Static Responsibility Analysis of Floating-Point Programs

by

Goktug Saatcioglu

A thesis submitted in partial fulfillment of

the requirements for the degree of

Master of Science

Computer Science Department
New York University

May 2020

Advisor: Prof. Thomas Wies

Second reader: Prof. Patrick Cousot

Acknowledgements

This thesis could have not materialized without the guidance of my adviser, Prof. Thomas Wies. I started my journey in research 3 years ago and I am thankful for the countless hours spent helping me understand both how to conduct research and the various aspects of program verification. I am also thankful for his support and patience throughout my time at NYU. I would like to thank Patrick Cousot for introducing abstract interpretation to me and meeting me whenever I had any questions. His thorough course has helped me to think of program analysis in a systematic manner.

I would also like to thank Chaoqiang Deng for his patience in explaining parts of responsibility analysis to me and the many discussions we had. I would like to thank Margaret Wright for introducing me to the issues of floating-point arithmetic and having interesting discussions with me. The project I did for her class is the starting point for my thesis. I would also like to thank Anasse Bari for supporting me during the last year of my undergraduate education.

I would like to thank the Computer Science Department at NYU for their generous support in awarding the M.S. Thesis/Research Fellowship when I was just beginning this work.

Finally, I want to thank my family for their everlasting support in pursuing my goals: baba, thank you for always being there to guide and advise me; anne, thank you for the pep talks and reminding me to have fun; and Nazli, thank you for supporting me whenever I needed it, especially during the time of writing this thesis.

Abstract

The last decade has seen considerable progress in the analysis of floating-point programs. There now exist frameworks to verify both the total amount of round-off error a program accrues and the robustness of floating-point programs. However, there is a lack of static analysis frameworks to identify causes of erroneous behaviors due to the use of floating-point arithmetic. Such errors are both sporadic and triggered by specific inputs or numbers computed by programs. In this work, we introduce a new static analysis by abstract interpretation to define and detect responsible entities for such behaviors in finite precision implementations. Our focus is on identifying causes of test discontinuity where small differences in inputs may lead to large differences in the control flow of programs causing the computed finite precision path to differ from the same ideal computation carried out in real numbers. However, the analysis is not limited to just discontinuity, as any type of error cause can be identified by the framework. We propose to carry out the analysis by a combination of over-approximating forward partitioning semantics and under-approximating backward semantics of programs, which leads to a forward-backward static analysis with iterated intermediate reduction. This gives a way to the design of a tool for helping programmers identify and fix numerical bugs in their programs due to the use of finite-precision numbers. The implementation of this tool is the next step for this work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Floating point numbers are the widely used standard of carrying out computations over real numbers in finite-precision. The IEEE Standard for Floating-Point Arithmetic [08] provides certain theoretical guarantees about how much error is accrued due to the use of finite-precision arithmetic. However, these results only hold for single opertions and are no longer true when we compose operations. Furthermore, basic algebraic facts about real numbers no longer hold when working with floating-point numbers. Thus, an unfamiliar programmer may observe seemingly unpredictable behaviors when using floating point arithmetic. This is due to the unintuitive semantics of the IEEE Standard for those who are not familiar with it. Round-off error can slowly build up and go undiscovered before abruptly crashing a program or causing catastrophes. Such examples include the resetting of the Vancouver stock exchange [Mac15] and the failure of the Partiot missile system during the Gulf war [Ske]. In both examples, the subtle build-up in rounding errors were only revealed after causing a major issue.

Another problem besides rounding is that the control flow of floating-point programs may not follow the ideal (expected) control flow of the same program under real semantics. For example, a conditional statement that we always expect to evaluate to true may sometimes become false. This leads to floating-point programs exhibiting instability in its control flow where for certain inputs the finite precision control flow differs from the same execution of the program using real numbers. Discontinuity can cause major bugs in critical systems such as when a F22 Raptor military aircraft almost crashed after crossing the international date line [Bus11]. This near-crash occurred because the software on board encountered a discontinuity in the treatment of dates.

There has been a lot of progress in statically analyzing programs to estimate the round-off error. However, these analyses are only sound under the so called stable test assumption where the analysis is sound only if no test conditional divergence occurs. Some recent work such as [GP13] has focused on statically verifying robustness properties of floating-point programs which is a concept closely related to the continuity of programs. Introduced in [Ham02], a program is called continuous if small pertubations in its input do not cause large pertubations in its output. So, a program that has a test discontinuity could lead to a large variation in its result which makes it 'un-robust.'

Static analysis by abstract interpretation [CC77] works especially well for this type of analysis as we can consider all possible inputs without testing. Exhaustive testing is difficult to carry out in general and especially difficult for floating-point computations as the inputs that cause errors can be sporadic and very specific. Furthermore, with abstract interpretation we may design a sound over-approximation of all the possible behaviors of a given program. Thus, all errors are guaranteed to be detected, though there may also be some false alarms.

Detecting round-off errors that exceed a threshold or test discontinuity is already a non-trivial task. An even harder problem that is overlooked in many frameworks is the identification of the causes of undesirable floating-point behaviors. Programmers who focus on numerical computing are able to look at the numerical software and identify the points in the program that may cause issues. But, not all programmers are experts and they may not be aware of stability and rounding issues in floating-point programs. They may not realize where an error in their numeric code comes from even if a tool alerts them to the presence of errors. To address this issue, this thesis introduces the static responsibility analysis of floating-point programs. Our first contribution is a formalization of when a program entity can be considered responsible for some bad floating-point behavior $\mathfrak{B}$. For this, we use the framework for responsibility analysis introduced in [DC19] where causation is defined counterfactually and characterized using hyperproperties of programs [CS10]. Then, we introduce the abstaction of this concrete semantics which utilizes both the forward and backward semantics of floating-point programs along with trace partitioning [RM07] to detect floating-point errors. Lastly, we present a method to identify responsible program entities with respect to our definition of responsibility. Our analysis is able to detect all types of floating-point errors along with their responsible entities. However, for this thesis we focus on the finding responsible entities for test discontinuity.

Organization of the thesis. We begin with Chapter 2 which presents an example program to demonstrate test discontinuity. Furthermore, this example program will be used as a running example throughout the thesis. In Chapter 3 we review the necessary background knowledge for this thesis. This review consists of floating-point numbers, abstract interpretation, constrained affine sets [GP13], definition of responsibility, trace partitioning and backward semantics. Chapter 4 introduces the responsibility analysis for floating-point programs where the analysis is built piece by piece. There are also detailed discussions for each section. The thesis concludes with Chapters 5 and 6 that discusses some related work and future directions for this research.

Chapter 2

Motivating Example

In this chapter we demonstrate the occurence of test discontinuity for a simple program and how the analysis of constrained affine sets for robustness introduced in [GP13] would detect the discontinuity. Throughout this section the words discontinuity and (conditional) divergence will be used interchangeably to refer to the same phenomenon. When we say if-else divergence we mean that the real program took the if branch of a conditional while the floating-point program took the else branch of a conditional. Similarly, else-if divergence is when the real program evaluates to the else branch while the floating-point program evaluates to the if branch. In the example Program 2.1 both types of divergence occurs for a given range of inputs for the program. Affine analysis will allow us to both identify the divergence and also split it into its cases such that we may characterize the traces that lead to specific types of discontinuity. Throughout the rest of the thesis we will refer back to Program 2.1 and how the proposed analysis will be able to identify the responsible entities for divergence through a series of analyses.

```
1     z := [0,1] + uz  // z is in [0,1] with uncertainty uz
2     x := [1,3] + ux
3     y := [0,2] + uy
4     z = z + 4
5     if (x <= 2 && y >= 1) {
6       z = x + y
7     } else {
8       z = x - y
9     }
```

Program 2.1: Example program.

2.1 Constrained Affine Sets for Detecting Discontinuity

The example program, which will be referred to as Program 2.1, is similar to the example program given in [GP13]. The program is specified with three input variables x, y and z. Each input variable is abstracted by an affine form which represents the range of values the real variables can take. So, following the notation of [GP13], the value of variable z at line 1 is represented by $\hat{r}^z_{[1]} = 0.5 + 0.5\epsilon^r_1$ where ϵ^r_1 is a symbolic variable whose values lies in the range $[-1, 1]$. Similarly, x at line 2 is represented by $\hat{r}^x_{[2]} = 2 + \epsilon^r_2$ and y at line 3 is represented by $\hat{r}^y_{[3]} = 1 + \epsilon^r_3$. The values u_z, u_x and u_y correspond to the error each of these real variables can have. This error is most of the time used to represent errors because of the finite-precision representation of floating-point numbers but can also come from other uncertainties such as imprecise data from sensors [GP13]. The values of u_i are assumed to be $0 \leq u_i << 1$ just as in [GP13]. The error of each variable is then given by $\hat{\epsilon}^z_{[1]} = u_z$, $\hat{\epsilon}^x_{[2]} = u_x$ and $\hat{\epsilon}^y_{[3]} = u_y$ which defines the floating-point values for each variable: $\hat{f}^z_{[1]} = \hat{r}^z_{[1]} + \hat{\epsilon}^z_{[1]} = 0.5 + 0.5\epsilon^r_1 + u_z$ ($\hat{f}^x_{[2]}$ and $\hat{f}^y_{[3]}$ are obtained similarly). Looking at variable `z` we see that `[0,1]` corresponds to $0.5 + 0.5\epsilon_1^r$ and its error is $\hat{\epsilon}_{[1]}^z = u_z$ is the symbolic term `uz` (here we omit the actual value of the error and treat it symbolically).

These forms can then be used to obtain other affine forms that correspond to the result of arithmetic operations and to interpret tests including possible test discontinuity. For example, the re-assignment of the value of variable `z` on line [4] leads to a real value of $\hat{r}_{[4]}^z = \hat{r}_{[1]}^z + 4 = 0.5 + 0.5\epsilon_1^r + 4 = 4.5 + 0.5\epsilon_1^r$ and leads to an error of $\hat{\epsilon}_{[4]}^z = \hat{\epsilon}_{[1]}^z + \delta\epsilon_4^e$ where δ bounds the rounding error on the new value of `z` due to floating-point addition. Thus, the floating point value of `z` at program location [4] becomes $\hat{f}_{[4]}^z = \hat{r}_{[4]}^z + \hat{\epsilon}_{[4]}^z$.

We proceed to line [5] where a possible discontinuity can occur in the test condition. For `x`, the conditions on both the real and floating-point values to take the `then` branch is given by $\hat{r}_{[2]}^x \leq 2 \implies 2 + \epsilon_2^r \leq 2 \implies \epsilon_2^r \leq 0$ and $\hat{f}_{[2]}^x \leq 2 \implies 2 + \epsilon_2^r + u_x \leq 2 \implies \epsilon_2^r \leq -u_x$ respectively. The condition $u_x > 0$ implies that if $\epsilon_2^r \leq -u_x$ then the computation for both the real and the floating-point values will take the same branch. For `y`'s real and floating-point values to take the `then` branch, the conditions for the test evaluating to `true` are $\hat{r}_{[3]}^y \geq 1 \implies \epsilon_3^r \geq 0$ and $\hat{f}_{[3]}^y \geq 1 \implies \epsilon_3^r \geq -u_y$. Again, since $u_y > 0$, the computation does not diverge if and only if $\epsilon_3^r \geq 0$. Then, to take the `else` branch of the conditional it must be that either $\hat{r}_{[2]}^x > 0$ or $\hat{r}_{[3]}^y < 0$ for the real case or $\hat{f}_{[2]}^x > -u_x$ or $\hat{f}_{[3]}^y < -u_y$ for the floating-point case. These sets of constraints in turn define the conditions for which test-divergence occurs for this program. Firstly, consider the case where the real program takes the `if` branch while the floating-point program takes the `else` branch. It must be that $-u_x < \epsilon_2^r \leq 0$ since if $\epsilon_2^r > -u_x$ the floating-point variable will take the `else` branch while as long as $\epsilon_2^r \leq 0$ the real variable will take the `if` branch. For the variable `y` no divergence occurs as for the real case we require $\epsilon_3^r \geq 0$ and for the floating-point case we require $\epsilon_3^r > -u_y$ which are incompatible. The conditions for the `else-if` discontinuity can be similarly obtained and this is given by the case $-u_y \leq \epsilon_3^r < 0$. In general, if Φ_r are the set of constraints on the real values of a variable and Φ_f are the ones for the floating-point value, then the unstable tests are obtained by $\Phi_r \cap \Phi_f$. Thus, we see that for this program it is possible to observe both an `if-else` divergence and an `else-if` divergence and the conditions for such divergences can be obtained from the affine forms. We also note that it is possible to bound the set of inputs that lead to the test instability as follows: $-u_x < \epsilon_2^r \leq 0$ corresponds to $2 - u_x < r^x \leq 2$ and $-u_y \leq \epsilon_3^r < 0$ corresponds to $1 - u_y \leq r^y < 1$ [GP13].

We do not outline how the affine forms in lines [6] and [8] are computed here as they are not necessary for determining the responsibility for the test divergence. Furthermore, we note that a join operator $\sqcup$ is needed for line [9] to consider the two possible values `z` can take but we do not discuss this here. The join will also take into account the issue of discontinuity and introduce two new error terms which are only accounted for if the value of $\hat{\epsilon}_{[9]}^r$ falls under the divergence conditions. A more detailed explanation of affine forms and their use in both discontinuity analysis and error estimation can be found in Section 3.3. The overall idea is that these affine forms may be used to detect when a divergence in real and floating-point control flow occurs and the constraints on the noise symbols characterizes the different possibilities of divergence and non-divergence.

2.2 Responsibility

We see that for program 2.1 while we have detected the discontinuity it is not immediately obvious who should be held responsible for this behavior. Of course, the candidate variables are `x` and `y` as they are the variables used in the test but which one of these are responsible and under what conditions? The results of the affine analysis shows the conditions under which conditional divergence occurs in terms of the uncertainty in the declared variables. For example, if variable `x` were to have zero uncertainty error, $u_x = 0$, then it might be that the conditional divergence does not occur while if u_x is close to 1 then divergence will occur for many traces. So it might make sense to vary the uncertainty and make some sort of observation. Furthermore, we might be able to obtain some information from the weakly relational property of affine forms and look at the error symbols used when the test `x <= 2 && y >= 1` is analyzed. But, it is possible that many error symbols are lost meaning in the general case we may obtain little information. However, these approaches are unsystematic which makes it difficult to design a static analysis.

Intuitively, the declarations of variables `x` and `y` should be the responsible program locations for the divergence. This is because if we were to change these variables to have zero error, i.e. $u_x = 0$ and $u_y = 0$, then the erroneous behavior would no longer occur. Regardless of the values of the uncertainty, if they are greater than zero then we are always guaranteed to cause errors. Since the uncertainty values correspond to converting a real value to a floating-point value, we may think of these variables as making a choice. Either they choose to not lose any precision or they choose their floating-point values which leads to some error. Additionally, we may also think of the other floating-point expressions including arithmetic as also making a choice. Here they may either compute their result exactly or round the result following the IEEE rules. This means that we consider all the choices that can be made in the program. In the case of Program 2.1 changes to other program locations in the form of a choice, including having zero-rounding error for arithmetic, does not have any effect on whether we would get a conditional divergence or not. Thus, it makes sense to assign responsibility to both `x` and `y` because if they chose to have zero uncertainty then the discontinuity would not occur.

There are two issues we must consider: (1) how should responsibility be defined in the context of floating-point programs, and (2) how can the responsible program entities be found with respect to this definition? For the first question the preceeding paragraph has given some intuition for this definition. To answer the second question we would ideally like to have an analysis that gives us the exact program locations that are responsible. However, this problem is undecidable due to Rice's Theorem. Therefore, we will soundly approximate the responsible entities using abstract interpretation [CC77]. Specifically, we will combine a series of analyses to show how we can obtain the responsible entities our inuition points us towards.

This thesis formally defines responsibility for floating-point programs and formulates an analysis that is able to determine the responsible program entities for all possible floating-point errors. The coming chapters will primialy focus on answering questions (1) and (2) while also presenting relevant extra material.

Chapter 3

Background Knowledge

We review the background knowledge that is used to construct the proposed analysis of this thesis. Relevant material for further reading is also pointed out.

3.1 Floating-Point Review

This section reviews the basics behind the IEEE Standard for Floating-Point Arithmetic (commonly referred to as IEEE-754) [08] [19] along with some basic facts regarding rounding.

3.1.1 Represenation

The IEEE standard is the current standard for working with finite preicison numbers and arithmetic. Let $\mathbb{F} \subset \mathbb{R}$ be the set of floating-point numbers. Then, a normalized number $\hat{x} \in \mathbb{F}$ is represented as

$$\hat{x} = (-1)^s(1 + m)b^e$$

Name	Total bits	p	k	e_{min}	e_{max}
Half precision	16	10	5	-14	15
Single precision	32	23	8	-126	127
Double precision	64	52	11	-1023	1022
Quadruple precision	128	112	15	-16382	16383

Table 3.1: IEEE754 normalized encoding formats.

where $b = 2$ or $b = 10$ is the base, s is a bit that indicates whether the number is negative or not, e is a signed integer in the range $[e_{min}, e_{max}]$ and $m = 0.b_1 \dots b_p$ is a fixed-point value in the range $[0, 1)$ which is defined using p bits. To compute e we subtract from the k-bit unsiged integer $e' = b_{k-1} \dots b_0$ the bias $2^{k-1} - 1$ where $e' \neq 0$ and $e' \neq 1$. The values of e and m are referred to respectively as the exponent and the mantissa of the floating-point number and are defined by the format of the floating-point number. Table 3.1 summarizes the standard formats of normalized floating-point numbers.

Normalized encoding is used to avoid multiple representations of the same number [Mar17]. However, we may sometimes wish to obtain numbers that are smaller in absolute value than the numbers representable in normalized encoding. For this case, IEEE754 defines denormalized numbers of the form

$$\hat{x} = (-1)^s (0 + m) b^e$$

where the exponent e is now obtained by setting $e = 0$. This format makes numbers very close to 0 representable by gradual underflow [Mul+18].

Furthermore, special values occur when $e = 0$ and $m = 0$ which gives $+0$ and -0 depending on the sign bit, $e = 1$ and $m = 0$ which gives $+\infty$ and $-\infty$ depending on the sign bit, and $e = 1$ and $m \neq 0$ which gives the value “not a number’, also known as `NaN`. In general, operations that analytically lead to an indeterminate form, such as $\infty - \infty$, will produce `NaN`.

Even though IEEE754 is defined for both $b = 2$ and $b = 10$, throughout this thesis we will assume without loss of generality $b = 2$ as this is most commonly used for floating-point number representation [Mar17].

3.1.2 Rounding

The IEEE Standard "mandates that floating-point operations be performed as if the computation was done with infinite precision and then rounded." [Bol+15] [08] In the context of representation, since not all numbers can be represented exactly (e.g. 0.1 cannot be written in base 2 with a finite number of digits), IEEE754 guarantees that a real number $x \in \mathbb{R}$ is represented by a floating-point number $\hat{x} \in \mathbb{R}$ that is the closest number to x. This guarantee is with respect to some rounding operation $\rho_\star : \mathbb{R} \to \mathbb{F}$ where $\star$ is the rounding mode [08]. The rounding modes that are given in [08] are:

- round towards nearest,
- round towards $+\infty$,
- round towards $-\infty$, and
- round towards 0.

In the case of round towards nearest, the "tablemaker's dilemma" situation arises where there is a tie between two possible numbers that could be rounded to. The two choices to solve the dilemma are:

- round to the nearest even, and

- round to the largest magnitude.

Given the value x and a fixed rounding mode $\star$, its corresponding rounded value $\hat{x}$ can be expressed as

$$\hat{x} = \rho_\star(x) = x(1+\delta_x) + \eta_x, \quad \text{with } |\delta_x| < \epsilon_M \text{ and } |\eta_x| < \epsilon_M \times 2^{e_{min}}$$

where δ is the error associated with rounding a normalized number, η is the error associated with rounding a denormalized number and $\delta \times \eta = 0$ as a number cannot be both normalized and denormalized at the same time [Gou14] [Chi+17]. The value ϵ_M is referred to as machine precision. It is the maximal relative error introduced by the rounding operation and is given by $2^{-(p+1)}$. Similarly, the value η is the absolute error associated with rounding to a denormalized number as relative error estimation does not work well for such small numbers [Chi+17]. If $x = \hat{x}$ then clearly the errors δ and η are 0.

3.1.3 Arithmetic

Elementary floating-point arithmetic operations are defined by the set $\{+, -, \times, /\}$ and standarized by the IEEE Standard [08]. We note that IEEE754 also standardizes the square root operation [08] but we do not deal with it in this thesis. For any $\hat{x}, \hat{y} \in \mathbb{F}$ and elementary floating-point operation $\circ$, let $z = \hat{x} \circ \hat{y}$ and $\hat{z} = \rho_\star(z) = \rho_\star(\hat{x} \circ \hat{y})$ where $\hat{y} \neq 0$ if $\circ = /$. Then, the IEEE Standard ensures as long as no overflow or underflow has occurred that the following holds

$$\hat{z} = \rho_\star(z) = \rho_\star(\hat{x} \circ \hat{y}) = (\hat{x} \circ \hat{y})(1+\delta_z) + \eta_z, \quad \text{with } |\delta_z| < \epsilon_M \text{ and } |\eta_z| < \epsilon_M \times 2^{e_{min}}$$

where δ is relative rounding error associated with rounding the result of arithmetic on two normalized numbers, η is the absolute error for denormalized numbers and $\delta \times \eta = 0$ [Gou14]. Overflow occurs when the computed result is larger in absolute value than the largest representable in a given floating-point format and if it occurs we may simply state that the difference between $\hat{z}$ and z is $\pm\infty$ [Gou14]. Underflow arises from the fact that numbers smaller than the machine precision ϵ_M cannot be represented in IEEE format and if a user tried to use such values then an underflow error occurs.

IEEE754 states the computed value of $\hat{x} \circ \hat{y}$ is "as good as" the rounded exact answer, meaning that the result will be equal to doing the operation in infinite precision and then rounding the result [08]. Furthermore, there is no possibility of error occurring due to $\hat{x}$ and $\hat{y}$ having different lengths (because of different formats) due to the use of guard digits [08].

