] Siegfried M. Rump. "Accurate solution of dense linear systems, Part II: Algorithms using directed rounding". In: *Journal of Computational and Applied Mathematics* 242 (2013), pp. 185–212. ISSN: 0377-0427.

[4] Siegfried M. Rump and Takeshi Ogita. "Super-fast validated solution of linear systems". In: *Journal of Computational and Applied Mathematics* 199.2 (2007). Special Issue on Scientific Computing, Computer Arithmetic, and Validated Numerics (SCAN 2004), pp. 199–206. ISSN: 0377-0427.

[5] G. Fiedler. *Floting point determinism.* URL: `https://gafferongames.com/post/floating_point_determinism/`.

[6] "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Std 754-2008* (2008), pp. 1–70. DOI: `10.1109/IEEESTD.2008.4610935`.

[7] H. So. *Introduction to Fixed Point Number Representation.* URL: `https://inst.eecs.berkeley.edu/~cs61c/sp06/handout/fixedpt.html`.

[8] Randy Yates. *Fixed-Point Arithmetic: An Introduction.* 2007. URL: `https://courses.cs.washington.edu/courses/cse467/08au/labs/l5/fp.pdf`.

[9] JMC47 neobrain MayImilae. *Pixel Processing Problems: On the Road to Pixel Perfection.* 2019. URL: `https://es.dolphin-emu.org/blog/2014/03/15/pixel-processing-problems/`.

[10] *PostgreSQL manual.* Chap. 8.1. Numeric Types. URL: `https://www.postgresql.org/docs/current/datatype-numeric.html`.

[11] Patrick Cousot and Radhia Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages.* POPL '77. Los Angeles, California: Association for Computing Machinery, 1977, 238–252. ISBN: 9781450373500. DOI: `10.1145/512950.512973`. URL: `https://doi.org/10.1145/512950.512973`.

[12] Jorge Stolfi, L. FIGUEIREDO, and Estrada Dona. "An Introduction to Affine Arithmetic". In: *TEMA. Tendências em Matemática Aplicada e Computacional* 4 (Dec. 2003). DOI: `10.5540/tema.2003.04.03.0297`.

[13] Eric Goubault and Sylvie Putot. “Static Analysis of Numerical Algorithms”. In: *Static Analysis*. Ed. by Kwangkeun Yi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 18–34. ISBN: 978-3-540-37758-0.

[14] Bala Surendra Adusumilii and Boddeti Kalyan Kumar. “Backward/Forward Sweep based Power Flow Analysis of Distribution Systems under Uncertainty using New Affine Arithmetic Division”. In: *2020 IEEE Power Energy Society Innovative Smart Grid Technologies Conference (ISGT)*. 2020, pp. 1–5. DOI: 10.1109/ISGT45199.2020.9087718.

[15] Liqian Chen, Antoine Miné, and Patrick Cousot. “A Sound Floating-Point Polyhedra Abstract Domain”. In: *Programming Languages and Systems*. Ed. by G. Ramalingam. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 3–18. ISBN: 978-3-540-89330-1.

[16] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. “The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems”. In: *Science of Computer Programming* 72.1 (2008). Special Issue on Second issue of experimental software and toolkits (EST), pp. 3–21. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2007.08.001. URL: https://www.sciencedirect.com/science/article/pii/S0167642308000415.

[17] Gagandeep Singh, Markus Püschel, and Martin Vechev. “Fast Polyhedra Abstract Domain”. In: *SIGPLAN Not.* 52.1 (Jan. 2017), 46–59. ISSN: 0362-1340. DOI: 10.1145/3093333.3009885. URL: https://doi.org/10.1145/3093333.3009885.

[18] Antoine Miné. “The Octagon Abstract Domain”. In: *Higher-Order and Symbolic Computation* 19 (2006), pp. 31–100. URL: https://hal.archives-ouvertes.fr/hal-00136639.