3.1.4 Measuring Error

There are two ways of measuring the error between some intended result $x^* \in \mathbb{R}$ and its computed approximation $\hat{x} \in \mathbb{F}$. The first is what is referred to as the absolute error and can be calculated as

$$\epsilon_{abs} = \left| x^* - \hat{x} \right| .$$

If $x^* \neq 0$ we may also compute what is called the relative error, which is calculated by

$$\epsilon_{rel} = \frac{\epsilon_{abs}}{|x^*|}.$$

The relative error gives a better sense of how much error has occurred when comparing the errors caused by approximations of varying sizes. For example, consider having a measurement where the absolute error is always 3. Now, if the actual value is 30 then the relative error is 0.1 while if the actual value is 300000 then the relative error is 0.00001. Clearly, the relative error can distinguish the situation where the error is negligible but the absolute error says nothing about this.

Floating-point errors occur either due to the finite-precision representation of real numbers (Section 3.1.2) or arise from floating-point arithmetic which introduces errors (Section 3.1.3). In both cases, we may want to measure the error so as to ascertain that a program computes some value up to this error. For this thesis we will be using the absolute error for this measurement as our approximation will actually compute an abstraction of the absolute error [GP11].

From the perspective of static analysis, the error being tracked is not as important. One can always recover the relative error from the absolute error as most analyses will compute both a real non-rounded value for some computation and its associated error. Also, it is easier to compute the absolute error at first and not deal with the divide by zero that might occur. For example, if an analysis were to compute only the relative error of some floating-point computation it would need to handle division by zero accurately. While the solution is simply to revert back to the absolute error it is even simpler to just not compute the relative error in the first place.

3.2 Abstract Interpretation

Abstract interpretation provides a way to soundly approximate the semantics of programs. The high level idea is to abstract sets of traces of programs to an abstract domain that approximates these sets which we refer to as the concrete domain. Then, the semantics of programs may be formulated as a fixpoint which may be computed by various iteration and fixpoint approximation techniques [Cou01]. In this section we review the basic concepts necessary to design a program analysis framework using abstract interpretation.

A partial order on $\sqsubseteq$ a set S is a binary relation over the elements of S that is (1) reflexive, (2) transitive and (3) anti-symmetric. A partially ordered set, referred to as a poset, is a set S that is equipped with a partial order $\sqsubseteq$. We denote such posets with $(S, \sqsubseteq)$. Two elements of the poset $x, y \in S$ are comparable when either $x \sqsubseteq y$ or $y \sqsubseteq x$ and otherwise incomparable.

Definition 3.2.1. Let $(C, \sqsubseteq_1)$ and $(A, \sqsubseteq_2)$ be two posets. We say that the pair (α, γ) of functions $\alpha \in A \to C$ and $\gamma \in C \to A$ form a Galois connection if and only if

$$\forall x \in C, \forall y \in A.\, \alpha(x) \sqsubseteq_2 y \iff x \sqsubseteq_1 \gamma(y),$$

which we denote by

$$(C, \sqsubseteq_1) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq_2).$$

We refer to C as the concrete domain and γ as the concretization function while A is called the abstract domain and α the abstraction function.

Given a poset $(P, \sqsubseteq)$ and a subset $S \in \wp(P)$ we say that $y \in P$ is an upper bound of S if and only if $\forall x \in S.\, x \sqsubseteq y$. Furthermore, we say that $\sqcup S$ is a least upper bound, referred to as a join, of S if and only if $\sqcup S$ is an upper bound of S and $\sqcup S$ is smaller than all other upper bounds of S. Similarly, we say that $y \in P$ is a lower bound of S if and only if $\forall x \in S.\, y \sqsubseteq x$ and $\sqcap S$ is a greatest lower bound, referred to as a meet, of S if and only if $\sqcap S$ is a lower bound and $\sqcap S$ is greater than all other lower bounds of S.

A complete lattice is a poset $(P, \sqsubseteq)$ in which all subsets $S \in \wp(P)$ have a join and a meet. We denote a complete lattice by $(P, \sqsubseteq, \bot, \top, \sqcup, \sqcap)$ where $\sqcap P = \bot$ and $\sqcup P = \top$. If $S = \{x, y\}$ (i.e. a set of two elements) then we will denote $\sqcup S$ as $x \sqcup y$ and, similarly, $\sqcap S$ as $x \sqcap y$.

Given two complete lattices $(P_1, \sqsubseteq_1, \bot_1, \top_1, \sqcup_1, \sqcap_1)$ and $(P_2, \sqsubseteq_2, \bot_2, \top_2, \sqcup_2, \sqcap_2)$, a function $f : P_1 \to P_2$ is a complete join-morphism if and only if

$$\forall S \in \wp(P_1).\, f(\sqcup_1 S) = \sqcup_2\{f(x) \mid x \in S\},\ f(\bot_1) = \bot_2.$$

Again, given the above two complete lattices, a function $g : P_2 \to P_1$ is a complete meet-morphism if and only if

$$\forall S \in \wp(P_2).\, g(\sqcap_2 S) = \sqcap_1\{g(y) \mid y \in S\},\ g(\top_2) = \top_1.$$

Proposition 3.2.1 ([CC79]). Let $(P_1, \sqsubseteq_1, \bot_1, \top_1, \sqcup_1, \sqcap_1)$, $(P_2, \sqsubseteq_2, \bot_2, \top_2, \sqcup_2, \sqcap_2)$ be two complete lattices and the pair (α, γ), where $\alpha \in P_1 \to P_2$ and $\gamma \in P_2 \to P_1$, form a Galois connection. Then each function in the pair uniquely determines the other

$$\alpha(x) = \sqcap_2\{y \in P_2 \mid x \sqsubseteq_1 \gamma(y)\}$$
$$\gamma(y) = \sqcup_1\{x \in P_1 \mid \alpha(x) \sqsubseteq_2 y\}$$

Also, α is a complete join-morphism and γ is a complete meet-morphism.

Proposition 3.2.2 ([CC79])**.** The following statements are equivalent:

1. (α, γ) is a Galois connection,
2. α and γ is monotone, $\alpha \circ \gamma$ is reductive $(\forall y \in P_2.\, \alpha(\gamma(y)) \sqsubseteq_2 y)$ and $\gamma \circ \alpha$ is extensive $(\forall x \in P_1.\, x \sqsubseteq_1 \gamma(\alpha(x)))$,
3. α is a complete join-morphism and γ is determined by α by Proposition 3.2.1,
4. γ is a complete meet-morphism and α is determined by γ by Proposition 3.2.1.

Proposition 3.2.3 ([CC79])**.** α is onto if and only if γ is one-to-one if and only if $\alpha \circ \gamma = \lambda y.\, y$.

Definition 3.2.2 ([CC79])**.** Let $(P, \sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice. Then, the function $\nabla : P \times P \to P$ is called a widening operator if for all $x, y \in P$ $x \sqsubseteq x \nabla y$ and $y \sqsubseteq x \nabla y$ and for all increasing chains $x_0 \sqsubseteq x_1 \sqsubseteq \ldots$, the increasing chain $y_0 = x_0, \ldots, y_i = y_{i-1} \nabla x_i$ for all $i > 0$ is not strictly increasing.

Given a function f over a poset $(P, \sqsubseteq)$ and $x \in P$, we denote $\mathbf{lfp}_x^{\sqsubseteq} f$ as the least fixpoint of that function where the function is computed as $f(\ldots(f(f(x))))$.

Proposition 3.2.4. Let $(P, \sqsubseteq, \perp, \top, \sqcup, \sqcap)$ be a complete lattice, $F : P \to P$ be a monotone function and ∇ a widening operator for P. Then, the sequence

$$\begin{aligned} X_0 &= \perp \\ X_{i+1} &= X_i && \text{if } F(X_i) \sqsubseteq X_i \\ &= X_i \nabla F(X_i) && \text{otherwise} \end{aligned}$$

is ultimately stationary with the limit X such that $\mathbf{lfp}_{\perp}^{\sqsubseteq} F \sqsubseteq X$ where the least fixpoint $\mathbf{lfp}_{\perp}^{\sqsubseteq} F$ of F exists due to [Tar+55].

3.3 Affine Sets for Floating-Point Analysis

We introduce here the necessary background knowledge to understand constrained affine sets [GGP10] which we will use as the underlying domain to compute invariants for floating-point programs. The section begins by presenting affine forms and affine arithmetic and then moves onto a series of subsections that describes the necessary machinery to understand constrained affine sets for floating-point analysis. An example of the analysis of a floating-point program using constrained affine forms can be found in Section 2.1.

3.3.1 Affine Forms

Affine forms, first introduced in [CS93], are a sum over a set of noise symbols ϵ_i of the form

$$\hat{x} = \alpha_0^x + \sum_{i=1}^{n} \alpha_i^x \epsilon_i$$

where each $\alpha_i^x \in \mathbb{R}$ and each ϵ_i is an unknown quantity bounded in the range $[-1, 1]$. Thus, "each noise symbol is an independent component of the total uncertainty" on the sum written above [GGP10]. The coefficients α_i^x are known real values and they express the magnitude of a given symbol ϵ_i. If more than one variables which are assigned an affine form share some common symbol then these symbols can express an implicit dependency between these variables, hence making the affine forms domain weakly relational [GP11].

Given an affine form $\hat{x}$, its concretization, which defines the range of values the affine form can take, is given by

$$\gamma(\hat{x}) = [\alpha_0^x - \sum_{i=1}^{n} |\alpha_i^x| \, , \alpha_0^x + \sum_{i=1}^{n} |\alpha_i^x|]$$

if each symbol is constrained in $[-1, 1]$. However, in [GGP10] and [GP13] the constrained noise symbols may actually be refined to ranges smaller than $[-1, 1]$. For example, we may have the range $[-1, 0]$ for ϵ_i. In that case, we need to write down a more general form of the concretization and for that we define the functions $u([a, b]) = b$ and $l([a, b]) = a$ to retrieve the upper and lower bounds of some noise symbol constrained in the range $[a, b]$. Now, the concretization function becomes

$$\gamma(\hat{x}) = [\alpha_0^x - \sum_{i=1}^{n} \alpha_i^x l(\epsilon_i), \alpha_0^x + \sum_{i=1}^{n} \alpha_i^x u(\epsilon_i)].$$

To abstract a set of real number $\mathbb{R}$ to an affine form it is enough to first abstract into an interval using the interval abstraction [CC76] to obtain the interval $[a, b]$ and then abstract the interval into an affine form using

$$\alpha_i([a, b]) = \frac{a+b}{2} + \frac{b-a}{2}\epsilon_i.$$

We parameterize the abstraction function α by i as we may want to control symbolically the name of the noise symbol. That is, if we have variables in affine forms that contain the symbols $\epsilon_1, \ldots, \epsilon_n$ and we wish to abstract a new variable that has no relation to any of our previous variables, then we should abstract this new value using α_{n+1} so as to not introduce false relationships between variables. We let the set $\mathbb{AR}$ be the set of affine forms. The order relation between any two elements is given by

$$\forall x, y \in \mathbb{AR}.\, x \sqsubseteq_{\mathbb{AR}} y \iff \gamma(x) \sqsubseteq_{\mathbb{IR}} \gamma(y)$$

where $\sqsubseteq_{\mathbb{IR}}$ is the interval order relation which is given by

$$\forall [a,b],[c,d] \in \mathbb{IR}.\, [a,b] \sqsubseteq_{\mathbb{IR}} [c,d] \iff a \geq c \wedge b \leq d.$$

3.3.2 Affine Arithmetic

Given two affine forms $\hat{x}$ and $\hat{y}$, we can also perform arithmetic on them. If the operation we are doing is linear, i.e. addition or subtraction, then the result is also a linear affine form [GP08]. Thus, given a real number λ we get

$$\lambda\hat{x} + \hat{y} = (\lambda\alpha_0^x + \alpha_0^y) + \sum_{i=1}^{n} (\lambda\alpha_i^x + \alpha_i^y)\epsilon_i.$$

For non-linear operations we must select an approximate affine form as our answer [GP11]. The following approximate form is used for multiplication operations

$$\hat{x}\hat{y} = (\alpha_0^x\alpha_0^y) + \sum_{i=1}^{n} (\lambda\alpha_i^x\alpha_0^y + \alpha_i^y\alpha_0^x)\epsilon_i + (\sum_{i=1}^{n} |\alpha_i^x\alpha_i^y| + \sum_{i<j}^{n} \left|\alpha_i^x\alpha_j^y + \alpha_j^x\alpha_i^y\right|)\epsilon_{n+1}$$

where the new noise symbol ϵ_{n+1} is introduced to bound the error created by the linearizing form [GP08].

3.3.3 Affine Sets

Let $\mathbb{M}^{n,m}$ be the space of matrices with n rows and m columns and we will say each element has a dimension of $n \times m$. Each matrix $M \in \mathbb{M}^{n,m}$ has entries $M_{i,j} \in \mathbb{R}$. A set of m affine forms over n noise symbols $\epsilon_1, \ldots, \epsilon_n$ can be represented by a matrix $M \in \mathbb{M}^{n+1,m}$ [GGP10]. The concretization $\gamma_{\mathbb{M}}$ of M defines a zonotope [GGP10].

Definition 3.3.1 ([GGP10])**.** Let $M \in \mathbb{M}^{n+1,m}$ be the matrix that defines m affine forms over n noise symbols. Then, the resulting zonotope from its concretization is given by

$$\gamma_{\mathbb{M}}(M) = \{M^\intercal e^\intercal \mid e \in \mathbb{R}^{n+1}, e_0 = 1, \|e\|_\infty = 1\} \subseteq \mathbb{R}^m.$$

To be able to define an order relation that preserves input/output relations we now define two zonotopes: (1) a central zonotope $\gamma_{\mathbb{M}}(C^X)$ and a pertubation zonotope $\gamma_{\mathbb{M}}(P^X)$ centered around 0 [GGP10]. So, we will represent an affine set X with noise symbols ϵ_i^r capturing uncertainty on the inputs to the program. These symbols will be contained in the central zonotope and the goal is to retain as many implicit relations as possible [GGP10]. Now, we will also define noise symbols ϵ_j^e which represent uncertainty that arises due to the abstraction of control-flow when carrying out a computation (for example, taking the join of two abstract elements after evaluating a conditional branch) [GGP10]. These symbols will be contained in the pertubation zonotope.

Definition 3.3.2. An affine set is defined by the pair of matrices $(C^X, P^X) \in \mathbb{M}^{n+1,m} \times \mathbb{0}, \mathbb{m}$. There are $1 \leq k \leq m$ variables in this set and each is defined by the affine form

$$X_k = c^X_{1,k} + \sum_{i=2}^{n+1} c^X_{i,k}\epsilon^r_i + \sum_{j=1}^{m} p^X_{i,k}\epsilon^r_j$$

where $c^X_{i,k}$ is the i, k-th entry from C^X, $p^X_{j,k}$ is the j, k-th entry from P^X and X_k is the k-th variable in the affine set [GGP10].

Now, the order relation between two variables may be defined by the matrix norm induced by vectors at $u \in \mathbb{R}^m$. We do not outline the relation here and instead refer the reader to look Definition 3 from [GGP10] and [GP08] for more details. For this thesis, we will assume that the order relation between two affine sets from [GGP10] and denote it as $\sqsubseteq_{\mathbb{X}}$ where $\mathbb{X}$ is the space of affine sets. When it is obvious from the context $\sqsubseteq_{\mathbb{X}}$ is being used in the subscript $\mathbb{X}$ may be dropped. Finally, we note that this order relation is "slightly more strict than the concretization inclusion" $\sqsubseteq_{\mathbb{AR}}$ as it takes into account the fact that the central noise symbols define an input-output relationship [GGP10].

3.3.4 Constrained Affine Sets from Zonotopes

Zonotopes describe affine sets where each noise symbol is constrained in the range $[-1, 1]$. While this is useful, we may also wish to refine such constraints to limit the set of possible values the affine set defines. For example, limiting the range of constraints will be useful in defining a new affine set after an if branch has been taken as we wish for the affine set to express a subset of the original values it initially represented. To this end, [GGP10] introduces constrained affine sets (mainly for the purpose of definining the intersection of affine sets). Let $\mathcal{A}$ be any lattice $(\mathcal{A}, \sqsubseteq_{\mathcal{A}}, \sqcup_{\mathcal{A}}, \sqcap_{\mathcal{A}})$ that is used to abstract the values of the noise symbols ϵ^r_i and ϵ^e_i [GGP10]. Some possible candidates for $\mathcal{A}$ include intervals [CC76], octagons

[Min06] or polyhedra [CH78]. Since $\mathcal{A}$ is used to abstract the values of the noise symbols we require a concretization function

$$\gamma_{\mathcal{A}} : \mathcal{A} \to \mathcal{P}(\{1\} \times \mathbb{R}^n \times \mathbb{R}^o)$$

which will concretize n noise symbols of the form ϵ_i^r and o noise symbols of the form ϵ_j^e. This function exists for the mentioned candidates for $\mathcal{A}$. Furthermore, we assume the existence of an abstraction function $\alpha_{\mathcal{A}}$ that need not necessarily be the most precise one (e.g. for polyhedra the most precise abstraction does not exist but for intervals it does) [GP13]. Thus, constrained affine sets are defined as follows.

Definition 3.3.3 ([GGP10]). A constrained affine set is defined by the pair $X = (C^X, P^X, \Phi^X)$ where (C^X, P^X) is an affine set and Φ^X is an element of $\mathcal{A}$ [GGP10]. Given two constrained affine sets $X = (C^X, P^X, \Phi^X)$ and $Y = (C^Y, P^Y, \Phi^Y)$, $X \sqsubseteq Y$ if and only if $\Phi^X \sqsubseteq_{\mathcal{A}} \Phi^Y$ and $(C^X, P^X) \sqsubseteq_{\mathbb{X}} (C^Y, P^Y)$ with respect to the constraints Φ^X and Φ^Y.

The most common instantiation of $\mathcal{A}$ is boxes as it provides a combination of good performance and accurate results [GGP09]. While using octagons or polyhedra could lead to more accurate results, the decrease in performance can become costly.

3.3.5 Constrained Affine Sets for Floating-Point Analysis

Affine sets describe an interval as the sum of symbolic terms e_i where each term is viewed as some noise limited to the range $[-1, 1]$. In the case of floating-point analysis, we will keep track of two independently acting sets. The first keeps track of the value of program variables with respect to real semantics and the second keeps track of the floating-point value associated with each real value. The floating-point forms may be viewed as a pertubation of the real forms [GP13].

Let R be the set of constrained affine forms where the noise symbols are only of the form ϵ_i^r such that R represents the real values of program variables. Similarly, let E be the set of constrained affine forms where the noise symbols can consists of a combination of ϵ_i^r and ϵ_i^e which means that E is the corresponding floating-point pertubation of the elements of R. Thus, the floating-point values can be obtained by computing $R + E$.

The sets of R and E are sufficient to compute the results of arithmetic operations by using affine arithmetic on linear expressions (such as addition) and linearizing non-linear ones (such as multiplication). However, to interpret tests accurately (such as <=) we need to be able to further refine the $[-1, 1]$ range each noise symbol has. As an example, consider the test from Section 2.1 where we have the affine form $2 + \epsilon_2^r$ and we wish to determine when it is ≤ 2. We see that $2 + \epsilon_2^r \leq 2$ means that ϵ_2^r must be ≤ 0 such that we constrain ϵ_2^r to the smaller interval of $[-1, 0]$. We will consider the constraints on a real and its corresponding floating-point value so we need to have two sets of constraints on the noise symbols. Φ_r are the constraints on the noise symbols when considering the real control flow while Φ_f are the constraints on the noise symbols when considering the finite precision control flow. Thus, we must use the constrained affine sets described in the previous section.

Definition 3.3.4. Let n be the number of real noise symbols ϵ_i^r and m be the number of error noise symbols ϵ_j^e. Given a program location that has p variables $x_1, \ldots, x_p$, the abstract value X is the tuple $X = (R^X, E^X, \Phi_r^X, \Phi_f^X) \in \mathbb{M}^{n+1,p} \times \mathbb{M}^{n+m+1,p} \times \mathcal{A} \times \mathcal{A}$ which describes the constrained affine values for each program variable along with constraints on all noise symbols. Then, R^X is the $(n+1) \times p$ matrix where the k-th column corresponds to the $n+1$ co-efficients of the real noise symbols for variable x_k and E^X is the $(n+m+1) \times p$ matrix where the k-th column corresponds to the $n+m+1$ co-efficients of the noise symbols of the real and error noise symbols for variable x_k. So, for all $k = 1, \ldots, p$ we have:

$$\begin{cases} R^X \ : \ \hat{r}_k^X = r_{1,k}^X + \sum_{i=2}^{n+1} r_{i,k}^X \epsilon_i^r & \text{where } e^r \in \Phi_r^X \\ E^X \ : \ \hat{e}_k^X = e_{1,k}^X + \sum_{i=2}^{n+1} e_{i,k}^X \epsilon_i^r + \sum_{j=1}^{m} e_{n+j,k}^X \epsilon_j^e & \text{where } (e^r, e^e) \in \Phi_f^X \\ \qquad \hat{f}_k^X = \hat{r}_k^X + \hat{e}_k^X & \end{cases}$$

This definition is almost the same as the definition given in [GP13] but here we omit the discontinuity terms D^X that are used to track errors due to test divergence. The reason for this is that our partitioning scheme will keep track of these terms instead and this will become clearer in Section 4.

Note that the symbols ϵ_i^r are used in an overloaded way. When referring to elements of R each ϵ_i^r models a real value while when referring to elements of E each ϵ_i^r models the uncertainty on the real value due to the numbers being floating-point. Then, for the latter case, ϵ_i^e models the uncertainty on the errors caused by rounding. Therefore, the floating-point value of some program variable is a pertubation of its real value [GP13].

We also need to define the transfer functions for arithmetic. These transfer functions are similar to those described in Section 3.3.2 but since we have additional information regarding the noise symbols they can be used to derive bounds for the linearization of the non-linear part of multiplication operations [GGP10]. Furthermore, just like in [GP11], extra error terms that describe the error due to floating-point operations must be introduced when computing new values for E^X. We will not detail transfer functions here and instead use them "as is" from the existing literature.

Definition 3.3.5 ([GGP10]). Let $X = (R^X, E^X, \Phi_r^X, \Phi_f^X) \in \mathbb{M}^{n+1,p} \times \mathbb{M}^{n+m+1,p} \times \mathcal{A} \times \mathcal{A}$ be an abstract element for a program with p variables $x_1, \dots, x_p$. Then, the function $\mathcal{AS}$ that computes the following:

1. The result of assigning to a new variable x_{p+1} the value of $[a,b] + u$, denoted $Z = \mathcal{AS}[\![x_{p+1} = [a,b] + u]\!]X$.

2. The result of adding two variables x_i and x_j and assigning the result to a new variable x_{p+1}, $Z = \mathcal{AS}[\![x_{p+1} = x_i + x_j]\!]$.

3. The result of multiplying two variables x_i and x_j and assigning the result to a new variable x_{p+1}, $Z = \mathcal{AS}[\![x_{p+1} = x_i \times x_j]\!]$.

For point 1 of Definition 3.3.5 Z is a matrix $Z \in \mathbb{M}^{n+2,p+1} \times \mathbb{M}^{n+m+2,p+1} \times \mathcal{A} \times \mathcal{A}$ and two new constraints ϵ_{n+1}^r and ϵ_{m+1}^e are generated to obtain the new set of constraints Φ_r^Z and Φ_f^Z. The first symbol models the uncertainty in the value of $[a,b]$ while the second one models the error u due to finite-precision representation.

The matrix Z in point 2 of Definition 3.3.5 is of the form $Z \in \mathbb{M}^{n+1,p+1} \times \mathbb{M}^{n+m+2,p+1} \times \mathcal{A} \times \mathcal{A}$ and a new constraint ϵ_{m+1}^e is generated to obtain Φ_r^Z and Φ_f^Z. This new symbol models the round-off error due to finite-precision arithmetic. We may use the rounding facts from Section 3.1.3 to bound this error.

The resulting matrix $Z \in \mathbb{M}^{n+2,p+1} \times \mathbb{M}^{n+m+3,p+1} \times \mathcal{A} \times \mathcal{A}$ for point 3 of Definition 3.3.5 is a little more involved. Three new constraints ϵ_{n+1}^r, ϵ_{m+1}^e and ϵ_{m+2}^e are generated to obtain Φ_r^Z and Φ_f^Z. The first symbol models the uncertainty in the new value x_{p+1} while the second models the error due to the linearization of the non-linear multiplication (see Section 3.3.2) while the third models the error due to finite-precision arithmetic which can be bound, again, using Section 3.1.3.

For more details on transfer functions for general constrained affine sets see [GGP10] while for the semantics of transfer functions for the domain defined in Definition 3.3.4 see [GP11]. Compared to [GGP10], [GP11] must introduce more error terms due to finite-precision arithmetic.

Definition 3.3.5 is enough to define all types of arithmetic operations. For example, re-assignment to a variable can be done by slightly modifying the definition so that it replaces a column p rather than creating a new column $p+1$. Thus, the function $\mathcal{AS}$ is general enough to compute the result of assignment to constrained sets and the addition, subtraction and multiplication of elements of constrained affine sets. Constants, such as 3.14, may also be made affine terms first before using $\mathcal{AS}$.

We also define the semantics of tests. When computing conditional tests we would like to also obtain constraints on the values of the real and floating-point values of variables that restrict the values of the affine forms to those that only pass the test. This can be used to compute the set of values that lead to a test discontinuity when evaluating a test. Test discontinuity occurs when a test leads to the control flow of a program taking an if or else branch in real semantics but the opposite branch in floating-point semantics. As mentioned before, [GP13] is sound even when discontinuity happens so we must take it into account in the abstract domain. So, the semantics of tests will evaluate a test under a given set of constraints Φ_r and Φ_f and produce a new set of constraints Φ'_r and Φ'_f respectively that refine the initial constraints. The affine forms themselves are not changed but the value obtained by their concretization may now lie in a smaller range due to test. Since we evaluate tests independently under Φ_r and Φ_f, we will obtain the set of conditions that the real and floating-point values of a variable will take a branch. This in turn can be used to discover any possible discontinuities.

Definition 3.3.6 ([GP13])**.** Given a constrained affine set $X = (R^X, E^X, \Phi_r^X, \Phi_f^X)$ over p variables, the semantics of tests is given by the function $\mathcal{BS}$ where the result is given by $Z = \mathcal{BS}[\![e1\ op^B e2]\!]X$ for $op^B \in \{\leq, <, \geq, >, =, \neq\}$. To compute Z we first compute $Y = \mathcal{AS}[\![x_{p+1} = e1 - e2]\!]X$ which then we use to compute $Z = drop_{p+1}(\mathcal{BS}[\![x_{p+1}\ op^B\ 0]\!]Y)$. The function $drop_{p+1}$ takes an affine set of $p+1$ variables and returns an affine set of p variables where the $p+1$ (intermediary) variable has been removed from the set. All that remains is to define $\mathcal{BS}[\![\mathtt{x}_k\ op^B\ \ \mathtt{0}]\!]$ and this is given by:

$$
\begin{aligned}
&(R^Z, E^Z) = (R^X, E^X)\\
&\qquad \Phi_r^Z = \Phi_r^X \cap \alpha_{\mathcal{A}}(\epsilon^r \mid r_{0,k}^X + \sum_{i=1}^{n} r_{i,k}^X \epsilon_i^r\ op^B\ 0)\\
&\qquad \Phi_f^Z = \Phi_f^X \cap \alpha_{\mathcal{A}}((\epsilon^r, \epsilon^e) \mid r_{0,k}^X + e_{0,k}^X + \sum_{i=1}^{n}(r_{i,k}^X + e_{i,k}^X +)\epsilon_i^r + \sum_{j=1}^{m} e_{n+j,k}^X \epsilon_j^e\ op^B\ 0)
\end{aligned}
$$

For more details on test interpretation for general constrained affine sets see [GGP10].

To fully define the constrained affine sets for floating-point analysis we also need to define a join, meet and widening operator for two constrained affine sets. For the join $\sqcup$ we use the join operator for constrained affine sets introduced in [GLP12]. This join, at a high level, tries to keep as much as possible of the relationships between variables and throws away any relationships it cannot keep by replacing them with new symbols. However, unlike other relational domains such as octagons a lot more information may be lost due to joins and the join we use here is only optimal in some settings [GLP12]. In the context of floating-point analysis the join for constrained affine sets occurs point-wise on the co-efficient matrices using the union of the appropriate set of constraints [GP13]. Here we omit the join of the discontinuity terms D^X as they are no longer computed.

For the meet $\sqcap$ we use the meet operator for constrained affine sets introduced in [GGP10] which is used to compute the results of tests as described in Definition 3.3.6. The general idea is to keep all noise symbols and refine the constraints on the symbols such that the two affine sets intersect.

Finally, [GGP09] defines a widening ∇ for constrained affine sets. ∇ works by keeping only the noise symbols with equal coefficients in two iterates and collapses the rest into new error symbols. The common way to utilize this widening is to unroll a loop for some N amount of iterations and apply no widening. Then, if a fixpoint is not reached in N iterations, ∇ may be used for the rest of the iterates. The number of iterates is a heuristic. For example, in [GGP09] N is set to 100 such that loops are unrolled 100 times before ∇ is used and a fixpoint is reached. Throughout the thesis we use these operators "as is" and for more details we refer the reader to the relevant citations.

3.3.6 Summary

Starting with [Gou01] sets of affine forms have been used in a variety of works for the analysis of floating-point programs with the framework evolving over time. Constrained affine forms can be thought of as a functional abstraction as the affine form represent a function from inputs variables to their output values [GP13] where the function is computed using the symbolic variables ϵ_i^r and ϵ_j^e introduced for each program location and variable by the analysis. The concretization of the elements of the domain correspond to zonotopes [GP11]. Additionaly, this functional abstraction also means that the domain is weakly relational [Min04b] just like other weakly relational numerical abstract domains such as octagons [Min06].

For this thesis we will use the constrained affine sets presented in Section 3.3.5 and more details can be found in [GP11] and [GP13].

3.4 Responsibility Analysis

We review responsibility analysis in tandem with how responsibility analysis would work in the context of floating-point programs. The explanation takes Program 2.1 as an example program and illustrates how responsibility analysis would work.

3.4.1 Definition of Responsibility

In Section 2.1 the question "how should responsibility be defined in the context of floating-point programs?" was posed. To answer this, we use the notion of responsibility introduced in [DC19] which proposes a novel definition of responsibility derived from the trace semantics of a program. In [DC19] traces are defined using events which in turn are actions in a system. Some examples of actions include reading input from the user, assigning a value to a variable or conditional branching. For floating-point programs we use the same definition of entities and the main events we are interested in are assigning values to variables, floating-point arithmetic and conditional tests that involve floating-point values. In summary, and adopting the notation used in [DC19], a responsible program entity E_R is one that is free to choose its values at its discretion, for example via inputs from a user. Then, E_R is responsible for behavior $\mathfrak{B}$ in a given trace if and only if, the choice of E_R's value is the first one amongst other responsibile entities that guarantees that $\mathfrak{B}$ occures in that trace. [DC19] details how this notion of responsibility can be defined as an abstract interpretation of the program's event trace semantics [CC77].

In the analysis of floating-point programs, the responsible entities of a program consist of each floating-point variable of the program along with the corresponding uses of these variables in floating-point arithmetic expressions and tests. For Program 2.1 the responsible entities then correspond to lines 1 through 6 and line 8. The program is interpreted with real semantics meaning the values of both the variables and the results of the arithmetic operations range over the elements of the real numbers, $\mathbb{R}$. Then, the choices that these entities can make are that they either exactly compute a result or round the result using some rounding operator $\rho : \mathbb{R} \to \mathbb{R}$. ρ is a function that takes any element $x \in \mathbb{R}$ and rounds it such that $\rho(x) \in \mathbb{R}$ is also an element of the floating-point numbers $\mathbb{F}$. Since floating-point numbers represent a finite subset of the real numbers, this operation corresponds to converting the result of some program expression to a floating-point value. For example, $\rho(\hat{r}^{z}_{[1]}) = \hat{f}^{z}_{[1]}$ and if every program location uses ρ then we end up with the corresponding floating-point semantics of the program. The details for the rounding-mode [Gol91] are abstracted away here but we assume ρ gives results depending on some fixed rounding as defined by the rounding modes in Section 3.1.2. So, for a single trace if an entity chooses its value to be a floating-point one and this choice is the first one that guarantees $\mathfrak{B}$ then this entity is responsible for $\mathfrak{B}$. For a given set of traces there may be more than one responsible entity for behavior $\mathfrak{B}$, but for any single one trace there will always be one entity that is

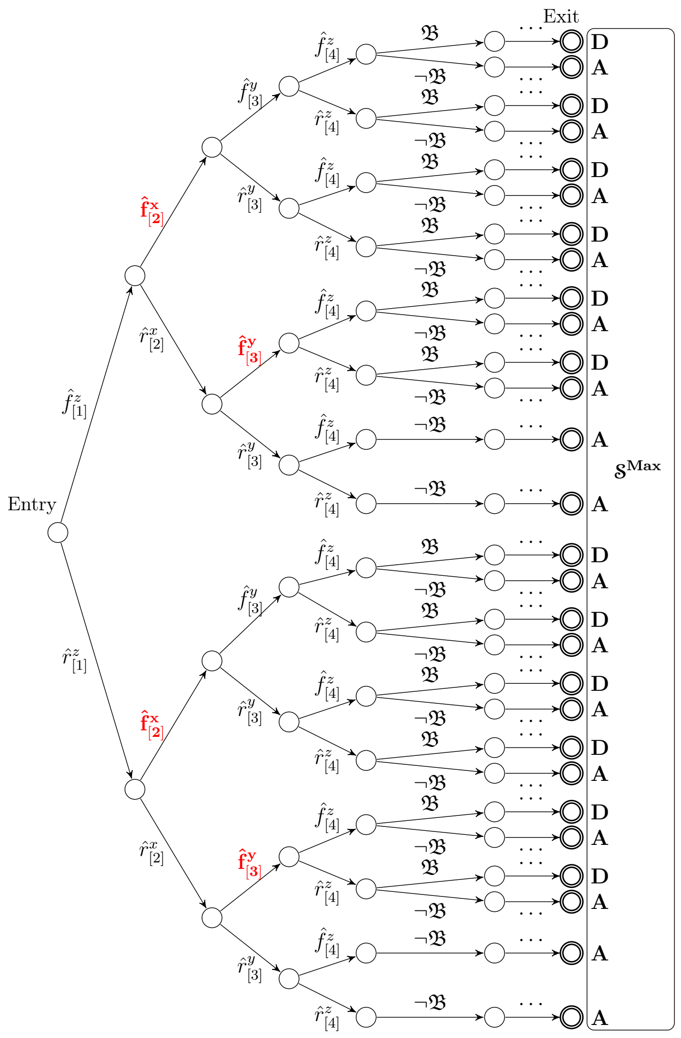


Figure 3.1: Possible traces of Program 2.1.

responsible for $\mathfrak{B}$. The determination of the responsible entity for any one trace does not only depend on that trace alone but on the semantics of the whole system. This makes responsibility a hyper-property [CS10], in contrast to a trace property, of the system [DC19].

We illustrate the concrete semantics of responsibility analysis for the program given in Program 2.1 with Figure 3.1. A more detailed treatement of how responsibility analysis works in general can be found in Section 3.4. The analysis begins by building the trace semantics of Program 2.1 and obtaining a lattice of system behaviors such that the divergent and non-divergent program traces can be identified. To make the example simpler, the join that occurs at line 9 of Program 2.1 is omitted and it is assumed that the program terminates when it reaches either lines 6 or 8. We overload $\hat{r}^z_{[1]}$ and $\hat{f}^z_{[1]}$ to denote the set of all prefixes of the program's traces that end with either z picking a real or floating-point value. In fact, Figure 3.1 shows a representation of an infinitely branching tree where each branch corresponds to a prefix of a trace for each possible value a variable may pick and the nodes marked as Exit complete the trace. We denote by $\mathfrak{B}$ and $\neg\mathfrak{B}$ all the possible traces that satisfy the condition $-u_x < \epsilon^r_2 \leq 0 \vee -u_y \leq \epsilon^r_3 < 0$ and $\neg(-u_x < \epsilon^r_2 \leq 0 \vee -u_y \leq \epsilon^r_3 < 0)$, respectively, where the former indicates that there was a conditional divergence (either if-else or else-if). For the sake of brevity, the updating of z to either $z_{[6]}$ or $z_{[8]}$ is abbreviated by the transition labeled with "$\ldots$".

3.4.2 Lattice of System Behaviors

Let $\mathcal{S}^{\text{Max}}$ be the set of all possible maximal traces of Program 2.1. **D** corresponds to the set of maximal traces where the behaviors of the real and floating-point programs diverge, while **A** denotes those where no divergence occurs. Similar to [DC19], we note that the traces we refer to are event traces. An example prefix event trace for Program 2.1 would be $\rho($`z := [0,1] + uz`$) \triangleright$ `x := [1,3] + ux` $\triangleright$ `y := [0,2] + uy`. $\triangleright$ separates events. Instead of using events, the tree of traces in Figure 3.1 is labeled by the values chosen (float vs real) and whether there was a test discontinuity or not to make the presentation succinct. Converting Figure 3.1 to a actual event traces is a simple task where we replace each label

$$\begin{tikzcd}
 & \top = \mathcal{S}^{\text{Max}} \arrow[dl, dash] \arrow[dr, dash] & \\
\neg\mathfrak{B} = \mathbf{A} \arrow[dr, dash] & & \mathfrak{B} = \mathbf{D} \arrow[dl, dash] \\
 & \bot = \emptyset &
\end{tikzcd}$$

Figure 3.2: Lattice of systems behaviors for Program 2.1.

with its corresponding event. For example, $\hat{r}^{z}_{[1]}$ becomes `x := [0,1]` as there is no uncertainty in the input while $\hat{f}^{z}_{[1]}$ becomes `x := [0,1] + uz` to reflect the initial error with a floating-point variable. It is clear that $\mathbf{D} \cup \mathbf{A} = \mathbf{\mathcal{S}^{Max}}$ and $\mathbf{D} \cap \mathbf{A} = \emptyset$ which means that the lattice of system behaviors given in Figure 3.2 can be built where the elements of the lattice are now “responsibility properties,” i.e. elements of the powerset of the set of maximal traces of the program.

The analysis now boils down to finding for any given trace the first entity that leads to the trace property $\mathfrak{B}$. This is achieved by first abstracting the maximal traces into prediction prefix traces, which maps the traces that define the property $\mathfrak{B}$ to a set of prefix traces that lead to $\mathfrak{B}$. This abstraction allows us to infer from any prefix trace the strongest possible property by the use of an inquiry function that maps prefixes to properties [DC19]. Then, a cognizance function is given which defines whether an observer of a trace, such as an attacker, can distinguish between given traces. For floating-point programs, observers have omniscient cognizance meaning they can distinguish between any two traces. Together, these two functions define the observation function used for responsibility abstraction [DC19]. The observation function takes a property B and for each maximal trace that defines B computes a single responsible entity. Note that it is actually also possible to find the responsible entities for the behavior $\neg\mathfrak{B}$. That is, we can determine program entities that guarantee $\neg\mathfrak{B}$ occurs.

In Figure 3.1, the first choices that lead to an erroneous behavior are in bold font and highlighted in red. We see that in the case that the variable $x_{[2]}$ chooses its floating-point value $\hat{f}^2_{[x]}$ we are guaranteed the divergent behavior $\mathfrak{B}$ under the constraints $-u_x < \epsilon^r_2 \leq 0$, and if $x_{[2]}$ chooses its real value then if variable $y_{[3]}$ chooses its floating-point value $\hat{f}^3_{[y]}$ we are guaranteed divergence under the constraints $-u_y \leq \epsilon^r_3 < 0$. Or in other words, if we restrict our attention to the traces with the constraints $-u_x < \epsilon^r_2 \leq 0 \vee -u_y \leq \epsilon^r_3 < 0$ then if $x_{[2]}$ chooses $\hat{f}^2_{[x]}$ divergence always occurs and, if not, then if $y_{[3]}$ chooses $\hat{f}^3_{[y]}$ divergence always occurs. No other entities of Program 2.1 under all constraints derived by the affine analysis will lead to $\mathfrak{B}$ in any choice they make. Thus, entities $x_{[2]}$ and $y_{[3]}$ are responsible entities of $\mathfrak{B}$ for Program 2.1.

$$\begin{tikzcd}
& \top = \mathcal{S}^{\text{Max}} \arrow[dl, dash] \arrow[dr, dash] & & \\
\neg\mathfrak{B} = \mathbf{A} \arrow[ddr, dash] & & \mathfrak{B} = \mathbf{D} \arrow[dl, dash] \arrow[dr, dash] & \\
& \mathbf{D}_{if-else} \arrow[d, dash] & & \mathbf{D}_{else-if} \arrow[dll, dash] \\
& \bot = \emptyset & &
\end{tikzcd}$$

Figure 3.3: Refined lattice of systems behaviors for Program 2.1.

3.4.3 Refining the Lattice

We note that $\mathfrak{B}$ could have been refined to the cases of if-else divergence and else-if divergence by taking the set $\mathbf{D}$ and splitting into the cases that correspond to each type of discontinuity. In this case, the lattice Figure 3.2 will get two new elements which are beneath **D** and above $\perp$ which allows for a finer analysis. The refined lattice is given in Figure 3.3. However, since we only wish to show responsible entities for any type of discontinuity here, Figure 3.2 works well.

3.4.4 Summary

The definition of responsibility and the semantics introduced here allows us to precisely define responsibility for floating-point errors. We see that in the case of Program 2.1, the responsible entities are $x_{[2]}$ and $y_{[2]}$, which correspond to the variables `x` and `y`. This is an intuitive result as we see that the affine analysis determines the conditions on the noise symbols associated to these variables that leads test divergence. We note that we are able to derive the same conditions derived from the affine analysis. Furthermore, if we were to drop the affine symbols and choose a different representation of the numerical values we would derive constraints isomorphic to the ones we derived here meaning we are not limited to using the affine analysis for the responsibility analysis of floating-point programs.

However, we should note that [DC19] only defines a concrete semantics for responsibility which is not computable in general. Thus, we require an abstract responsibility analysis for floating-point programs which is one of the contributions of this paper.

3.5 Trace Partitioning

When designing a static analysis by abstract interpretation a common strategy is to abstract program traces and prove properties on the set of reachable states

[RM07]. For example, to obtain the set of values a program computes one can compute an overapproximation of the set of reachable states and a value associated with each state using interval analysis [CC76]. However, when computing such reachable states the analysis loses information regarding the flow of computation [RM07]. This makes it difficult to prove certain properties about programs as the result from the analysis is too coarse. For example, consider Program 3.4, taken from [RM07], that is analyzed using interval analysis.

```
ℓ0: int x, y, s;
ℓ1: if (x < 0) {
ℓ2:  s = -1;
ℓ3: } else {
ℓ4:   s = 1;
ℓ5: }
ℓ6: y = x / s;
```

Program 3.4: Motivating example program for trace partitioning.

Clearly a divide by zero error cannot occur as the variable `s` is either -1 or 1 depending on the branch taken. However, an interval analysis would not discover this issue. The assignment `s = -1` will be abstracted as the interval $[-1, -1]$ and `s = 1` will be abstracted as $[1, 1]$. The join of these intervals upon exit of the conditional branch is $[-1, 1]$ and since $0 \in [-1, 1]$ the analysis will report an alarm (i.e. there is a possible error). This is a well known issue with the use of the interval domain as joins may add elements in the convex hull of the two intervals [RM07]. There are a number of possible fixes that include using a more expressive abstract domain such as relational domains (octagons [Min06] or polyhedra [CH78]) or expressing the intervals using disjunctive completion [CC79]. We do not go into details here but the introduction section of [RM07] describes the possible approaches and trade-offs.

An intuitive way to solve this problem, as proposed by [RM07], is to have the value of `s` be related to the control flow of the program. This results in analyzing Program 3.5 which is a re-writing of Program 3.4.

```
(ℓ0, t0): int x, y, s;
(ℓ1, t1): if (x < 0) {
(ℓ2, t1):   s = -1;
(ℓ6, t1):   y = x / s;
(ℓ3, t2): } else {
(ℓ4, t2):   s = 1;
(ℓ6, t2):   y = x / s;
(ℓ5, t0): }
```

Program 3.5: Re-writing of Program 3.4.