[19] Antoine Miné. “Weakly Relational Numerical Abstract Domains”. PhD thesis. École Normale Supérieure, 2004. URL: https://www-apr.lip6.fr/~mine/these/these-color.pdf.

[20] Antoine Miné. “Symbolic Methods to Enhance the Precision of Numerical Abstract Domains”. In: LNCS 3855. Springer, Jan. 2006, pp. 348–363. URL: https://hal.archives-ouvertes.fr/hal-00136661.

[21] Alexandre Chapoutot. “Interval Slopes as a Numerical Abstract Domain for Floating-Point Variables”. In: *Lecture Notes in Computer Science* (2010), 184–200. ISSN: 1611-3349. DOI: 10.1007/978-3-642-15769-1_12. URL: http://dx.doi.org/10.1007/978-3-642-15769-1_12.

[22] Manuel V. Hermenegildo et al. “An overview of Ciao and its design philosophy”. In: *Theory and practice of logic programming* 12.1-2 (2012), pp. 219–252.

[23] G. Puebla, F. Bueno, and M. V. Hermenegildo. “Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs”. In: *Logic-based Program Synthesis and Transformation (LOPSTR’99)*. LNCS 1817. Springer-Verlag, 2000, pp. 273–292. DOI: 10.1007/10720327_16.

[24] M. V. Hermenegildo, G. Puebla, and F. Bueno. “Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging”. In: *The Logic Programming Paradigm: a 25–Year Perspective*. Ed. by K. R. Apt et al. Springer-Verlag, 1999, pp. 161–192.

[25] M. V. Hermenegildo et al. “Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor)”. In: *Science of Computer Programming* 58.1–2 (2005), pp. 115–140. ISSN: ISSN 0167-6423. DOI: 10.1016/j.scico.2005.02.006.

[26] M. V. Hermenegildo et al. “The Ciao Approach to the Dynamic vs. Static Language Dilemma”. In: *Proceedings for the International Workshop on Scripts to Programs (STOP'11)*. Austin, Texas, USA: ACM, 2011.

[27] Cormac Flanagan. “Hybrid Type Checking”. In: *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*. Ed. by J. Gregory Morrisett and Simon L. Peyton Jones. ACM, 2006, pp. 245–256. ISBN: 1-59593-027-2. DOI: 10.1145/1111037.1111059. URL: https://doi.org/10.1145/1111037.1111059.

[28] Jeremy G. Siek. “Gradual typing for functional languages”. In: *In Scheme and Functional Programming Workshop*. 2006, pp. 81–92.

[29] M. V. Hermenegildo et al. “An Overview of Ciao and its Design Philosophy”. In: *Theory and Practice of Logic Programming* 12.1–2 (2012), pp. 219–252. ISSN: 1471-0684. DOI: 10.1017/S1471068411000457. URL: http://arxiv.org/abs/1102.5497.

[30] E. Albert, G. Puebla, and M. V. Hermenegildo. “Abstraction-Carrying Code”. In: *Proc. of LPAR'04*. Vol. 3452. LNAI. Springer, 2005.

[31] M. Méndez-Lojo, J. Navas, and M. Hermenegildo. “A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs”. In: *LOPSTR*. Vol. 4915. LNCS. Springer-Verlag, 2007, pp. 154–168. DOI: 10.1007/978-3-540-78769-3_11.

[32] J.C. Peralta, J. Gallagher, and H. Sağlam. “Analysis of Imperative Programs through Analysis of Constraint Logic Programs”. In: *Static Analysis. 5th International Symposium, SAS'98, Pisa*. Ed. by G. Levi. Vol. 1503. LNCS. 1998, pp. 246–261. DOI: 10.1007/3-540-49727-7_15.

[33] Kim S. Henriksen and John P. Gallagher. “Abstract Interpretation of PIC Programs through Logic Programming”. In: *SCAM '06*. IEEE Computer Society, 2006, pp. 184–196. ISBN: 0-7695-2353-6. DOI: 10.1109/SCAM.2006.1.