Now we are able to prove that a divide by zero never occurs. In general, a technique based on syntactic program rewriting may be useful as not all partitions of a program's control flow may not be expressible by the programming language under consideration [RM07]. Thus, [RM07] proposes an abstract domain construction for the partitioning of a program's control flow. This approach has two advantages: (1) it is more expressive compared to syntactic re-writing and (2) the domain is formalized in a way that makes it possible to integrate it into other static analyses

[RM07]. In this section we give a high-level overview of the trace partitioning domain and point the reader towards [RM07] for further reading.

3.5.1 Preliminary Definitions

For this thesis we only need to cover the case for partitioning when there are no function calls. Let $\mathbb{X}$ be the set of values and $\mathbb{V}$ the finite set of variables. A store is a mapping from variables to values which we denote by $\rho \in \mathbb{M}$ where $\mathbb{M} = \mathbb{V} \to \mathbb{X}$. A control state (i.e. program point) is given by a label $\ell \in \mathbb{L}$ and can be thought of as being similar to a program counter that keeps track of program points during an execution of a program. We define the set of states $\mathbb{S} = \mathbb{L} \times \mathbb{M}$ where one such state may be written as $s = (\ell, \rho)$. Next, we define a transition system by a set of initial states $\mathbb{S}^i$ and a transition relation $(\to) \subseteq \mathbb{S} \times \mathbb{S}$ which describes how a computation proceeds from one state to another. Typically, the starting state is given by $\mathbb{S}^i = \{\ell^i\} \times \mathbb{M}$ where ℓ^i is the first point in the program. We will use the words transition system and program interchangeably. Finally, a trace σ is a finite sequence $\sigma_0\sigma_1 \ldots \sigma_n \in \mathbb{S}$ and we denote the set of traces as $\mathbb{S}^*$.

3.5.2 Partitioning and Coverings

We introduce the notions of partitioning and extended transition systems.

Definition 3.5.1 ([RM07])**.** Let I, F be two sets and $\delta : I \to \wp(F)$. Then, δ is a covering of F if and only if $\forall x \in I.\, \delta(x) \neq \emptyset$ and $F = \bigcup_{x \in I} \delta(x)$. Also, δ is a partitioning of F if and only if it is a covering of F and $\forall x, y \in I.\, x \neq y \implies \delta(x) \cap \delta(y) = \emptyset$ [RM07].

We note that it is possible to define a Galois connection between the poset

$(\wp(F), \subseteq)$ and the poset $(I \to \wp(F), \subseteq)$ [RM07].

Now, assume a program P is given as a transition system defined as the tuple $(\mathbb{S}^i, \to)$. An extended transition system can be defined as a transition system where each label is also attached some token from a set of tokens $T \subseteq \mathbb{T}$. The purpose of the tokens is to capture information regarding the history of execution and associate them with program points. Furthermore, the tokens can guide partitioning [RM07].

Definition 3.5.2 ([RM07])**.** Let $T \in \mathbb{T}$. The set of extended control states is given by $\mathbb{L}_T = \mathbb{L} \times T$ and $\mathbb{S}_T = \mathbb{L}_T \times \mathbb{M}$ is the set of extended states. An extended transition system is defined by the tuple $(T, \mathbb{S}^i_T, \to_T)$ where $\mathbb{S}^i_T \subseteq \mathbb{S}$ is the set of extended initial states and $\to_T$ is the transition relation for extended states.

The notions of covering, partitioning and complete covering/partitioning are formally defined in [RM07]. Here we describe these notions informally. Let P_0 with tokens T_0 and P_1 and tokens T_1 be two extended transition systems and define $\tau : T_0 \to T_1$ which we refer to as the forget function, then for all τ:

- P_0 is a τ-covering of P_1 if and only if it simulates all of the transitions of P_1,
- P_0 is a τ-partitioning of P_1 if and only if any transition in P_1 is simulated by exactly one transition in P_0, and
- P_0 is a complete τ-covering/partitioning of P_1 if and only if P_0 is a covering/partitioning of P_1 and P_0 does not add any fictitious transition when compared to P_1 [RM07].

Intuitively, a complete covering/partitioning describes the same set of traces as the original system with the only extra information being added is tokens.

Referring back to our example programs, Program 3.4 is the original system P where we have labelled each line with labels ℓ_i. Then, Program 3.5 is the extended system P' which is a partitioning of P as each transition, namely from the branch exits to ℓ_6 is simulated by exactly one transition: ℓ_2 to ℓ_6 and ℓ_4 to ℓ_6. The tokens t_1 and t_2 indicate which of the branches we took so that we can distinguish the control flow of the program. We also note that this partitioning is complete as any execution run of P' corresponds to some execution of P. We could have also defined P using the trivial extension of a transition system.

Definition 3.5.3 ([RM07])**.** Let $T_\epsilon \in \mathbb{T}$ where $T_\epsilon = \{t_\epsilon\}$. The trivial extension of a transition system $P = (\mathbb{S}^i, \rightarrow)$ is $P_\epsilon = (\mathbb{L}_\epsilon, \mathbb{S}^i_\epsilon, \rightarrow_\epsilon)$ where $\mathbb{L}_\epsilon = \mathbb{L} \times T_\epsilon$, $\mathbb{S}^i_\epsilon = \{((\ell, t_e), \rho) \mid (\ell, \rho) \in \mathbb{S}^i\}$ and $((l_i, t_\epsilon), \rho_i) \rightarrow_\epsilon ((l_j, t_\epsilon), \rho_j) \iff (l_i, \rho_i) \rightarrow (l_j, \rho_j)$.

3.5.3 The Trace Partitioning Domain

We can define an ordering between extended transition systems with respect to some forget function τ. The relations “is a covering of,” “is a partition of” and “is complete with respect to” all form a preorder $\preceq$ on transition systems induced by partitioning. Such an ordering can be used to define valid computational orderings [CC92] [RM07].

Let $P = (\mathbb{L}, \mathbb{S}^i, \rightarrow)$ be a transition system. Then, P_ϵ is the trivial extension of P as described in Definition 3.5.3. Given the set $\mathfrak{P}$ which is the set of all complete coverings of P, we say that P_ϵ is the basis of $\mathfrak{P}$. The ordering amongst any two extended systems is given by

$$P_{T_0} \preceq P_{T_1} \iff \exists \tau : T_1 \rightarrow T_0, P_{T_1} \text{ is a } \tau\text{-covering of } P_{T_0}$$

and P_ϵ is the least element of $\mathfrak{P}$ ordered by $\preceq$. Note that it is also possible to define other ordering such as “is a τ-partitioning” of P_{T_0}. To define a trace partitioning domain we need elements of the domain to specify a covering P_T of the original transition system and a function mapping each extended control state of the covering to some abstract domain. Then, the (store abstracted) trace partitioning domain is defined by an abstract poset $(D^\sharp_{\mathbb{M}}, \sqsubseteq)$ which abtracts sets of traces to abstract invariants.

Definition 3.5.4 ([RM07])**.** An element of the partitioning abstract domain is a tuple $(P_T, \Phi^\sharp)$ where $T \in \mathbb{T}$, $P_T = (T, \mathbb{S}^i_T, \rightarrow_T)$ is a complete covering and $\Phi^\sharp$ is a function $\Phi^\sharp : \mathbb{L}_T \rightarrow D^\sharp_{\mathbb{M}}$. The domain of such tuples is denoted as $\mathbb{D}^\sharp$.

Now, an abstract value is a value in $\mathbb{L}_T \rightarrow D^\sharp_{\mathbb{M}} = (\mathbb{L} \times T) \rightarrow D^\sharp_{\mathbb{M}}$ which by curryfication is isomorphic to value in $\mathbb{L} \rightarrow T \rightarrow D^\sharp_{\mathbb{M}}$. The latter representation is useful for the implementation of the partitioning abstract domain as we now have a mapping from program locations to partitioning tokens that describe an abstract invariant at each location with respect to some partitioning.

The ordering between any two abstract elements can be defined with respect to the ordering of the extended transition systems and the function $\Phi^\sharp$.

Definition 3.5.5 ([RM07])**.** Let $(P_{T_0}, \Phi^\sharp_0)$, $(P_{T_1}, \Phi^\sharp_1)$ be two transition systems and $\tau : T_1 \rightarrow T_0$. Then, we define

$$\begin{aligned} \Gamma_\tau \, : \, (\mathbb{L}_{T_1} \rightarrow D^\sharp_{\mathbb{M}}) &\rightarrow (\mathbb{L}_{T_0} \rightarrow D^\sharp_{\mathbb{M}}) \\ \Phi^\sharp_0 &\mapsto \lambda(l^{\ell'} \in \mathrm{dom}(\Phi^\sharp_1)). \bigsqcup \{\Phi^\sharp_0(l) \mid \ell \in \mathrm{dom}(\Phi^\sharp_0), \tau(\ell) = \ell'\} \end{aligned}$$

which we can then use to define the ordering between any two transition systems as

$$(P_{T_0}, \Phi_0^\sharp) \preceq^\sharp (P_{T_1}, \Phi_1^\sharp) \iff P_{T_0} \preceq_\tau P_{T_1} \wedge \Phi_0^\sharp \sqsubseteq \Gamma_\tau^\sharp(\Phi_1^\sharp).$$

We may also define a concretization function that maps elements of $\mathbb{D}^\sharp$ to elements of $\mathbb{D}$ where $\mathbb{D}$ is defined by the pair $(P_T, \Phi) = ((T, \mathbb{S}_T^i, \rightarrow_T), \mathbb{L}_T \rightarrow \wp(\mathbb{S}^*))$. That is, we may map abstract partitions to their concrete ones.

Definition 3.5.6 ([RM07])**.** Let $\gamma_\mathbb{P}^\sharp$ be the function $\gamma_\mathbb{P}^\sharp : D^\sharp \rightarrow D$ which we define as

$$(P_T, \Phi^\sharp) \rightarrow (P_T, \lambda(l \in \mathbb{L}_T).\, \gamma_\mathbb{M} \circ \Phi^\sharp(l))$$

where $\gamma_\mathbb{M} : D_\mathbb{M}^\sharp \rightarrow \wp(\mathbb{M})$ is the concretization function for the abstract domain $D_\mathbb{M}^\sharp$.

Finally, it is possible to define widening for elements of $D^\sharp$ so that we may ensure convergence of the partitioning analysis. We must choose $\nabla_\mathbb{M}$ as a widening for $D_\mathbb{M}^\sharp$ and $\nabla_\mathfrak{P}$ as a widening over the basis so as to obtain a pairwise widening $\nabla_{\mathbb{D}^\sharp}$.

Definition 3.5.7 ([RM07])**.** Let $(P_{T_0}, \Phi_0^\sharp), (P_{T_1}, \Phi_1^\sharp) \in \mathbb{D}^\sharp$, then the widening is defined as

$$(P_{T_0}, \Phi_0^\sharp) \nabla_{D^\sharp} (P_{T_1}, \Phi_1^\sharp) = (P_{T_2}, \Phi_2^\sharp)$$

where

- $P_{T_2} = P_{T_0} \nabla_\mathfrak{P} P_{T_1}$ such that $P_{T_0} \preceq_{\tau_0} P_{T_2}$ and $P_{T_1} \preceq_{\tau_1} P_{T_2}$, and
- $\Phi_2^\sharp = (\Phi_0^\sharp \circ \tau_0) \nabla_\mathbb{M} (\Phi_1^\sharp \circ \tau_1)$.

The intuition behind the widening operation is that it should first stabilize the partitioning of the transition system so as to enforce termination of the partitioning strategy and then stabilize the abstract elements $D^\sharp_{\mathbb{M}}$.

3.5.4 Remarks

We see that the trace partitioning domain formalizes the notion of trace partitioning as an abstract interpretation of an underlying transition system. Soundness of the analysis follows by the soundness of the abstract domain being used and the soundness of the partitioning strategy. As long as our created partitions are either coverings or partitions, soundness is provided free. It is also possible to relate one partitioning to another using the computational ordering $\preceq_\tau$ and relate the abstract partitions to their concrete counterparts via $\gamma^\sharp_{\mathbb{P}}$. Furthermore, by curryfication we can view partitions as mappings from program points to partitioning tokens which makes it possible to incorporate partitioning domains into the calculational designs of generic abstract interpreters [Cou99]. Finally, partitioning may occur either statically or dynamically. This makes the trace partitioning domain a powerful abstract domain that we will use for the responsibility analyses of floating-point programs.

3.6 Under-approximating Backward Semantics

Historically, the majority of abstract interpretation research has focused on designing sound forward semantics. Given some abstract pre-condition we may deduce an over-approximation of a post-condition for programs. The dual of this problem is to consider an abstract post-condition and infer an under-approximation of a pre-condition that causes the post-condition to hold true. Thus, we wish to automatically infere sufficient pre-conditions using abstract interpretation. In this section we review some definitions from [Min14] which outlines how to design backward under-approximation for numeric abstract domains such as boxes, octagons and polyhedra.

3.6.1 Summary

Classical abstract domains have abstract operators for both backward and forward analyses. When an exact result cannot be computed due to inherent limitations in the abstract domain (e.g. boxes can only describe convex sets) an over-approximation is computed. These approximate results can be used to infer necessary conditions but are not suitable for the inference of sufficient conditions. Thus, to soundly approximate sufficient condition we must compute under-approximations. Backward analysis is the process of inferring a pre-condition given a post-condition. [Min14] addresses the problem of inferring sufficient pre-conditions using backward under-approximations.

Definition 3.6.1 ([Min14])**.** Given two sets A and B, the backward function $\overleftarrow{f}$ of a function $f : \wp(A) \to \wp(B)$ is defined as

$$\begin{aligned} \overleftarrow{f} &: \wp(B) \to \wp(A) \\ \overleftarrow{f}(B) &\triangleq \{a \in A \mid f(\{a\}) \subseteq B\}. \end{aligned} \tag{3.1}$$

Definition 3.6.1 can be used to define the backward semantics of a function that computes the forward semantics. We can lift $\subseteq$, $\cup$ and $\cap$ element-wise to functions $A \to \wp(B)$. We say that $f \subseteq g \iff \forall a \in A. f(a) \subseteq g(a)$, $f \cup g \triangleq \lambda a. f(a) \cup g(a)$

and $f \cap g \triangleq \lambda a.\, f(a) \cap g(a)$. We denote function compositions with $\circ$.

Proposition 3.6.1 ([Min14])**.** The following properties are true.

1. $\overleftarrow{f}$ is monotonic and a $\cap$-morphism.
2. If f is monotonic, then $\overleftarrow{f} \circ f$ is extensive, i.e. $A \subseteq (\overleftarrow{f} \circ f)(A)$.
3. If f is a $\cup$-morphism then $f \circ \overleftarrow{f}$ is reductive, i.e. $(f \circ \overleftarrow{f})(B) \subseteq B$.
4. If f is extensive then $\overleftarrow{f}$ is reductive. If f is reductive then $\overleftarrow{f}$ is extensive.
5. If f is a $\cup$-morphism then $\wp(X) \underset{f}{\overset{\overleftarrow{f}}{\leftrightarrows}} \wp(Y)$ is a Galois connection.

By Property 1 from Proposition 3.6.1 we know that fixpoints exist for backward functions. Furthermore, using the properties from Proposition 3.6.1 we may prove certain properties about inverting forward functions.

Proposition 3.6.2 ([Min14])**.** The following properties are true.

1. $\overleftarrow{\lambda A.\, A} = \lambda B.\, B$.
2. $\overleftarrow{f \cup g} = \overleftarrow{f} \cap \overleftarrow{g}$.
3. $\overleftarrow{f \cap g} \supseteq \overleftarrow{f} \cup \overleftarrow{g}$.
4. If f is a $\cup$-morphism then $\overleftarrow{f \circ g} = \overleftarrow{g} \circ \overleftarrow{f}$
5. $f \subseteq g \implies \overleftarrow{g} \subseteq \overleftarrow{f}$. If f is a $\cup$-morphism then $f \subseteq g \iff \overleftarrow{g} \subseteq \overleftarrow{f}$ and so $f = g \iff \overleftarrow{f} = \overleftarrow{g}$.
6. If f and g are monotonic and $f \circ g = g \circ f = \lambda x.\, x$ then $\overleftarrow{f} = g$ and $\overleftarrow{g} = f$.
7. If f is an extensive $\cup$-morphism then $\overleftarrow{\lambda x.\, \mathrm{lfp}_x f} = \lambda y.\, \mathrm{gfp}_y \overleftarrow{f}$ where gfp is the greatest fix-point.

8. If f is a $\cup$-morphism then $\overleftarrow{\lambda x.\, \mathrm{lfp}_x(\lambda z.\, z \cup f(z))} = \lambda y.\, \mathrm{gfp}_y(\lambda z.\, z \cap \overleftarrow{f}(z))$.

Now, using the properties from Proposition 3.6.2 we may define abstract semantics that compute backward under-approximations for programs. Given a concrete domain $\mathbb{D}$ and an abstract domain $\mathbb{D}^\sharp$ that is connected by a Galois connection (α, γ) we may define the abstract forward necessary conditions.

Definition 3.6.2 ([Min14]). The soundness condition of an abstract over-approximating forward function $F^\sharp : \mathbb{D}^\sharp \to \mathbb{D}^\sharp$ with respect to the concrete forward function $F : \mathbb{D} \to \mathbb{D}$ is given as

$$\forall X^\sharp \in \mathbb{D}^\sharp.\, (F \circ \gamma)(X^\sharp) \subseteq (\gamma \circ F^\sharp)(X^\sharp).$$

Using Proposition 3.6.2 and Definition 3.6.2 we then define the abstract sufficient conditions for backward under-approximations.

Definition 3.6.3 ([Min14]). The soundness condition of an abstract under-approximating backward function $F^\sharp : \mathbb{D}^\sharp \to \mathbb{D}^\sharp$ with respect to the concrete backward function $F : \mathbb{D} \to \mathbb{D}$ is given as

$$\forall X^\sharp \in \mathbb{D}^\sharp.\, (\gamma \circ F^\sharp)(X^\sharp) \subseteq (F \circ \gamma)(X^\sharp).$$

Furthermore, using the definition of forward widenings given in Definition 3.2.2 we define a lower widening that obeys the soundness condition in Definition 3.6.3 and ensures the termination of greatest fix-point computations.

Definition 3.6.4 ([Min14]). Let $(\mathbb{D}^\sharp, \sqsubseteq, \bot, \top, \sqcup, \sqcap)$ be a complete lattice. Then, the function $\underline{\nabla} : \mathbb{D}^\sharp \times \mathbb{D}^\sharp \to \mathbb{D}^\sharp$ is a lower widening operator if for all $x, y \in \mathbb{D}^\sharp$

$x\underline{\nabla}y \sqsubseteq x$ and $x\underline{\nabla}y \sqsubseteq y$ and for all decreasing chains $x_0 \sqsubseteq x_1 \sqsubseteq \dots$ the decreasing chain $y_0 = x_0, \dots, y_i = y_{i-1}\underline{\nabla}x_i$ for all $i > 0$ is not strictly decreasing.

Definition 3.6.5 ([Min14]). Given $F^\sharp$ and $X^\sharp$ along with the corresponding concrete F and X, the sequence $Y_0^\sharp = X^\sharp$, $Y_{i+1}^\sharp = Y_i^\sharp\underline{\nabla}F^\sharp(Y_i^\sharp)$ is utimately stationary which we denote by $\lim_{X^\sharp} \lambda Y^\sharp.\, Y^\sharp\underline{\nabla}F^\sharp(Y^\sharp)$. This limit is a sound under-approximation of the greatest fixpoint of the concrete function F:

$$\gamma(\lim_{X^\sharp} \lambda Y^\sharp.\, Y^\sharp\underline{\nabla}F^\sharp(Y^\sharp)) \subseteq \mathrm{gfp}_X F.$$

Thus, we see that there is a duality between the forward over-approximating necessary semantics and the backward under-approximating sufficient semantics. This duality applies to defining backward operators from forward operators and to the calculational design of a backward abstract interpreter which can be expressed as the dual of a forward abstract interpreter. An example backward abstract interpreter is given in [Min14] which is similar to the one given in [Cou19]. We will be using this construction for the design of our backward abstract semantics which means we only need to define backward semantics for assignments and Boolean expressions. Some key results from [Min14] to help define backward semantics is given in the next section.

3.6.2 Review of Key Results

Definition 3.6.6 ([Min14]). Given a function $\tau[\![\mathtt{B}]\!]$ that computes the forward semantics of Boolean tests with respect to the abstract environment $\mathcal{E}$, we know that $\tau[\![\mathtt{B}]\!] \subseteq \lambda x.\, x$. By Proposition 3.6.1 we get that the backward function $\overleftarrow{\tau}[\![\mathtt{B}]\!] \supseteq \overleftarrow{\lambda x.\, x} = \lambda x.\, x$. Thus, the abstract fallback operator for Boolean tests is defined as

$$\overleftarrow{\tau}^{\sharp}[\![\texttt{B}]\!] \triangleq \lambda x.\, x.$$

The fallback operator given Definition 3.6.6 provides too coarse of an under-approximation but is useful for when no other backward Boolean semantics may be defined. In the case of boxes, octagons and polyhedra, [Min14] also provides backward under-approximating Boolean tests that evaluate different types of Boolean expressions. The general idea is to remove the test constraint from the current abstract environment which has the effect of adding all points that satisfy the negation of the test. Note that it is possible to define other types of backward Booleans tests that compute smaller invariants since we are computing under-approximations. Next, we consider backward projections and assignment statements.