[34] M. Gómez-Zamalloa, E. Albert, and G. Puebla. “Decompilation of Java Bytecode to Prolog by Partial Evaluation”. In: *JIST* 51 (10 2009), pp. 1409–1427. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2009.04.010.

[35] Sergey Grebenshchikov et al. “Synthesizing software verifiers from proof rules”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12*. Ed. by Jan Vitek, Haibo Lin, and Frank Tip. ACM, 2012, pp. 405–416. DOI: 10.1145/2254064.2254112.

[36] Arie Gurfinkel et al. “The SeaHorn Verification Framework”. In: *International Conference on Computer Aided Verification, CAV 2015*. LNCS 9206. Springer, 2015, pp. 343–361. DOI: 10.1007/978-3-319-21690-4_20.

[37] Emanuele De Angelis et al. “Semantics-based generation of verification conditions by program specialization”. In: *17th International Symposium on Principles and Practice of Declarative Programming*. ACM, 2015, pp. 91–102. DOI: 10.1145/2790449.2790529.

[38] Temesghen Kahsai et al. “JayHorn: A Framework for Verifying Java Programs”. In: *Computer Aided Verification - 28th International Conference, CAV 2016*. Ed. by Swarat Chaudhuri and Azadeh Farzan. Vol. 9779. LNCS. Springer, 2016, pp. 352–358. DOI: 10.1007/978-3-319-41528-4_19.

[39] J. Gallagher et al. “From big-step to small-step semantics and back with interpreter specialization (invited paper)”. In: *International WS on Verification and Program Transformation (VPT 2020)*. EPTCS. Open Publishing Association, 2020, pp. 50–65. DOI: 10.4204/EPTCS.320.4.

[40] P. Lopez-Garcia et al. “Interval-based Resource Usage Verification by Translation into Horn Clauses and an Application to Energy Consumption”. In: *Theory and Practice of Logic Programming, Special Issue on Computational Logic for Verification* 18.2 (2018), pp. 167–223. DOI: 10.1017/S1471068418000042. URL: https://arxiv.org/abs/1803.04451.

[41] J. Navas, M. Méndez-Lojo, and M. V. Hermenegildo. “User-Definable Resource Usage Bounds Analysis for Java Bytecode”. In: *BYTECODE’09*. Vol. 253. ENTCS 5. Elsevier, 2009, pp. 6–86. DOI: 10.1016/j.entcs.2009.11.015. URL: http://cliplab.org/papers/resources-bytecode09.pdf.

[42] J. Navas, M. Méndez-Lojo, and M. Hermenegildo. “Safe Upper-bounds Inference of Energy Consumption for Java Bytecode Applications”. In: *The Sixth NASA Langley Formal Methods Workshop (LFM 08)*. Extended Abstract. 2008, pp. 29–32.

[43] U. Liqat et al. “Energy Consumption Analysis of Programs based on XMOS ISA-Level Models”. In: *Proceedings of LOPSTR’13*. Vol. 8901. LNCS. Springer, 2014, pp. 72–90. DOI: 10.1007/978-3-319-14125-1_5.

[44] U. Liqat et al. “Inferring Parametric Energy Consumption Functions at Different Software Levels: ISA vs. LLVM IR”. In: *Proc. of FOPARA*. Vol. 9964. LNCS. Springer, 2016, pp. 81–100. DOI: 10.1007/978-3-319-46559-3_5.

[45] V. Perez-Carrasco et al. “Cost Analysis of Smart Contracts via Parametric Resource Analysis”. In: *Static Aanalysis Simposium (SAS’20)*. Vol. 12389. LNCS. Springer, 2020, pp. 7–31. DOI: 10.1007/978-3-030-65474-0_2.

[46] E. Mera et al. “Towards Execution Time Estimation in Abstract Machine-Based Languages”. In: *PPDP’08*. ACM Press, 2008, pp. 174–184. DOI: 10.1145/1389449.1389471.

[47] U. Liqat et al. “Inferring Energy Bounds via Static Program Analysis and Evolutionary Modeling of Basic Blocks”. In: *Logic-Based Program Synthesis and Transformation - 27th International Symposium*. Vol. 10855. LNCS. Springer, 2018. DOI: 10.1007/978-3-319-94460-9_4.

[48] G. Puebla, F. Bueno, and M. V. Hermenegildo. "An Assertion Language for Constraint Logic Programs". In: *Analysis and Visualization Tools for Constraint Programming*. Ed. by P. Deransart, M. V. Hermenegildo, and J. Maluszynski. LNCS 1870. Springer-Verlag, 2000, pp. 23–61. ISBN: 978-3-540-40016-5.

[49] F. Bueno et al. "Global Analysis of Standard Prolog Programs". In: *European Symposium on Programming*. LNCS 1058. Sweden: Springer-Verlag, 1996, pp. 108–124. ISBN: ISBN 3-540-61055-3.

[50] J.W. Lloyd. *Foundations of Logic Programming*. Springer, second, extended edition, 1987.

[51] K. R. Apt. "Introduction to Logic Programming". In: *Handbook of Theoretical Computer Science*. Ed. by J. van Leeuwen. Elsevier, 1990, pp. 493–576.

[52] M. Bruynooghe. "A Practical Framework for the Abstract Interpretation of Logic Programs". In: *Journal of Logic Programming* 10 (1991), pp. 91–124.

[53] K. Muthukumar and M. Hermenegildo. *Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation*. Technical Report ACA-ST-232-89. Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759, 1989.

[54] K. Muthukumar and M. Hermenegildo. "Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation". In: *1989 North American Conference on Logic Programming*. MIT Press, 1989, pp. 166–189.

[55] K. Muthukumar and M. Hermenegildo. "Compile-time Derivation of Variable Dependency Using Abstract Interpretation". In: *Journal of Logic Programming* 13.2/3 (1992). Ed. by S. Debray, pp. 315–347.

[56] M. V. Hermenegildo et al. "Incremental Analysis of Constraint Logic Programs". In: *ACM TOPLAS* 22.2 (2000), pp. 187–223. DOI: 10.1145/349214.349216.

[57] P. Lopez-Garcia, F. Bueno, and M. V. Hermenegildo. "Automatic Inference of Determinacy and Mutual Exclusion for Logic Programs Using Mode and Type Analyses". In: *New Generation Computing* 28.2 (2010), pp. 117–206. ISSN: 0288-3635.

[58] Andy King, Lunjin Lu, and Samir Genaim. "Detecting Determinacy in Prolog Programs." In: *Logic Programming, 22nd International Conference, ICLP 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*. Ed. by Sandro Etalle and Miroslaw Truszczynski. Vol. 4079. Lecture Notes in Computer Science. Springer, 2006, pp. 132–147. ISBN: 3-540-36635-0.

[59] C. Braem et al. "Cardinality Analysis of Prolog". In: *Proc. International Symposium on Logic Programming*. Ithaca, NY: MIT Press, 1994, pp. 457–471.

[60] S.K. Debray, P. Lopez-Garcia, and M. V. Hermenegildo. "Non-Failure Analysis for Logic Programs". In: *1997 International Conference on Logic Programming*. Cambridge, MA: MIT Press, Cambridge, MA, 1997, pp. 48–62.

[61] K. Muthukumar and M. Hermenegildo. *Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation*. 1991.

[62] J. A. Robinson. “A Machine Oriented Logic Based on the Resolution Principle”. In: *Journal of the ACM* 12.23 (1965), pp. 23–41.

[63] Michael Codish et al. “Improving Abstract Interpretations by Combining Domains.” In: *ACM Trans. Program. Lang. Syst.* 17 (Jan. 1995), pp. 28–44. DOI: 10.1145/154630.154650.