Definition 3.6.7 ([Min14])**.** Let $\overleftarrow{\tau}[\![\texttt{V := [}-\infty\texttt{,}\infty\texttt{]}]\!]$ be a backward projection operator. For any convex closed domain $\overleftarrow{\tau}[\![\texttt{V := [}-\infty\texttt{,}\infty\texttt{]}]\!]$ can be efficiently and exactly implemented using the forward projection operator $\tau[\![\texttt{V := [}-\infty\texttt{,}\infty\texttt{]}]\!]$.

$$\overleftarrow{\tau}[\![\texttt{V := [}-\infty\texttt{,}\infty\texttt{]}]\!]X^{\sharp} \triangleq \begin{cases} X^{\sharp} & \text{if } \gamma(\tau[\![\texttt{V := [}-\infty\texttt{,}\infty\texttt{]}]\!]X^{\sharp}) = \gamma(X^{\sharp}), \\ \bot^{\sharp} & \text{otherwise.} \end{cases}$$

Proposition 3.6.3 ([Min14])**.** The projection operator for the forward semantics over-approximates the semantics of assignments: $\tau[\![\texttt{V := e}]\!] \subseteq \tau[\![\texttt{V := [}-\infty\texttt{,}\infty\texttt{]}]\!]$. By Proposition 3.6.2 we have that the backward projection under-approximates the backwards semantics of assignments: $\overleftarrow{\tau}[\![\texttt{V := [}-\infty\texttt{,}\infty\texttt{]}]\!] \subseteq \overleftarrow{\tau}[\![\texttt{V := e}]\!]$.

Definition 3.6.8 ([Min14])**.** Given the forward functions $\tau[\![\texttt{add V}]\!]$ and $\tau[\![\texttt{del V}]\!]$ that, respectively, add and remove variables from abstract environments, we can compute the semantics of backward assignments as follows

$$\overleftarrow{\tau}[\![\texttt{V := e}]\!] \triangleq \tau[\![\texttt{del V'}]\!] \circ \overleftarrow{\tau}[\![\texttt{V := [}-\infty\texttt{,}\infty\texttt{]}]\!] \circ \overleftarrow{\tau}[\![\texttt{V' = e}]\!] \circ \tau[\![\texttt{add V}]\!] \circ [\texttt{V'} \setminus \texttt{V}]$$

where $[\texttt{V'} \setminus \texttt{V}]$ renames all variables `V` in the abstract environment $X^\sharp$ by `V'` (which is always capture-free).

Definition 3.6.9 ([Min14])**.** If $X^\sharp$ is a closed convex set then the backward semantics of $\overleftarrow{\tau}[\![\texttt{v := [a,b]}]\!]$ is given by

$$\begin{aligned}\overleftarrow{\tau}[\![\texttt{v := [a,b]}]\!]X^\sharp \triangleq & (\tau[\![\texttt{V := [}-\infty\texttt{,}\infty\texttt{]}]\!] \circ \\ & (\tau[\![\texttt{V := a}]\!] \sqcap \tau[\![\texttt{V := V - b}]\!]) \circ \\ & \tau[\![\texttt{V >= a \&\& V <= b}]\!])X^\sharp.\end{aligned}$$

In Definition 3.6.8 the operation $\overleftarrow{\tau}[\![\texttt{V' = e}]\!]$ can be defined by reduction to the sound under-approximation of Boolean expressions. Furthermore, [Min14] also presents a few other backward assignment operators that can be used for particular cases such as assignments that assign a value to a variable in some given range (Definition 3.6.9) or when $\overleftarrow{\tau}[\![\texttt{V' = e}]\!]$ can be exactly modeled in the abstract domain. For numeric domains, backwards abstract assignments lead to an over-approximation so they cannot exactly model $\overleftarrow{\tau}[\![\texttt{V' = e}]\!]$.

Finally, we note that a lower widening may always jump to $\bot$ after some number of iterations. Since the lower widening is supposed to under-approximate greatest fixpoints of a decreasing function, we may jump below the actual fixpoint to $\bot$ and this will be sound. In [Min14] lower widening for polyhedra and intervals are given with the latter being based on widening thresholds [Ber+15] which then jump to $\bot$

if a fixpoint is not reached after widening to the threshold limits.

Chapter 4

Floating-Point Responsibility Analysis

The concrete semantics for responsibility as described in the Section 3.4 is not computable for all programs in general. Thus, we must soundly approximate the responsibility semantics and compute an approximate set of entities that may be responsible for bad behaviors. We wish to compute what we will refer to as a set of left bounds and right bounds of responsibility. A left bound and its corresponding right bound are program locations where if there is a responsible entity for some trace then it must lie within that bound. This means that for any single trace the left and right bound should soundly approximate the responsible entity for that trace. Since we will be computing an approximation, in our case the left and right bounds will include the responsible entity for a set of traces which share a common responsible entity (recall that there is only one responsible entity per trace). Now, the responsible entity may change depending on the trace so we must compute a set of left and right bounds where each of them contains a responsible entity. Let $[i, j]$ denote the interval of responsibility where $i \leq j$ means that the responsibility program locations are between points i and j. $i > j$ means there is no responsible entity. Finally, we note that if the left bound is chosen to be the start of the program and the right bound of the program is chosen to be the end of the program then this is the trivial sound interval for responsibility for all traces.

This chapter begins by outlining the syntax of programs. Then, we discuss the forward followed by the backward semantics of floating-point programs and how they are used to partition the sets of traces that are being approximated. This partitioning may be used to possibly refine the left and right bounds of responsibility to obtain more accurate bounds. Throughout the explanation Program 2.1 is used as a running example.

4.1 Syntax

We present a simple C-like language to be used for the formal construction of the analysis. The syntax reflects some of the restrictions we pose on the analysis. Firstly, there are no function calls and instead the program starts executing from the first statement it encounters. Secondly, the numeric values in our programs are real numbers and we do not allow for integer values. This restriction allows us to avoid the issue of casting between integers and floating-point types which can lead to errors such as integer overflow or underflow and some other bugs if not treated properly [Dan02]. Thirdly, we do not have floating-point division as part of our arithmetic operations. New program features could be added by first extending the syntax and then defining a sound semantics for the expression and/or statement just added to the program. We will show a sound semantics for this syntax so this extension would only focus on the features added to the language. For example, floating-point division can be added by using the semantics of affine division given in [GP11].

The set $\mathbb{V}$ defines the infinite set of program variables that is disjoint from the other syntactic structures of a program. We do not explicitly support the Boolean constants $\top$ and $\perp$ but these are easily obtainable (for example doing the Boolean test `x == x` which is trivially true). Program variables represent two different values. They can be thought of as a tuple where the first component is the "errorless" real value of the variable and the second component is the floating-point counterpart of this value. Thus, the first element keeps track of what the computation would ideally compute, i.e. without any floating-point error, while its corresponding floating-point value is what is actually computed. The error assertion is a limited form of a C assertion where the only condition being checked is whether the absolute error between a program variable **var**'s real and floating-point components is smaller than **num**. The programmer may check that a program does not accrue too much error using this construct.

Another interesting construct of the language is the assignment statement. The assignment with uncertainty allows the programmer to associate with a real interval some error, typically in the range $0 \leq u << 1$. This error can correspond to either an error due to floating-point representation (as not all real values are representable in floating-point) or some other uncertainty such as measurement error. If there is no uncertainty in the assignment then this becomes a special case of the uncertain assignment where $u = 0$. We syntactically separate the case where an assignment with uncertainty occurs and where a variable is assigned the result of some arithmetic expression as intervals are not a part of the syntax for arithmetic expressions.

num $\in \mathbb{R}$		*real numbers*
var $\in \mathbb{V}$		*variables*
`A` $\in \mathbb{A}$ `::=`	**num** `|` **var**	*arithmtic expressions*
	`| A` op^A `A`	where $op^A \in \{$`+`,`-`,`*`$\}$
`B` $\in \mathbb{B}$ `::=`	`not B | B or B | B and B`	*boolean expressions*
	`| A` op^B `A`	where $op^B \in \{$`<`,`<=`,`>`,`>=`,`=`,`<>`$\}$
`S` $\in \mathbb{S}$ `::=`		*program statement*
	var `= [`**num**`,`**num**`] + u ;`	*assignment with uncertainty*
	`|` **var** `= A;`	*arithmetic assignment*
	`| if (B) S else S`	*conditional*
	`| while (B) S`	*iteration*
	`| assert(`**var**`,`**num**`)`	*error assertion*
	`| ;`	*skip statement*
	`| { Sl }`	*multiple statements*
`Sl` $\in \mathbb{Sl}$ `::=`	`Sl S |` ϵ	*statement list*
`P` $\in \mathbb{P}$ `::=`	`Sl`	*programs*

Figure 4.1: Syntax of programs to be analyzed.

$\mathit{at}\llbracket \mathbb{S} \rrbracket$	$: \mathbb{Pc} \rightarrow \mathbb{L}$	label for program point where execution of S starts
$\mathit{after}\llbracket \mathbb{S} \rrbracket$	$: \mathbb{Pc} \rightarrow \mathbb{L}$	label for program point after the execution of S has completed (assuming the instruction at S terminates)
$\mathit{in}\llbracket \mathbb{S} \rrbracket$	$: \mathbb{Pc} \rightarrow \mathcal{P}(\mathbb{L})$	labels for all the potentially reachable program points inside S if S is executed (includes $\mathit{at}\llbracket S \rrbracket$, excludes $\mathit{after}\llbracket \mathbb{S} \rrbracket$)
$\mathit{labs}\llbracket \mathbb{S} \rrbracket$	$: \mathbb{Pc} \rightarrow \mathcal{P}(\mathbb{L})$	$\mathit{labs}\llbracket \mathbb{S} \rrbracket \triangleq \mathit{in}\llbracket \mathbb{S} \rrbracket \cup \{\mathit{after}\llbracket \mathbb{S} \rrbracket\}$

Figure 4.2: Description of label functions for programs.

4.1.1 Labels

We use the explicit labelling scheme from [Cou19] and [CC98] to attach unique labels to each program point. This in turn allows the construction of the semantics of programs where an abstract invariant is attached to each program point. The functions defined below are same or similar to those given in [Cou19] and [CC98].

Let $\mathbb{L}$ be the set of labels and $\mathbb{Pc} = \mathbb{S} \cup \mathbb{Sl} \cup \mathbb{P}$ be the set of all program components. Then, we define the functions given in Figure 4.2.

Next, we decorate the syntax in Figure 4.1 with labels to obtain an explicit labelling [CC98]. The definition of $\mathit{at}\llbracket \mathbb{S} \rrbracket$ is given by Figure 4.3 and $\mathit{after}\llbracket \mathbb{S} \rrbracket$ by Figure 4.4. Using $\mathit{at}\llbracket \mathbb{S} \rrbracket$ we can obtain $\mathit{in}\llbracket \mathbb{S} \rrbracket$ where we note that while Figure 4.5 does not enforce the uniqueness of labels, it can easily be made to do so by adding the following conditions:

S	::=	$^{\ell}$ **var** = [**num**,**num**] + u ;	$\mathit{at}[\![\texttt{S}]\!] \triangleq \ell$
S	::=	$^{\ell}$ **var** = A ;	$\mathit{at}[\![\texttt{S}]\!] \triangleq \ell$
S	::=	if $^{\ell}$ (B) St else Sf	$\mathit{at}[\![\texttt{S}]\!] \triangleq \ell$
S	::=	while $^{\ell}$ (B) Sw	$\mathit{at}[\![\texttt{S}]\!] \triangleq \ell$
S	::=	$^{\ell}$ assert(**var**,**num**)	$\mathit{at}[\![\texttt{S}]\!] \triangleq \ell$
S	::=	$^{\ell}$;	$\mathit{at}[\![\texttt{S}]\!] \triangleq \ell$
S	::=	$^{\ell}$ { Sl }	$\mathit{at}[\![\texttt{S}]\!] \triangleq \mathit{at}[\![\texttt{Sl}]\!]$
Sl	::=	Sl' S	$\mathit{at}[\![\texttt{Sl}]\!] \triangleq \mathit{at}[\![\texttt{Sl'}]\!]$
Sl	::=	ϵ	$\mathit{at}[\![\texttt{Sl}]\!] \triangleq \mathit{after}[\![\texttt{Sl}]\!]$
P	::=	Sl $^{\ell}$	$\mathit{at}[\![\texttt{P}]\!] \triangleq \mathit{at}[\![\texttt{Sl}]\!]$

Figure 4.3: Computing $\mathit{at}[\![\texttt{S}]\!]$ structurally on the syntax.

S	::=	if $^{\ell}$ (B) St else Sf	$\mathit{after}[\![\texttt{St}]\!] \triangleq \mathit{after}[\![\texttt{Sf}]\!] \triangleq \mathit{after}[\![\texttt{S}]\!]$
S	::=	while $^{\ell}$ (B) Sw	$\mathit{after}[\![\texttt{Sw}]\!] \triangleq \ell$
S	::=	$^{\ell}$ { Sl }	$\mathit{after}[\![\texttt{Sl}]\!] \triangleq \mathit{at}[\![\texttt{S}]\!]$
Sl	::=	Sl' S	$\mathit{after}[\![\texttt{Sl'}]\!] \triangleq \mathit{at}[\![\texttt{Sl}]\!]$, $\mathit{after}[\![\texttt{S}]\!] \triangleq \mathit{after}[\![\texttt{Sl}]\!]$
P	::=	Sl $^{\ell}$	$\mathit{after}[\![\texttt{P}]\!] \triangleq \mathit{after}[\![\texttt{Sl}]\!] \triangleq \ell$

Figure 4.4: Computing $\mathit{after}[\![\texttt{S}]\!]$ structurally on the syntax.

- $\mathit{at}[\![\texttt{S}]\!] \notin \mathit{in}[\![\texttt{St}]\!] \cup \mathit{in}[\![\texttt{Sf}]\!]$, $\mathit{in}[\![\texttt{St}]\!] \cap \mathit{in}[\![\texttt{Sf}]\!] = \emptyset$ for the conditional statement;
- $\mathit{at}[\![\texttt{S}]\!] \notin \mathit{in}[\![\texttt{Sw}]\!]$ for the while statement; and
- $\mathit{after}[\![\texttt{Sl}]\!] \notin \mathit{in}[\![\texttt{Sl}]\!]$ for the top-level.

The labelling functions will be used to give the semantics of programs in the following sections. Also, note that the labelling given here corresponds to the labelling of nodes in the transition system Figure 4.7 where we pick each ℓ from the lines of the program. To obtain the more refined transition systems from Section 2,

`S ::= var = [num,num] + u ;`	$in[\![\texttt{S}]\!] \triangleq \{at[\![\texttt{S}]\!]\}$
`S ::= var = A ;`	$in[\![\texttt{S}]\!] \triangleq \{at[\![\texttt{S}]\!]\}$
`S ::= if (B) St else Sf`	$in[\![\texttt{S}]\!] \triangleq \{at[\![\texttt{S}]\!]\} \cup in[\![\texttt{St}]\!] \cup in[\![\texttt{Sf}]\!]$
`S ::= while` $^{\ell}$ `(B) Sw`	$in[\![\texttt{S}]\!] \triangleq \{at[\![\texttt{S}]\!]\} \cup in[\![\texttt{Sw}]\!]$
`S ::= assert(var,num)`	$in[\![\texttt{S}]\!] \triangleq \{at[\![\texttt{S}]\!]\}$
`S ::= ;`	$in[\![\texttt{S}]\!] \triangleq \{at[\![\texttt{S}]\!]\}$
`S ::= { Sl }`	$in[\![\texttt{S}]\!] \triangleq \{at[\![\texttt{Sl}]\!]\}$
`Sl ::= Sl' S`	$in[\![\texttt{Sl}]\!] \triangleq in[\![\texttt{Sl'}]\!] \cup in[\![\texttt{Sl}]\!]$
`Sl ::=` ϵ	$in[\![\texttt{Sl}]\!] \triangleq \{at[\![\texttt{Sl}]\!]\}$
`P ::= Sl` $^{\ell}$	$in[\![\texttt{P}]\!] \triangleq in[\![\texttt{Sl}]\!]$

Figure 4.5: Computing $in[\![\texttt{S}]\!]$ structurally on the syntax.

we will have to incorporate more information than these labels into our program semantics.

4.2 Lattice of System Behaviors

We formally define the system behaviors of interest [DC19] for floating-point programs using event traces. The semantics of programs interpreted using constrained affine sets are with respect to both real and floating-point numbers. So, for the simple language given in Section 4.1, we need to define so-called real events and their floating-point counterparts. Let E_R be the set of real events and E_F be the set of floating-point events. In our case, we have the following real events $e_r \in E_R$: assignment (both with uncertainty and arithmetic), Boolean test, error assertion and skip. Since we also need to obtain a floating-point event for any given real event we also introduce the event rounding operator $\rho : E_R \to E_F$ that takes any event $e_r \in E$ and returns a floating-point version of that event $\rho(e_r) \in E_F$. As an example, if we have the assignment `x = [1,3] + ux` in the source code then the corresponding real event and floating-point events are `x = [1,3] + ux` and $\rho($`x = [1,3] + ux`$)$ respectively. Note that these events have an implicit semantics where the real event corresponds to assigning `x` the interval `[1,3]` while the floating-point event also adds the error term `ux` to this event. For arithmetic and Boolean operations ρ will round all sub-expressions to their floating-point values. The rounding for skip and error assertion will not do anything and return the original event. The semantics of error assertions for real events will always evaluate to true as real numbers have no round-off error while for floating-point events we will measure the absolute error of the variable in the assertion. Thus, the concrete semantics of programs will be all possible combinations of its real and floating-point events and this can be represented as a tree such as in Figure 3.1.

Let $E = E_R \cup E_F$ be the set of all events. Then, a trace $\sigma \in E^{*\infty}$ is either a finite or infinite sequence of events with length $|\sigma|$ that represents a single execution of the system. Note that ϵ is the empty trace with length 0 and the length of any infinite trace is denoted by ∞. We may also define a prefix ordering of traces. A trace σ is less than or equal to another trace σ', denoted by $\sigma \preceq \sigma'$, if and only if σ is a prefix of σ'. An example prefix trace is given in Section 3.4.2. In the case of responsibility analysis, we are concerned with the set of maximal traces of the program which we denote as $\mathcal{S}^{Max} \in \wp(E^{*\infty})$ [DC19]. The maximal traces of an empty program is $\{\epsilon\}$. Given a set of traces T, the function $\text{Pref}(P) \in \wp(E^{*\infty}) \rightarrow \wp(E^{*\infty})$ returns all prefixes of every trace in T:

$$\text{Pref}(T) \triangleq \{\sigma' \in E^{*\infty} \mid \exists \sigma \in T.\, \sigma' \preceq \sigma \wedge |\sigma'| \leq |\sigma|\}.$$

Almost all behaviors of a given system can be represented as sets of maximal traces [DC19]. We refer to such sets as trace properties, denoted $\mathcal{P} \in \wp(\mathcal{S}^{Max})$, which describe a set of traces where a given property holds. For the responsibility analysis of floating-point programs the set of bad behaviors $\mathfrak{B}$ is defined as

$$\begin{aligned}\mathfrak{B} = \{\sigma \in \mathcal{S}^{Max} \mid \exists i < |\sigma| - 1.\ \ & \sigma_i = \texttt{B} \wedge \neg\rho(\texttt{B}) \\ & \vee \sigma_i = \rho(\texttt{B}) \wedge \neg\texttt{B} \\ & \vee \sigma_i = \neg\texttt{B} \wedge \rho(\texttt{B}) \\ & \vee \sigma_i = \neg\rho(\texttt{B}) \wedge \texttt{B} \\ & \vee \sigma_i = \neg(\texttt{assert(var,num)}) \\ & \vee \sigma_i = \neg\rho(\texttt{assert(var,num)})\}.\end{aligned}$$

$\mathfrak{B}$ is the set of all maximal traces where there exists at least one event σ_i such that there is a conditional divergence or the absolute error of a variable `var` exceeds some bound `num`. The set of OK behaviors, i.e. those are not bad, is given by $\neg\mathfrak{B} = \mathcal{S}^{Max} \setminus \mathfrak{B}$. We can build a complete lattice of maximal trace properties for floating-point properties of interest $(\mathfrak{L}^{Max}, \subseteq, \top^{Max}, \bot^{Max}, \cup, \cap)$ where $\mathfrak{L}^{Max} = \{\mathfrak{B}, \neg\mathfrak{B}, \mathcal{S}^{Max}, \emptyset\} \in \wp(\wp(E^{*\infty}))$ is the set of behaviors of interest and $\top^{Max} = \mathcal{S}^{Max}$, $\bot^{Max} = \emptyset$ and $\subseteq, \cup, \cap$ are standard set operations. The Hasse diagram of this lattice is given in Figure 3.2 but now $\mathfrak{B}$ also covers the case of the program accruing too much error due to rounding. Note that we could also refine our lattice to specific cases similar to Figure 3.3 to discern between bad responsible entities for different bad behaviors. In the case for this thesis we will be using the 4-element lattice construction as described above. Finally, this definition defines the case of detecting too much error as an annotation that the programmer writes to the program. If the programmer never inserts any error assertions then regardless of how high the error gets it will never be included in the set of bad behaviors $\mathfrak{B}$.

Despite its expresivness, the maximal trace property does not reveal the point along a maximal trace where exactly in the trace is a property guaranteed to hold. So, it is difficult to characterize who can be considered responsible in any given maximal trace. Because of this we abstract every maximal trace property $\mathcal{P} \in \mathfrak{L}^{Max}$ to prefixes of maximal traces. These prefixes can be grouped as the set of all prefixes where $\mathcal{P}$ will hold at some point in the computation and exclude those whose maximal extension will not satisfy $\mathcal{P}$. As presented in [DC19], this abstraction is called prediction abstraction.

$$\begin{aligned}
\alpha_{\text{Pred}}[\![\mathcal{S}^{Max}]\!] &\in \wp(E^{*\infty}) \to \wp(E^{*\infty}) \qquad \text{prediction abstraction} \\
\alpha_{\text{Pred}}[\![\mathcal{S}^{Max}]\!](\mathcal{P}) &\triangleq \{\sigma \in \text{Pref}(\mathcal{P}) \mid \forall \sigma' \in \mathcal{S}^{Max}.\, \sigma \preceq \sigma' \implies \sigma' \in \mathcal{P}\} \\
\gamma_{\text{Pred}}[\![\mathcal{S}^{Max}]\!] &\in \wp(E^{*\infty}) \to \wp(E^{*\infty}) \qquad \text{prediction concretization} \\
\alpha_{\text{Pred}}[\![\mathcal{S}^{Max}]\!](\mathcal{Q}) &\triangleq \{\sigma \in \mathcal{Q} \mid \sigma \in \mathcal{S}^{Max}\} \cap \mathcal{S}^{Max} = \mathcal{Q} \cap \mathcal{S}^{Max}
\end{aligned}$$

There is a Galois bijection between maximal trace properties and prediction trace properties:

$$(\wp(\mathcal{S}^{Max}), \subseteq) \underset{\alpha_{\text{Pred}}[\![\mathcal{S}^{Max}]\!]}{\overset{\gamma_{\text{Pred}}[\![\mathcal{S}^{Max}]\!]}{\leftleftarrows\!\!\rightrightarrows}} (\bar{\alpha}_{\text{Pred}}[\![\mathcal{S}^{Max}]\!](\wp(\mathcal{S}^{Max})), \subseteq)$$

where

$$\begin{aligned}
\bar{\alpha}_{\text{Pred}}[\![\mathcal{S}^{Max}]\!] &\in \wp(\wp(E^{*\infty})) \to \wp(\wp(E^{*\infty})) \\
\bar{\alpha}_{\text{Pred}}[\![\mathcal{S}^{Max}]\!](\mathcal{X}) &\triangleq \{\alpha_{\text{Pred}}[\![\mathcal{S}^{Max}]\!](\mathcal{P}) \mid \mathcal{P} \in \mathcal{X}\}
\end{aligned}$$

is the prediction trace domain. Now every behavior in the floating-point lattice $\mathfrak{L}^{Max}$ can be abstracted as follows:

- $\alpha_{\text{Pred}}[\![\mathcal{S}^{Max}]\!](\top^{Max}) = \text{Pref}(\mathcal{S}^{Max})$, meaning every valid prefix trace guarantees $\top^{Max}$.
- $\alpha_{\text{Pred}}[\![\mathcal{S}^{Max}]\!](\mathfrak{B})$ is the set of all prefixes where divergence occurs or the error assertion fails.
- $\alpha_{\text{Pred}}[\![\mathcal{S}^{Max}]\!](\neg\mathfrak{B})$ is the set of all prefixes where divergence does not occur and the assertion does not fail.
- $\alpha_{\text{Pred}}[\![\mathcal{S}^{Max}]\!](\bot^{Max}) = \emptyset$ as no valid trace can guarantee $\bot^{Max}$.

In this thesis we associate with program prefix traces the strongest possible maximal trace property that can be inferred from them.

4.3 Forward Semantics of Expressions

To define the forward semantics of programs we must first give the forward semantics for arithmetic and boolean expressions. Responsibility analysis for floating-point programs can actually be made generic with respect to some underlying abstract domain $\mathbb{D}^\sharp$ as long as the following three conditions hold:

1. $\mathbb{D}^\sharp$ provides a way to compute $\mathbb{D}^\sharp_{\mathbb{R}}$ which provides a sound abstraction of the values of variables and arithmetic expressions with respect to real semantics,
2. there exists a way to obtain for each $x \in \mathbb{D}^\sharp_{\mathbb{R}}$ an abstract element $\hat{x} \in \mathbb{D}^\sharp_{\mathbb{F}}$ which corresponds to a sound approximation of x if all operations and declarations

used to obtain the value of x were replaced by floating-point numbers which has some error, and

3. $\mathbb{D}^\sharp$ is a mapping from variables to $\mathbb{D}^\sharp_\mathbb{R}$ and $\mathbb{D}^\sharp_\mathbb{F}$ (i.e. $\mathbb{D}^\sharp : \mathbb{V} \rightarrow (\mathbb{D}^\sharp_\mathbb{R} \times \mathbb{D}^\sharp_\mathbb{F})$).

Two possible ways to satisfy the above three conditions are to either compute an error value associated with each element of $\mathbb{D}^\sharp_\mathbb{R}$ such that corresponding elements in $\mathbb{D}^\sharp_\mathbb{F}$ can be recovered or to compute both $\mathbb{D}^\sharp_\mathbb{R}$ and $\mathbb{D}^\sharp_\mathbb{F}$ which then allows for the recovering of the error between the two values. The former approach is taken in [Tit+18], [GP11] and [GP13].

We also define $\mathbb{A}$ as the transfer functions associated with doing arithmetic operations and $\mathbb{B}$ be the function associated with boolean comparisons. As long as these transfer functions are shown to be sound we may use the domain $\mathbb{D}^\sharp$ throughout the responsibility analysis. In this work, $\mathbb{D}^\sharp$ will be isntantiated with the constrained affine sets domain from [GP13] which is described in Section 3.3.5.

4.3.1 Instantiating $\mathbb{D}^\sharp$

The constrained affine sets for floating-point analysis is a suitable candidate for the domain $\mathbb{D}^\sharp$. The constrained affine set $X = (R^X, E^X, \Phi^X_r, \Phi^X_f)$ lets us compute both a real value $D^\sharp_\mathbb{R}$ with the constrained affine set R^X. Then, $D^\sharp_\mathbb{F}$ can be computed by calculating $R^X + E^X$. Thus, we instantiate $D^\sharp$ using this domain and assume the existence of the functions $\mathcal{AS}$ as a transfer function for artihmetic expressions and assignments and $\mathcal{BS}$ for the computation of Boolean comparisons. Furthermore, we assume the existence of join $\sqcup_{\mathbb{D}^\sharp}$, meet $\sqcap_{\mathbb{D}^\sharp}$ and widening $\nabla_{\mathbb{D}^\sharp}$. All of these functions and operations are described in Section 3.3.5.

Definition 4.3.1. The semantics of arithmetic expressions is given by the function

$$\mathcal{A} \,:\, \mathbb{A} \times \mathbb{D}^\sharp \rightarrow \mathbb{D}^\sharp$$

where $\mathcal{A} \triangleq \mathcal{AS}$.

Definition 4.3.2. The semantics of Boolean comparison operators is given by the function

$$\mathcal{B} \,:\, \mathbb{B} \times \mathbb{D}^\sharp \rightarrow \mathbb{D}^\sharp$$

which is defined by the function $\mathcal{BS}$ from Section 3.3.5. To handle conjuncts, disjuncts and negations we proceed structurally on the syntax of Boolean expressions.

$$\begin{aligned}
\mathcal{B}[\![\mathtt{A}_1\; op^B\; \mathtt{A}_2]\!]X &\triangleq \mathcal{BS}[\![\mathtt{A}_1\; op^B\; \mathtt{A}_1]\!]X \\
\mathcal{B}[\![\mathtt{B}_1\;\mathtt{and}\;\mathtt{B}_1]\!]X &\triangleq \mathcal{B}[\![\mathtt{B}_1]\!]X \sqcap \mathcal{B}[\![\mathtt{B}_2]\!]X \\
\mathcal{B}[\![\mathtt{B}_1\;\mathtt{or}\;\mathtt{B}_1]\!]X &\triangleq \mathcal{B}[\![\mathtt{B}_1]\!]X \sqcup \mathcal{B}[\![\mathtt{B}_2]\!]X
\end{aligned}$$

The `not` operation is handled during a pre-processing phase where the negation operations are pushed into the other Boolean operations like in [Min06]. The negation of `and` and `or` follows from DeMorgan's Laws.

`not (B`$_1$` and B`$_2$`)`	$\rightarrow$	`(not B`$_1$`) or (not B`$_2$`)`
`not (B`$_1$` or B`$_2$`)`	$\rightarrow$	`(not B`$_1$`) and (not B`$_2$`)`
`not (not B)`	$\rightarrow$	`B`
`not (B`$_1$` < B`$_2$`)`	$\rightarrow$	`B`$_1$` >= B`$_2$
`not (B`$_1$` <= B`$_2$`)`	$\rightarrow$	`B`$_1$` > B`$_2$
`not (B`$_1$` > B`$_2$`)`	$\rightarrow$	`B`$_1$` <= B`$_2$
`not (B`$_1$` >= B`$_2$`)`	$\rightarrow$	`B`$_1$` < B`$_2$
`not (B`$_1$` = B`$_2$`)`	$\rightarrow$	`B`$_1$` <> B`$_2$
`not (B`$_1$` <> B`$_2$`)`	$\rightarrow$	`B`$_1$` = B`$_2$

The reason for this pre-processing is that for constrained affine sets the complementation operation may not exist for the underlying constraints, which is the case for octagons [Min06], or it might become too costly to compute complements of constraints via disjunctive completion [CC79] as in the case of intervals. Finally, the function

$$\bar{\mathcal{B}} \ : \ \mathbb{B} \rightarrow \mathbb{D}^{\sharp} \rightarrow \mathbb{D}^{\sharp}$$

computes the negation of a Boolean condition and is defined as $\mathcal{B}[\![\neg \mathtt{B}]\!]$.

The soundness of the above two definitions with respect to sets of both the real and floating-point numbers follows from the soundness of constrained affine sets as presented in [GGP10] and [GP13]. In turn, the soundness of the constrained affine sets follows from the soundness of a sound implementation of affine forms, such as in [GP11], and the soundness of the underlying domain used to interpret the constraints.

For example, the domain can be instantiated with a boxes interpretation for the noise symbols and we know that boxes are sound. The poset $(\mathbb{V} \rightarrow \wp(\mathbb{R}) \times \wp(\mathbb{R}), \subseteq)$ and the poset formed from the domain of constrained affine sets $(\mathbb{D}^\sharp, \subseteq_{\mathbb{D}^\sharp})$ forms a Galois connection

$$(\mathbb{V} \rightarrow \wp(\mathbb{R}) \times \wp(\mathbb{R}), \subseteq) \underset{\alpha_{\mathbb{D}^\sharp}}{\overset{\gamma_{\mathbb{D}^\sharp}}{\leftrightarrows}} (\mathbb{D}^\sharp, \sqsubseteq_{\mathbb{D}^\sharp}).$$

The order on $\mathbb{V} \rightarrow (\wp(\mathbb{R}) \times \wp(\mathbb{R}))$ is defined as point-wise set-inclusion on each variable and we know that the set of subsets of a power set ordered by inclusion is always a poset. The definitions of $\alpha_{\mathbb{D}^\sharp}$ and $\gamma_{\mathbb{D}^\sharp}$ are presented in Section 3.3.

4.4 Forward Semantics of Programs

The forward abstract affine semantics of program Program 2.1 can be computed by abstracting the forward reachability semantics of programs [CC04] to the invariants represented by the constrained affine sets. This gives way to a forward static analysis with forward iteration by abstract interpretation. The results of such an analysis can also be represented using a transition system (sometimes referred to as a flowchart or control-flow graph) [CC98] where the transitions correspond to program statements (or events) and each node is attached an invariant that is calculated by the analyses. The transition system of Program 2.1 is given in figure Figure 4.7 where transitions correspond to events in the system. In fact, this system is also equivalent to the trivial extension (see Section 3.5) of the non-tokenized system where we now use the empty token t_ϵ (in this case each node is marked as $[i]_\epsilon$).

Each node of the transition system is labelled above with the strongest property

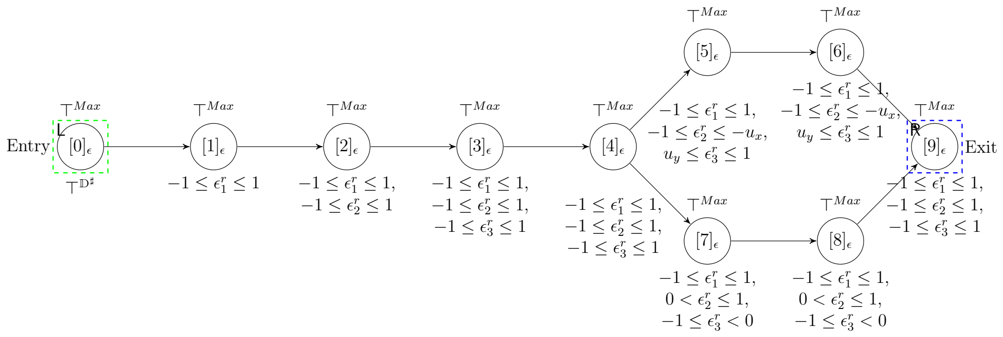

Entry
$[0]_\epsilon$
$\top^{Max}$
$\top^{\mathbb{D}^\sharp}$
$[1]_\epsilon$
$\top^{Max}$
$-1 \leq \epsilon_1^r \leq 1$
$[2]_\epsilon$
$\top^{Max}$
$-1 \leq \epsilon_1^r \leq 1,$
$-1 \leq \epsilon_2^r \leq 1$
$[3]_\epsilon$
$\top^{Max}$
$-1 \leq \epsilon_1^r \leq 1,$
$-1 \leq \epsilon_2^r \leq 1,$
$-1 \leq \epsilon_3^r \leq 1$
$[4]_\epsilon$
$\top^{Max}$
$-1 \leq \epsilon_1^r \leq 1,$
$-1 \leq \epsilon_2^r \leq 1,$
$-1 \leq \epsilon_3^r \leq 1$
$[5]_\epsilon$
$\top^{Max}$
$-1 \leq \epsilon_1^r \leq 1,$
$-1 \leq \epsilon_2^r \leq -u_x,$
$u_y \leq \epsilon_3^r \leq 1$
$[6]_\epsilon$
$\top^{Max}$
$-1 \leq \epsilon_1^r \leq 1,$
$-1 \leq \epsilon_2^r \leq -u_x,$
$u_y \leq \epsilon_3^r \leq 1$
$[7]_\epsilon$
$\top^{Max}$
$-1 \leq \epsilon_1^r \leq 1,$
$0 < \epsilon_2^r \leq 1,$
$-1 \leq \epsilon_3^r < 0$
$[8]_\epsilon$
$\top^{Max}$
$-1 \leq \epsilon_1^r \leq 1,$
$0 < \epsilon_2^r \leq 1,$
$-1 \leq \epsilon_3^r < 0$
$[9]_\epsilon$
$\top^{Max}$
Exit
$-1 \leq \epsilon_1^r \leq 1,$
$-1 \leq \epsilon_2^r \leq 1,$
$-1 \leq \epsilon_3^r \leq 1$

Figure 4.6: Original transition system of Program 2.1.

that can be guaranteed after that point where the properties come from the lattice given in Figure 3.2 and Section 4.2. The reason every node is labelled $\top^{Max}$ is because both conditional divergence and non-divergence occur in Program 2.1 but the transition system does not split the analysis into four cases to distinguish between them. Instead, the analysis will compute the condition for the if branch and the else branch and once a join happens it will also introduce the difference of the results of the two branches as a discontinuity term as described in [GP13]. However, we will no longer be computing the terms here and instead perform partitions. Below each node, we give the set of constraints the affine analysis will derive for each program point. We start with the empty set represented by $\top^{\mathbb{D}^\sharp}$. Each transition either adds a constraint to the set or refines an existing constraint. For the sake of brevity, only the constraints on the symbolic error symbols ϵ_i^r are given (and they are actually the only ones needed for this example). The introduction of the new error symbol due to arithmetic that occurs from the transition from nodes [3] to [4] is omitted.

Let the right bound of the responsible entities for a system behavior be defined as the first node n from the end node of the transition system that guarantees the behavior $\mathfrak{B}$ or $\neg\mathfrak{B}$ after any outgoing transition from n. This means that if all the outgoing transitions of a node only lead to one behavior and the node is the left-most such node then it is a right bound for that behavior. If no such node exists then the last node of the transition system is the right bound. This definition makes sense because if we can show in the abstract that all traces from n lead to $\mathfrak{B}/\neg\mathfrak{B}$ then it must be that either the concrete traces obtained by the concretization of n will also lead to $\mathfrak{B}/\neg\mathfrak{B}$ or there is a possibility of a false positive. We denote the right bounds of a system with a blue box around the node and labeled R at the top left corner of the box.

For Figure 4.6, it is clear that the right bound for the bad behavior $\mathfrak{B}$ must be the node $[9]_\epsilon$ as every node is marked with $\top^{Max}$. We could say that location 9 is the right bound of responsibility but clearly this is unsatisfactory as the result is too coarse. So, we need to refine Figure 4.6 in some way to obtain a more precise abstraction. The key insight lies in the fact that the affine domain actually computes the conditions of discontinuity in its forward pass. Thus, it is possible to use trace partitioning [RM07] to refine Figure 4.6 in our forward computation to the 4 possible cases of 'no if discontinuity,' 'if discontinuity', 'no else discontinuity,' and 'else discontinuity'. This will lead to some gains in precision.

We formalize the forward partitioning semantics of programs. First, the abstract domain and the partitioning tokens are defined. Then, we describe the partitioning functions we will use for our semantics. Finally, the forward semantics is presented as a refinement of a generic abstract interpreter similar to the ones presented in [Cou19] and [Cou99].

4.4.1 Abstract Responsibility Domain

Definition 4.4.1. The abstract responsibility domain

$$\mathbb{D}^\sharp_R \triangleq \mathbb{D}^\sharp \times \mathcal{L}^{Max}$$

is defined as the product of abstract elements $\mathbb{D}^\sharp$ and the set of system behaviors of interest $\mathcal{L}^{Max}$.

An instance of $\mathbb{D}^\sharp_R$ is given as $D^\sharp_R = (D^\sharp, \mathcal{P})$. Here, $D^\sharp$ is a constrained affine set associated with a program point and $\mathcal{P}$ is the strongest program property that can be inferred from $D^\sharp$. By strongest property that can be inferred we mean that $D^\sharp$ implies that property $\mathcal{P}$ holds if we use $D^\sharp$ as an abstraction of the program's prefix traces. There is an implicit lifting of the states $D^\sharp$ characterizes to program traces which is defined in Section 4.4.5 where the forward semantics of programs are given. So, an alternative definition of the strongest property that can be inferred from $D^\sharp$ is the smallest set $\mathcal{P}$ from the lattice of system behaviors such that the prefix traces described by $D^\sharp$ is a subset of $\mathcal{P}$. If $\mathcal{P}$ is $\top^{Max}$ then all traces are responsible for either $\mathfrak{B}$ or $\neg\mathfrak{B}$.

Definition 4.4.2. Let ${}^1D^\sharp_R = ({}^1D^\sharp, {}^1\mathcal{P})$ and ${}^2D^\sharp_R = ({}^2D^\sharp, {}^2\mathcal{P})$. Then, we define the join $\sqcup_{\mathbb{D}^\sharp_R}$ and meet $\sqcup_{\mathbb{D}^\sharp_R}$ of the two abstract elements component-wise

$${}^1D^\sharp_R \sqcup_{\mathbb{D}^\sharp_R} {}^2D^\sharp_R \triangleq ({}^1D^\sharp \sqcup_{\mathbb{D}^\sharp} {}^2D^\sharp, {}^1\mathcal{P} \cup {}^2\mathcal{P}),$$

$${}^1D^\sharp_R \sqcap_{\mathbb{D}^\sharp_R} {}^2D^\sharp_R \triangleq ({}^1D^\sharp \sqcap_{\mathbb{D}^\sharp} {}^2D^\sharp, {}^1\mathcal{P} \cap {}^2\mathcal{P}).$$

Similarly, the order relation $\sqsubseteq_{\mathbb{D}^\sharp_R}$ is defined as

$$^1D^\sharp_R \sqsubseteq_{\mathbb{D}^\sharp_R} {}^2D^\sharp_R \iff {}^1D^\sharp \sqsubseteq_{\mathbb{D}^\sharp} {}^2D^\sharp \wedge {}^1\mathcal{P} \subseteq {}^2\mathcal{P}.$$

From the above definitions, it is evident that

$$\top^{\mathbb{D}^\sharp_R} = (\top^{\mathbb{D}^\sharp}, \top^{Max}),$$
$$\bot^{\mathbb{D}^\sharp_R} = (\bot^{\mathbb{D}^\sharp}, \bot^{Max}).$$

Note that we denote the ordering of the elements of the lattice of system behaviors using subset inclusion $\subseteq$. However, the concrete responsibility semantics of programs is not computable meaning such a subset check cannot be implemented. So, we must abstract these hyperproperties to symbolic terms that represent them. Then, there is an abstract ordering of responsiblity properties which is equivalent to the ordering on the Hasse diagram given in Figure 3.2. Throughout this thesis we overload $\subseteq$ to mean both concrete and abstract inclusion which means that elements $\mathcal{P}$ can be thought of as either concrete or abstract elements of the responsibility domain.

Proposition 4.4.1. The poset $(\mathbb{D}^\sharp_R, \sqsubseteq_{\mathbb{D}^\sharp_R})$ forms a complete lattice

$$(\mathbb{D}^\sharp_R, \sqsubseteq_{\mathbb{D}^\sharp_R}, \bot^{\mathbb{D}^\sharp_R}, \top^{\mathbb{D}^\sharp_R}, \sqcup_{\mathbb{D}^\sharp_R}, \sqcap_{\mathbb{D}^\sharp_R}).$$

There is a simple reduced product between $\mathbb{D}^\sharp$ and $\mathcal{L}^{Max}$ [CCM11]. If either element of the pair $(D^\sharp, \mathcal{P})$ is the bottom element of their respective lattices then it must be that the other is also bottom of its respective lattice. We do not explicitly write a reduced product operator for any of our abstract operations but it is implicit that such reductions are done throughout the analysis.

4.4.2 Partitioning Directives

For abstract responsibility analysis for floating-point programs we will be using a single partitioning token. Unlike [RM07], the goal is not to partition all control flows of programs but rather to partition control flows depending only on the values of variables. The intuition behind this approach follows from the definition of responsibility. We say that a responsible entity for a behavior $\mathfrak{B}$ (or $\neg\mathfrak{B}$) is one that is part of a trace that exhibits $\mathfrak{B}$ and it must be the first entity that chooses its value such that $\mathfrak{B}$ is guaranteed to occur. Thus, we wish to separate sets of traces into those that guarantee $\mathfrak{B}$ occurs when some variable chooses a real value and its corresponding floating-point value leads to the error. This type of partitioning is referred to as a value-based trace partitioning [MR05].

We define two partitioning directives which constitute a subset of the partitioning directives defined in [RM07]. Each directive includes a label so that we discern the program point where the directive was created and keep track of a history of directives. For example, given two tokens t_1 and t_2 it is possible to say that the partition associated with t_1 was created before t_2. Taking this further, if we associate t_1 with program location ℓ_1 and t_2 with location ℓ_2 then we can obtain even more refined information of the form partition t_2 created at ℓ_2 is a partitioning of t_1 created at ℓ_1.

Definition 4.4.3. The set of partitioning directives, denoted $\mathbb{D}$, consists of tokens $t^{\ell}_{\mathbf{val}}$ which indicate that a value based partitioning has been performed at point ℓ and the null directives $t_{\mathbf{None}}$ and $t^{m}_{\mathbf{None}}$ which indicate that no partitioning has been performed.

Directives of the form $t^m_{\mathbf{None}}$ and $t_{\mathbf{None}}$ are used for partition merging just like in [RM07]. For example, we may wish to collapse multiple partitions into a single partition and remove the tokens from the set of partition directives. In that case, we may replace all collapsed directives by the null directive that forgets the control flow information of the partitions we just removed. The only difference between the two tokens is that $t^m_{\mathbf{None}}$ indicates a possible loss information due to a merge while $t_{\mathbf{None}}$ is a place-holder to ensure two trees have the same height. The purpose of $t^m_{\mathbf{None}}$ will become clearer when we reach the backward semantics of programs in Section 4.5. There is also a practical purpose for having the token $t^m_{\mathbf{None}}$ as an implementation of our partitioning scheme may want to distinguish between merge points that could potentially lose information from those that do not lose any information to see where the analysis may encounter problems.

The mapping of partitions to elements of the abstract resposibility domain will correspond to a partitioned abstract program environment. So, instead of mapping variables to values, which is a standard construction in abstract interpretation, we want to map partitions to variables to values. Furthermore, as stated previously, we will keep a history of partitions that lead to a current partition. Trees are a good representation for storing this kind of history as multiple partitions may share the same history which can be compactly represented in tree form. Thus, abstract program environments are defined inductively using a recursive tree definition.

Definition 4.4.4 ([RM07])**.** An abstract environment $\mathcal{E} \in \mathbb{D} \to \mathbb{D}^\sharp_R$ is of the form

$$\begin{aligned} \mathcal{E} ::= \ & \text{leaf}[D^\sharp_R] && \text{where } D^\sharp_R \in \mathbb{D}^\sharp_R, \\ \mid \ & \text{node}[\phi] && \text{where } \phi \in \mathbb{D} \to (\mathbb{D} \to \mathbb{D}^\sharp_R). \end{aligned}$$

The set of abstract environments is denoted by $\mathbb{E}$.

The join of two abstract environments can be computed using the scheme given in [RM07]. The idea is to consider the set of paths in both tree environments and if there exist two tokens t_0 and t_1 such that t_0 is a strict prefix of t_1 then t_0 is replaced by a node with the token $t_{\mathbf{None}}$ that maps to t_0. We create a new tree such that the prefix sequences that are common to both trees are shared in the resulting representation. Then, the join is simply defined as the point-wise join of both environment where each corresponding leaves of the tree are joined to creating the resulting joined tree.

Definition 4.4.5 ([RM07])**.** The function

$$\text{join}_t \; : \; \mathbb{E} \times \mathbb{E} \rightarrow \mathbb{E}$$

takes two abstract environments $\mathcal{E}_1$ and $\mathcal{E}_2$ and produces the join of them $\mathcal{E}$ by using $\mathbb{D}_R^\sharp$ point-wise on the leaves of the joined tree.

To generate and destroy tokens we also define the partition creation and merging functions. To add a token to the current environment we will keep all current tokens and add a token just above the leaf levels of the tree. The history of partitions for any one leaf can be obtained by tree traversal from the root of the tree.

Definition 4.4.6 ([RM07])**.** The function

$$\text{generate} \; : \; \wp(\mathbb{D}) \times \mathbb{E} \rightarrow \mathbb{E}$$

takes as input a set of directives δ and an abstract environment $\mathcal{E}$ and further partitions as follows.

$$\text{generate}(\delta, \text{leaf}[D^\sharp_R]) = \text{node}[\lambda(t \in \delta).\, \text{leaf}[D^\sharp_R]]$$

$$\text{generate}(\delta, \text{node}[\phi]) = \text{node}[\lambda(t \in \text{dom}(\phi)).\, \text{generate}(\delta, \phi(t))]$$

Throughout partition generation we assume that when adding new directives, unique names are introduced for them. For merging partitions, we remove the relevant token, replace it by either the null merge directive $t^m_{\textbf{None}}$ or $t_{\textbf{None}}$ depending on whether the join lost information. After the token replacement, the join of all children of the sub-tree induced by the token is taken to collapse the partitions. This means that the loss of information due to a join is characterized as whether the join forced the strongest property inferred to become $\top^{Max}$. If the resulting abstract environment still either guarantees $\mathfrak{B}$ or $\neg\mathfrak{B}$, then clearly this operation did not lose any information with respect to responsibility.

Definition 4.4.7 ([RM07])**.** The function

$$\text{merge} \;:\; \wp(\mathbb{D}) \times \mathbb{E} \to \mathbb{E}$$

takes as input a set of directives δ and an abstract environment $\mathcal{E}$ and removes all partitions $t \in \delta$ from $\mathcal{E}$.

$$\begin{aligned}
\text{merge}(D, \text{leaf}[d]) &= \text{leaf}[d] \\
\text{merge}(D, \text{node}[\phi]) &= \text{node}[\phi'] \\
\text{where } \phi' &\triangleq \begin{cases} t \notin D \mapsto \text{merge}(D, \phi(t)) \\ t' \mapsto \bigsqcup_{\mathbb{D}^\sharp_R} \{D^\sharp_R \text{ at a leaf of } \phi(t) \mid t \in D\} \end{cases} \\
\text{and where } t' &= \begin{cases} t^m_{\textbf{None}} & \text{if } \bigsqcup_{\mathbb{D}^\sharp_R} \text{ results in } \top^{Max}, \\ t_{\textbf{None}} & \text{otherwise.} \end{cases}
\end{aligned}$$

Finally, to interpret tests and assignments for partitions we can apply the semantic function for tests and arithmetic on each leaf of the partition tree.

Definition 4.4.8 ([RM07])**.** The function

$$\text{test}_{\mathcal{B}} \;:\; \mathbb{B} \times \mathbb{E} \to \mathbb{E}$$

takes as input a semantic function $\mathcal{B}$ that interprets tests, a Boolean test `B` and an environment $\mathcal{E}$. The result is a new environment where all leaves have been restricted by the semantics of tests.

$$\begin{aligned}
\text{test}_{\mathcal{B}}(\mathtt{B}, \text{leaf}[(D^\sharp, \mathcal{P})]) &= \text{leaf}[(\mathcal{B}[\![\mathtt{B}]\!]D^\sharp, \mathcal{P})] \\
\text{test}_{\mathcal{B}}(\mathtt{B}, \text{node}[\phi]) &= \text{node}[\lambda(t \in \text{dom}(\phi)).\, \text{test}_{\mathcal{B}}(\mathtt{B}, \phi(t))]
\end{aligned}$$

Definition 4.4.9 ([RM07])**.** The function

$$\text{assn}_{\mathcal{AD}} \;:\; \mathbb{A} \times \mathbb{E} \to \mathbb{E}$$

takes as input a semantic function $\mathcal{AS}$ that interprets assignments, an arithmetic operation $\mathtt{A}$ and an environment $\mathcal{E}$. The result is a new environment where all leaves have been re-interpreted under the assignment.

$$\begin{aligned}\mathrm{assn}_{\mathcal{AS}}(\mathtt{A}, \mathrm{leaf}[(D^\sharp, \mathcal{P})]) &= \mathrm{leaf}[(\mathcal{AS}[\![\mathtt{A}]\!]D^\sharp, \mathcal{P})] \\ \mathrm{assn}_{\mathcal{AS}}(\mathtt{A}, \mathrm{node}[\phi]) &= \mathrm{node}[\lambda(t \in \mathrm{dom}(\phi)).\, \mathrm{assn}_{\mathcal{AS}}(\mathtt{A}, \phi(t))]\end{aligned}$$

4.4.3 Partitioning Functions

Besides the functions associated with directives from the previous section, it is also necessary to define partitioning functions based on the results of the constrained affine analysis. The goal is to identify when a bad behavior occurs, which is detected using constrained affine sets. Then, based on the values we obtain, we will perform partitions on the traces of the program to refine the transition system and possibly improve the bounds of responsibility.

The constrained affine sets are able to detect when a test discontinuity occurs. For example, if $\Phi^r_{\mathtt{if}}$ is the set of real constraints on the noise symbols associated with the if-branch and $\Phi^f_{\mathtt{else}}$ is the set of floating-point constraints on the else-branch then the intersection $\Phi^r_{\mathtt{if}} \cap \Phi^f_{\mathtt{else}}$ gives us the set of constraints on the real noise symbols that lead to test discontinuity. Similarly, taking the intersection of $\Phi^r_{\mathtt{if}}$ and $\Phi^f_{\mathtt{if}}$ gives us the set of constraints on the real noise symbols where both the real values and the floating-point values take the same branch. We overload $\Phi^r_{\mathtt{if}} \cap \Phi^f_{\mathtt{if}}$ to also give constraints on the floating-point noise symbols. In practice, it is possible to realize this intersection by adding all real noise symbols in $\Phi^f_{\mathtt{if}}$ to $\Phi^r_{\mathtt{if}}$ and then taking the intersection.

These refined constraints may be used to perform partitions. The constraints given by $\Phi^r_{\texttt{if}} \cap \Phi^f_{\texttt{else}}$ guarantee that an affine set will always lead to the behavior $\mathfrak{B}$ from the point of its detection and onwards. This result follows from the soundness of constrained affine sets. Thus, conditional divergence may be detected when the semantics of Boolean tests are being evaluated. A first idea is to partition the abstract environments right after the interpretation of tests. However this idea will not work and the actual partitioning of the affine sets must occur during the join after each branch of the conditional is evaluated. This is because the results of the computations that occur in each branch of the test are interpreted independetly. That is, the real value may go one branch and the floating-point another branch but we cannot take both branches simultaneously. This means that if divergence occurs we must consolidate the result of the two branches after we compute the results for the real value that took some branch and the floating-point value that took a different branch. We refer to these consolidating points as join points.

At any join point, the relevant intersections for the constraints on the affine sets may be computed and then the result partitioned based on these values. To this end, we define a partitioning function that we will use at the join points of conditional statements.

Definition 4.4.10. The partitioning join $\sqcup^{\ell}_{P}$ of two abstract elements ${}^{1}D^{\sharp}_{R} = ({}^{1}D^{\sharp}, {}^{1}\mathcal{P})$ and ${}^{2}D^{\sharp}_{R} = ({}^{2}D^{\sharp}, {}^{2}\mathcal{P})$ at program location ℓ splits the join into the four cases of agreement and disagreement. Let

$$((R^X, E^X, \Phi^X_r, \Phi^X_f), {}^{1}\mathcal{P}) = ({}^{1}D^{\sharp}, {}^{1}\mathcal{P})$$

and

$$((R^Y, E^Y, \Phi^Y_r, \Phi^Y_f), {}^{2}\mathcal{P}) = ({}^{2}D^{\sharp}, {}^{2}\mathcal{P})$$

be the decomposition of the abstract elements to their affine sets. Then, $\sqcup^{\ell}_{P}$ is defined as

$$\sqcup^{\ell}_{P}\,(({}^1D^\sharp, {}^1\mathcal{P}), ({}^2D^\sharp, {}^1\mathcal{P})) = \text{node}[\phi]$$

$$\text{where } \phi \triangleq \begin{cases} {}^1t^{\ell}_{\textbf{val}} \mapsto \text{leaf}[((R^X, E^X, \Phi^X_r \cap \Phi^X_f, \Phi^X_r \cap \Phi^X_f), {}^1\mathcal{P})] \\ {}^2t^{\ell}_{\textbf{val}} \mapsto \text{leaf}[((R^X, E^X + (R^Y - R^X), \Phi^X_r \cap \Phi^Y_f, \Phi^X_r \cap \Phi^Y_f), \mathfrak{B})] \\ {}^3t^{\ell}_{\textbf{val}} \mapsto \text{leaf}[((R^Y, E^Y, \Phi^Y_r \cap \Phi^Y_f, \Phi^Y_r \cap \Phi^Y_f), {}^2\mathcal{P})] \\ {}^4t^{\ell}_{\textbf{val}} \mapsto \text{leaf}[((R^Y, E^Y + (R^X - R^Y), \Phi^Y_r \cap \Phi^X_f, \Phi^Y_r \cap \Phi^X_f), \mathfrak{B})] \end{cases}$$

where the result of the join that would normally occur here is split into the four cases. The terms $(R^Y - R^X)$ and $(R^X - R^Y)$ introduce freshly named noise symbols similar to the discontinuity terms in [GP13]. The token ${}^1t^{\ell}_{\textbf{val}}$ corresponds to if agreement, ${}^2t^{\ell}_{\textbf{val}}$ to if divergence, ${}^3t^{\ell}_{\textbf{val}}$ to else agreement and ${}^4t^{\ell}_{\textbf{val}}$ to else disagreement. If any one of the tokens maps to $\bot^{\mathbb{D}^\sharp_R}$ then that partition is omitted as there is no purpose in partitioning based on bottom.

Note that the above definition partitions any two abstract elements into 4 partitions but does not actually consider the values of ${}^1\mathcal{P}$ and ${}^2\mathcal{P}$. If both are $\mathfrak{B}$ then there is no added value in performing any more partitions as the behavior $\mathfrak{B}$ has already occurred meaning a responsible entity must be somewhere before the join point. The same reasoning holds when both behaviors are $\neg\mathfrak{B}$ as the join of these traces must continue to guarantee that $\neg\mathfrak{B}$ holds. Thus, in these cases the analysis should actually fall back to using $\sqcup_{\mathbb{D}^\sharp_R}$ for such pairs of abstract elements. While $\sqcup_P$ is given as a very general definition, it is assumed that such checks occur so that the number of partitions created can be lowered. Furthermore, if one of the branches is $\mathfrak{B}$ and the other is $\neg\mathfrak{B}$ we may simply perform no processing and keep the partitions as is. Thus, the 4-partitioning only occurs when either ${}^1\mathcal{P}$ or ${}^2\mathcal{P}$ is $\top^{Max}$.

An alternative definition of $\sqcup_P$ is to collapse the non-discontinuity and discontinuity cases together to only produce 2 partitions rather than 4. This could lower the cost of partitioning. However, the join of two constrained affine sets may lose a lot of information in certain cases. Thus, further investigation into this alternative definition is required and experimental evaluation could prove useful. With the current definition of $\sqcup_P$ we essentially ensure that no information loss occurs as 'traditional' joins $\sqcup_{\mathbb{D}^\sharp_R}$ have been eliminated and the intersection of the constraints on affine symbols is exact.

Proposition 4.4.2. The operation $\sqcup_P$ is a complete partitioning of its arguments.

Proposition 4.4.2 follows from the fact that the join of two abstract elements are split into four partitions where taking the join over all partitions gives the join of the original two abstract elements.

Definition 4.4.10 can be used to define the join any two partitioned abstract environments.

Definition 4.4.11. The partitioning join $\sqcup_{\texttt{ite}} : \mathbb{E} \times \mathbb{E} \to \mathbb{E}$ takes two abstract environments $\mathcal{E}_{\texttt{i}}$ and $\mathcal{E}_{\texttt{e}}$ that correspond to the interpretation of the if and else branches respectively. The output is a new environment that is a partitioning of $\mathcal{E}$. $\sqcup_{\texttt{ite}}$ is defined as $\sqcup_t$ but now the use of $\sqcup_{\mathbb{D}^\sharp_R}$ is replaced with $\sqcup_P$.

The above definition allows us to match each leaf node of $\mathcal{E}$ that took the if branch with its corresponding value for the else branch. Since if-then-else statements may nest, care must be taken for when the two branches $\mathcal{E}_{\texttt{i}}$ and $\mathcal{E}_{\texttt{e}}$ do not share the same tokens and have different heights. In this case, we introduce null tokens to the tree so that the trees are structurally the same. This partitioning scheme may end up being too costly in degenerate cases due to possible exponential blow-up. For example, assume before evaluating an if-then-else statement there is only 1 partition and the if branch has another if statement inside it while the else branch has no further branching. Then, in the worst case where discontinuity occurs at every test, the partitioning will produce 16 partitions upon exit. This is because the if branch's environment will have 4 partitions due to discontinuity and when joining with the else branch we will create 4 partitions per partition from the if branch. While such degenerate cases are unlikely to occur often, a merging strategy for when they do occur might be useful.

Lemma 4.4.1. *The partitioning $\sqcup_{ite}$ is a complete partitioning as long as $\mathcal{E}_{i}$ and $\mathcal{E}_{e}$ are complete partitionings.*

Proof. The proof of the above lemma follows from induction. As a base case consider when $\mathcal{E}_{\mathtt{i}}$ and $\mathcal{E}_{\mathtt{e}}$ have no uncommon partitions. Then the partitioning using $\sqcup_{\mathtt{ite}}$ is a complete partitioning because the underlying function $\sqcup_P$ is a complete partitioning. Next, we may assume that any $\mathcal{E}_{\mathtt{i}}$ and $\mathcal{E}_{\mathtt{e}}$ is generated from this base case. Since $\sqcup_{\mathtt{ite}}$ is complete in the base case, it is also complete at each step it is used from that point on. All that remains to consider is whether the branches are partitioned in any other way except for the partitions generated from $\sqcup_{\mathtt{ite}}$. In this case, we assume that if such other partitions are created then they are also complete which completes the proof sketch. □

In addition to conditional branching, test divergence may also occur at loop heads. That is, when evaluating a test for whether an iterate of a while loop should be executed, the floating-point and real values may evaluate to different conditions.

This means that the while loop could potentially execute for a different amount of iterations depending on whether the value we are considering is floating-point or not. In some cases the loop may even fail to terminate depending on the value. However, we must not make the mistake of partitioning the conditional branches during the evaluation of the iterations of the loop iterates. This is because, as noted before, the constrained affine sets interpret the results of tests independently on the real and floating-point affine sets. Thus, it is actually sufficient to perform the partitioning $\sqcup_{\texttt{ite}}$ upon exit from the loop as this will soundly split the loop into the divergent and non-divergent cases. If either the floating-point or real value leads to a non-terminating loop then upon exit we will either assign the real value ∞ or associate with it an infinite error.

Besides the partitioning functions defined above we need one more partitioning function for the case of checking whether the rounding-error of a program has exceeded some threshold n. This partitioning function is similar to the interpretation of tests where we can simply work towards the leaves of the partitioning tree and perform partitions by introducing another level in the tree. These partitions are performed only if the floating-point rounding error of some variable has exceeded the threshold.

Definition 4.4.12. The error-threshold partitioning function $\mathrm{e}_n^\ell : \mathbb{L} \times \mathbb{V} \times \mathbb{E} \to \mathbb{E}$ takes a program variable $\mathtt{x} \in \mathbb{V}$ and an abstract environment $\mathcal{E}$ at program location ℓ and checks all leaves of the partitioning tree on whether that variable's floating-point error has exceeded the threshold n. For a constrained affine set $D^\sharp$, we define $D^\sharp[\mathtt{x}]$ as the operation that retrieves the variable $\mathtt{x}$ from the affine sets and the relevant noise symbols for the affine terms in those sets.

$$\mathrm{e}^{\ell}_{n}(\mathtt{x}, \mathrm{leaf}[(D^\sharp, \mathcal{P})]) = \begin{cases} \mathrm{node}[t^{\ell}_{\mathbf{val}} \mapsto \mathrm{leaf}[(D^\sharp, \mathfrak{B})]] & \text{if } \mathcal{P} \neq \mathfrak{B} \wedge \gamma_{\mathbb{D}^\sharp}(E^X[\mathtt{x}]) \geq n, \\ \mathrm{node}[t_{\mathbf{None}} \mapsto \mathrm{leaf}[(D^\sharp, \mathcal{P})]] & \text{otherwise.} \end{cases}$$

$$\mathrm{e}^{\ell}_{n}(\mathtt{x}, \mathrm{node}[\phi]) = \mathrm{node}[\lambda(t \in \mathrm{dom}(\phi)).\, \mathrm{e}^{\ell}_{n}(\mathtt{x}, \phi(t))]$$

This function does not perform actual partitions but instead serves as a filter on the leaves of the tree. Thus, it is a trivially complete partitioning of the input environment $\mathcal{E}$.

The partitioning functions described in this section will be used to dynamically partition the forward semantics of programs.

4.4.4 Partitioning Hints

The partitioning strategies of the previous section will lead to some improvements in the discovery of the bounds of responsibility. However, we wish to obtain more refined results. To that end, at each program location we will compute what we refer to as partitioning hints. These hints give underapproximations of sufficient conditions for ending up in a state that can be marked as $\mathfrak{B}$ or $\neg\mathfrak{B}$. This means that every trace described by the hint invariant guarantees $\mathfrak{B}/\neg\mathfrak{B}$ occurs at some point, possibly later, in the computation. We do not outline how these conditions are computed here but instead assume that there exists a function that gives a set of such invariants along with the property it guarantees for each program location.

Definition 4.4.13. The function $\mathcal{H} \in \mathbb{L} \to (\mathbb{D} \to \mathbb{D}^\sharp_R)$ attaches to each program point a set of abstract elements of the form $(D^\sharp, \mathcal{P})$. If the behavior is $\mathcal{P} = \mathfrak{B}$ or

$\neg\mathfrak{B}$ then all traces described by $D^\sharp$ are sufficient to cause $\mathfrak{B}$ or $\neg\mathfrak{B}$ respectively.

The structure of $\mathcal{H}$ is a tree which is equivalent to the final abstract environment $\mathcal{E}$ that is produced by the forward analysis. The backward semantics of programs, as described in Section 4.5, will compute for each leaf of this tree new abstract invariants that are the sufficient conditions to reach the invariants in $\mathcal{E}$. Then, in another pass of the forward semantics the hints may be used to further partition the program. The idea is to first compute the new abstract environment at some program location and then partition each leaf node further using the hints in $\mathcal{H}$. This way we can associate with each program point the sets of traces that cause bad behaviors much earlier than when the partitioning functions of Section 4.4.3 detect such behaviors. Since these hints describe an under-approximation of the sufficient conditions that cause $\mathfrak{B}$, it must be that the concrete traces described by the hint contain a responsible entity for $\mathfrak{B}$ somewhere in their prefixes.

There is a slight caveat to the partitioning proposed here as we require the existence of a complementation operation for constrained affine sets. That is, if we have a set of traces ${}^1D^\sharp$ with behavior $\top^{Max}$ and we know another set of traces ${}^2D^\sharp \sqsubseteq_{\mathbb{D}^\sharp} {}^1D^\sharp$ with behavior $\mathfrak{B}$ then we would like to split ${}^1D^\sharp$ into ${}^2D^\sharp$ and ${}^1D^\sharp \setminus {}^2D^\sharp$. However, such an operation does not exist for zonotopes as the complement of any zonotope is not necessarily convex. This also holds for other numeric domains such as boxes, octagons and polyhedra. One option is to define the complement set in terms of boxes which can be abstracted back into constrained affine sets. Since there is a Galois connection between the two domains and the composition $\alpha \circ \gamma$ will always produce the same abstract element in terms of its concretization we may first concretize any $D^\sharp$ into boxes and then represent the complement set as a disjunction of boxes. This representation is known as disjunctive completion

[Min+17]. For example, given a pair of intervals $[a, b] \times [c, d] \in \mathbb{R}^2 \times \mathbb{R}^2$ which forms a rectangle we may represent the complement set as the set of disjuncts $\{[-\infty, \infty] \times [-\infty, c], [-\infty, \infty] \times [d, \infty], [-\infty, a] \times [c, d], [b, \infty] \times [c, d]\}$. A naive implementation for a cube would lead to 2^3 disjunctions but there also exists a construction with 2^2 of them. As the size of the set increases the representation of the complement will increase which may cause the analysis to get unwieldly, considering the fact that we are already creating many other partitions. Another drawback of this approach is that the relationships between variables will be lost as the concretization function will throw away all noise symbols and the re-abstraction will introduce fresh ones.

Instead of relying on disjunctive representations of complements we will use a notion similar to pseudo-complements of abstract elements [Cor+97]. Given an element D from a complete lattice L, its pseudo-complement S is the most general element such that $D \sqcap S = \bot$. If the pseudo-complement of $D \in L$ exists then we may define it as $S = \bigsqcup\{s \in C \mid D \sqcap s = \bot\}$. In our case we do not seek to find a pseudo-complement of abstract elements but instead we define the partitioning set of an abstract element.

Definition 4.4.14. Let D be an element of a complete lattice L. The set $S \in \wp(L)$ is a covering set for D if and only if $\bigsqcup S = D$ and $\bigsqcap S = \bot$. If all elements of S are also pair-wise independent of each other, that is for any element $x, y \in S$ we have $\neg(x \sqsubseteq y \wedge y \sqsubseteq x) \implies x \sqcap y = \bot$, then S is also a partitioning set for D.

Definition 4.4.15. Let t be a partitioning token of $\mathcal{E} \in \mathbb{E}$ that points to a leaf node leaf$[D^\sharp_R]$ and $\mathcal{H}(\ell) \in \mathbb{D} \to \mathbb{D}^\sharp_R$ be the partitioning hint tree associated with some program location ℓ. If the tree for $\mathcal{E}$ contains any new tokens besides those of the form $t_{\textbf{None}}$ then $\mathcal{H}(\ell)$ cannot contain a candidate partitioning set for elements of $\mathcal{E}$. Otherwise, define S to be the set of hints at the leaves of the sub-tree induced by t in $\mathcal{H}(\ell)$:

$$S = \{(D^\sharp, \mathcal{P}) \text{ at a leaf of } t \mid t \in \mathcal{H}(\ell)\}.$$

If $\bigsqcup_{\mathbb{D}^\sharp_R} S = D^\sharp_R$ and if for all non-equal pairs $({}^1D^\sharp, {}^1\mathcal{P}), ({}^2D^\sharp, {}^2\mathcal{P}) \in S$ there exists a variable $\mathtt{x}$ for each pair (this variable might change from pair to pair) such that ${}^1D[\mathtt{x}]^\sharp \sqcap_{\mathbb{D}^\sharp} {}^2D[\mathtt{x}]^\sharp = \bot_{\mathbb{D}^\sharp}$ then S is a partitioning set for D.

Definition 4.4.15 might seem like a relaxation of Definition 4.4.14 but this is not the case. If all pairs indeed split on some variable, i.e. they describe different ranges for that variable, then these must be disjoint zonotopes in a single dimension. This in turn means that elements of S partition the values of one or more variables and S is a partitioning set of D. Furthermore, this definition allows us to define more refined partitions. For example, if the forward semantics already creates a few partitions before the partitioning hint is applied, then not all of the leaves of $\mathcal{H}(\ell)$ may be a part of the partitioning set S. In fact, it is likely in this case that for any leaf in $\mathcal{E}$ the join over all leaves of $\mathcal{H}(\ell)$ produces a much larger invariant than that leaf. We take advantage of the fact that we store histories of partitions which we then use to find a candidate set of leaves. If a current environment is a strict prefix of the hint environment then it must be that the forward pass will eventually produce the same structure if no partitioning using hints is done. Recall that this is the case because the tree for $\mathcal{H}$ has the same structure as the tree from the resulting abstract environment of a previous forward pass of the program. Any two iterates of the forward semantics using just the functions from Section 4.4.3 will necessarily produce the same result. This is why Definition 4.4.15 checks for whether the trees $\mathcal{E}$ and $\mathcal{H}(\ell)$ have any different tokens because if they do then the guarantee just described no longer holds. Now, we can use Definition 4.4.15 and the function $\mathcal{H}$ to define complementing partitions of abstract environments $\mathcal{E}$.

Definition 4.4.16. Given partitioning hints $\mathcal{H}$ at program location ℓ (which can be obtained by $\mathcal{H}(\ell)$) and an abstract environment $\mathcal{E} \in \mathbb{E}$, we define the hint based partitioning of $\mathcal{E}$ with the function

$$\mathrm{hint}_P \;:\; (\mathbb{D} \to \mathbb{D}^\sharp_R) \times \mathbb{E} \to \mathbb{E}.$$

The function hint^ℓ_P considers every token t that points to a leaf node $\mathrm{leaf}[D^\sharp_R]$ and if Definition 4.4.15 holds then replaces every leaf node of $\mathcal{E}$ by a function that points to new leaves as defined by its corresponding partitioning set.

$$\mathrm{hint}^\ell_P(\mathcal{H}, \mathrm{leaf}[D^\sharp_R]) = \begin{cases} \forall\, {}^sD^\sharp_R \in S.\, \mathrm{node}[t^\ell \mapsto \mathrm{leaf}[{}^sD^\sharp_R]] & \text{if } \exists S \in \mathcal{H}(\ell) \text{ such that} \\ & S \text{ is a partitioning set} \\ & \text{of } D^\sharp_R, \\ \mathrm{leaf}[D^\sharp_R] & \text{otherwise,} \end{cases}$$

$$\mathrm{hint}^\ell_P(\mathcal{H}, \mathrm{node}[\phi]) = \mathrm{node}[\lambda(t \in \mathrm{dom}(\phi)).\, \mathrm{hint}^\ell_P(\mathcal{H}, \phi(t))]$$

Definition 4.4.16 allows us to partition traces based on hints. To reduce the number of partitions created hint_P may choose to ignore leaves that already guarantees $\mathfrak{B}$ or $\neg\mathfrak{B}$ as it does not make sense to further partition a trace where we already know a specific behavior either occurred or will occur. So, in practice hint_P will only partition the leaves of an environment where the associated behavior is $\top^{Max}$ to try to refine the partitioning scheme and remove traces where we are certain some behavior are bound to occur from those we do not know what behavior is bound to occur. Note that since hint_P introduces new partitions we can only use it once during the forward semantics of programs. This is because as soon as hint_P creates the partitions it will generate fresh tokens that are no longer shared with $\mathcal{H}$ at any program location ℓ. By Definition 4.4.15, $\mathcal{H}$ can no longer produce valid partitioning sets. This is not a problem as we shall see later that responsibility analysis involves multiple forwad-backward passes which may generate new valid hints at each backward analysis.

Lemma 4.4.2. hint_P *is a complete partitioning of its input.*

Lemma 4.4.2 follows from Definition 4.4.15.

4.4.5 Forward Semantics

The forward semantics of programs is a refinement of a generic abstract interpreter for abstracting the set of reachable states [Cou19] to constrained affine sets. Our goal is to utilize both the set of partition hints and the partitioning functions of Section 4.4.3 to refine the forward semantics. We will construct the function $\overrightarrow{\mathcal{S}}$ of type

$$\overrightarrow{\mathcal{S}} : \mathbb{P}\mathbb{c} \times (\mathbb{L} \to (\mathbb{D} \to \mathbb{D}^{\sharp}_{R})) \times \mathbb{E} \to (\mathbb{L} \to \mathbb{E})$$

by calculational design. Here, $\overrightarrow{\mathcal{S}}$ takes the syntax of programs, a function $\mathcal{H}$ that gives partitioning hints to every program location and an initial environment $\mathcal{E}$. In turn, it outputs a function that maps program labels to abstract environments which approximate the set of reachable states of the concrete trace semantics of the program. Furthermore, the output also provides a sound approximation of sets of concrete traces that lead to behaviors $\mathfrak{B}$ and $\neg\mathfrak{B}$.

Before giving the abstract semantics, we also need to define the ordering relation between any two partitioned transition systems so that we may compute fixpoints for iteration statements.

Definition 4.4.17. The ordering of extended transition systems $P_T \in \mathbb{L} \to \mathbb{E}$ is given by $\preceq_{\mathbb{E}}$ which is an instantiation of Definition 3.5.5 with the abstract domain $\mathbb{D}^{\sharp}_{R}$.

We can now present the partitioned forward semantics of programs. Initially, $\mathcal{H} = \lambda(\ell \in \mathit{labs}[\![\texttt{S}]\!].\, \emptyset$ because there will be no partition hints from the backward analysis. This means that partitions will only occur from the partitioning functions.

The semantics of programs.

- Abstract semantics outside a statement `S`:

$$\ell \notin \mathit{labs}[\![\texttt{S}]\!] \implies \overrightarrow{\mathcal{S}}[\![\texttt{S}]\!]\mathcal{H}\mathcal{E}^{\ell} = \text{leaf}[\bot^{\mathbb{D}^{\sharp}_{R}}]$$

- Abstract semantics of a program `P ::= Sl` :

$$\overrightarrow{\mathcal{S}}[\![\texttt{P}]\!] \triangleq \overrightarrow{\mathcal{S}}[\![\texttt{Sl}]\!]$$

- Abstract semantics of a statement list `Sl ::= Sl' S`:

$$\overrightarrow{\mathcal{S}}[\![\texttt{Sl}]\!]\mathcal{H}\mathcal{E}^{\ell} \triangleq \begin{cases} \overrightarrow{\mathcal{S}}[\![\texttt{Sl'}]\!]\mathcal{H}\mathcal{E}^{\ell} & \text{if } \ell \in \mathit{labs}[\![\texttt{Sl'}]\!] \setminus \{\mathit{at}[\![\texttt{S}]\!]\}, \\ \overrightarrow{\mathcal{S}}[\![\texttt{S}]\!]\mathcal{H}(\overrightarrow{\mathcal{S}}[\![\texttt{Sl'}]\!]\mathcal{H}\mathcal{E}\ \mathit{at}[\![\texttt{S}]\!])^{\ell} & \text{if } \ell \in \mathit{labs}[\![\texttt{S}]\!], \\ \text{leaf}[\bot^{\mathbb{D}^{\sharp}_{R}}] & \text{otherwise.} \end{cases}$$

- Abstract semantics of an empty statement list `Sl ::=` ϵ:

$$\overrightarrow{\mathcal{S}}[\![\texttt{Sl}]\!]\mathcal{H}\mathcal{E}^{\ell} \triangleq \begin{cases} \mathcal{E} & \text{if } \ell = \mathit{at}[\![\texttt{Sl}]\!], \\ \mathrm{leaf}[\bot^{\mathbb{D}^{\sharp}_{R}}] & \text{otherwise.} \end{cases}$$

- Abstract semantics of assignment `S ::= `**`v`**` = [`**`n1`**`,`**`n2`**`] + u ;` with uncertainty:

$$\overrightarrow{\mathcal{S}}[\![\texttt{S}]\!]\mathcal{H}\mathcal{E}^{\ell} \triangleq \begin{cases} \mathcal{E} & \text{if } \ell = \mathit{at}[\![\texttt{S}]\!], \\ \mathrm{hint}^{\ell}_{P}(\mathcal{H}, \mathrm{assn}_{\mathcal{AS}}(\mathbf{v}\texttt{=[}\mathbf{n1}\texttt{,}\mathbf{n2}\texttt{]+u;}, \mathcal{E})) & \text{if } \ell = \mathit{after}[\![\texttt{S}]\!], \\ \mathrm{leaf}[\bot^{\mathbb{D}^{\sharp}_{R}}] & \text{otherwise.} \end{cases}$$

- Abstract semantics of assignment `S ::= `**`v`**` = A ;`:

$$\overrightarrow{\mathcal{S}}[\![\texttt{S}]\!]\mathcal{H}\mathcal{E}^{\ell} \triangleq \begin{cases} \mathcal{E} & \text{if } \ell = \mathit{at}[\![\texttt{S}]\!], \\ \mathrm{hint}^{\ell}_{P}(\mathcal{H}, \mathrm{assn}_{\mathcal{AS}}(\mathbf{v}\texttt{ = A;}, \mathcal{E})) & \text{if } \ell = \mathit{after}[\![\texttt{S}]\!], \\ \mathrm{leaf}[\bot^{\mathbb{D}^{\sharp}_{R}}] & \text{otherwise.} \end{cases}$$

- Abstract semantics of an error assertion `S ::= assert(`**`v`**`,`**`n`**`) ;`:

$$\overrightarrow{\mathcal{S}}[\![\texttt{S}]\!]\mathcal{H}\mathcal{E}^{\ell} \triangleq \begin{cases} \mathcal{E} & \text{if } \ell = \mathit{at}[\![\texttt{S}]\!], \\ \mathrm{e}^{\ell}_{\mathbf{n}}(\mathbf{v}, \mathcal{E}) & \text{if } \ell = \mathit{after}[\![\texttt{S}]\!], \\ \mathrm{leaf}[\bot^{\mathbb{D}^{\sharp}_{R}}] & \text{otherwise.} \end{cases}$$

- Abstract semantics of a conditional `S ::= if (B) St else Sf`:

$$\overrightarrow{\mathcal{S}}[\![\texttt{S}]\!]\mathcal{H}\mathcal{E}^{\ell} \triangleq \begin{cases} \mathcal{E} & \text{if } \ell = \mathit{at}[\![\texttt{S}]\!], \\ \overrightarrow{\mathcal{S}}[\![\texttt{St}]\!]\mathcal{H}\,\mathrm{test}_{\mathcal{B}}(\texttt{B},\mathcal{E})^{\ell} & \text{if } \ell \in \mathit{in}[\![\texttt{St}]\!], \\ \overrightarrow{\mathcal{S}}[\![\texttt{Sf}]\!]\mathcal{H}\,\mathrm{test}_{\bar{\mathcal{B}}}(\texttt{B},\mathcal{E})^{\ell} & \text{if } \ell \in \mathit{in}[\![\texttt{Sf}]\!], \\ \sqcup_{\texttt{ite}}(\mathcal{E}^i,\mathcal{E}^e,\mathcal{E})) & \text{if } \ell \in \mathit{after}[\![\texttt{S}]\!], \\ \mathrm{leaf}[\bot^{\mathbb{D}^{\sharp}_{R}}] & \text{otherwise.} \end{cases}$$

where $\mathcal{E}^i = \overrightarrow{\mathcal{S}}[\![\texttt{St}]\!]\mathcal{H}\,\mathrm{test}_{\mathcal{B}}(\texttt{B},\mathcal{E})^{\ell}$ and $\mathcal{E}^e = \overrightarrow{\mathcal{S}}[\![\texttt{Sf}]\!]\mathcal{H}\,\mathrm{test}_{\bar{\mathcal{B}}}(\texttt{B},\mathcal{E})^{\ell}$.

- Abstract semantics of a while loop `S ::= while` $^{\ell}$ `(B) St`:

$$\overrightarrow{\mathcal{S}}[\![\texttt{while }^{\ell}\texttt{ (B) St}]\!]\mathcal{H}\mathcal{E}^{\ell'} \triangleq \mathrm{lfp}^{\preceq_{\mathbb{E}}}(\overrightarrow{\mathcal{F}}[\![\texttt{while (B) St}]\!]\mathcal{H}\,\mathcal{X})^{\ell'}$$

where $\overrightarrow{\mathcal{F}}[\![\texttt{while }^{\ell}\texttt{ (B) St}]\!]\mathcal{H}\mathcal{E} \in \mathbb{E} \to ((\mathbb{L} \to \mathbb{E}) \to (\mathbb{L} \to \mathbb{E}))$ is defined as

$$\overrightarrow{\mathcal{F}}[\![\texttt{while }^{\ell}\texttt{ (B) St}]\!]\mathcal{H}\mathcal{E}\mathcal{X}^{\ell'} \triangleq$$

$$\begin{cases} \mathrm{join}_P(\mathcal{E}, \overrightarrow{\mathcal{S}}[\![\texttt{Sb}]\!]\mathcal{H}\,\mathrm{test}_{\mathcal{B}}(\texttt{B},\mathcal{X}(^{\ell}))^{\ell}) & \text{if } \ell' = \ell, \\ \overrightarrow{\mathcal{S}}[\![\texttt{Sb}]\!]\mathcal{H}\,\mathrm{test}_{\mathcal{B}}(\texttt{B},\mathcal{X}(^{\ell}))^{\ell'} & \ell' \in \mathit{in}[\![\texttt{Sb}]\!] \setminus \{\ell\}, \\ \sqcup_{\texttt{ite}}(E^e, E^i(^{\ell'}), \mathcal{E}) & \ell' = \mathit{after}[\![\texttt{S}]\!], \\ \mathrm{leaf}[\bot^{\mathbb{D}^{\sharp}_{R}}] & \text{otherwise.} \end{cases}$$

and $E^i = \overrightarrow{\mathcal{S}}[\![\texttt{Sb}]\!]\mathcal{H}\,\mathrm{test}_{\mathcal{B}}(\texttt{B},\mathcal{X}(\ell))$, $E^e = \mathrm{test}_{\bar{\mathcal{B}}}(\texttt{B},\mathcal{X}(\ell))$.

- Abstract semantics of a skip `S ::= ;`:

$$\overrightarrow{\mathcal{S}}[\![\mathtt{S}]\!]\mathcal{H}\mathcal{E}^{\ell} \triangleq \begin{cases} \mathcal{E} & \text{if } \ell = \mathit{at}[\![\mathtt{S}]\!] \vee \ell = \mathit{after}[\![\mathtt{S}]\!] \\ \mathrm{leaf}[\bot^{\mathbb{D}^{\sharp}_{R}}] & \text{otherwise.} \end{cases}$$

- Abstract semantics of a compound statement `S ::= { Sl }`:

$$\overrightarrow{\mathcal{S}}[\![\mathtt{S}]\!] \triangleq \overrightarrow{\mathcal{S}}[\![\mathtt{Sl}]\!]$$

We see that the forward semantics uses our partitioning functions and hints throughout the computation of the semantics to perform partitions. These partitions will either detect errors, such as in the case of the evaluation of if-then-else statements or while loops, or apply the hints after computing an abstract value. Note that the hints of $\mathcal{H}$ are only accessed during assignment statements as stated previously. We do not expect to produce a large amount of partitions using hints. As an example for why this is the case, when entering a loop, $\mathcal{H}$ might be used to partition and refine the set of abstract invariants and these new abstract invariants will most likely be smaller than the partition hints we just used. So, at the next iteration of the loop, it is unlikely that they will be used again. In practice, one could only apply the hints at the first iterate of the loop and then not use them again. Conversely, the use of the join partitioning after branching will cause at least an exponential blow up by a factor of 4. This can be mitigated with partition merges which start happening after the number of partitions reaches a threshold.

The forward semantics presented above is a very general semantics that applies partitioning freely. One can always replace the behaviors of the partitioning functions such that less partitions are created. In fact, experimental evaluation of this liberal abstract interpreter would allow for the design of better abstract interpreters that take into account the trade-off betwen the accuracy of the analysis and the cost of computation. The generic nature of this abstract interpreter is an advantage when it comes to refining its design as parts can be changed locally without effecting the soundness of the rest of the analysis.

Proposition 4.4.3. The forward semantics given by $\overrightarrow{\mathcal{S}}$ is sound with respect to the forward reachable states of the concrete semantics of programs. Furthermore, the forward semantics is sound with respect to the detection of behaviors $\mathfrak{B}$ and $\neg\mathfrak{B}$.

The above proposition can be proved in a few steps. Firstly, we may lift the Galois connection of Section 4.3 to program prefix traces

$$(\wp(E^{*\infty}) \rightarrow (\mathbb{L} \rightarrow \wp(E^{*\infty}), \ddot{\subseteq}) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathbb{D}^\sharp \rightarrow (\mathbb{L} \rightarrow \mathbb{D}^\sharp), \ddot{\sqsubseteq}_{\mathbb{D}^\sharp})$$

where $\ddot{\subseteq}$ indicates point-wise inclusion on functions. Here α and γ are not defined but we know that they exist from [Cou02]. The prefix trace semantics sits at the bottom and the constrained affine forms domain sits at a higher level in the hierarchy of semantics given in [Cou02]. In fact, α and γ can be written as a composition of Galois connections starting from the prefix trace semantics followed by the relational reachability semantics which is then followed by the constrained affine sets semantics. Thus, an abstract interpreter using constrained affine sets without partitioning is sound with respect to the prefix traces of programs. Due to [RM07], the trace partitioning abstract interpreter is also sound with respect to the prefix traces of the program as at any given point only complete partitionings are performed. All that remains to show is that the behaviors we associate with abstract invariants is sound.

We define an inquiry function that maps every trace $\sigma \in E^{*\infty}$ to the strongest maximal trace property in $\mathcal{L}^{Max}$ that can be guaranteed [DC19].

$$\mathbb{I} \in \wp(E^{*\infty}) \rightarrow \wp(\wp(E^{*\infty})) \rightarrow E^{*\infty} \rightarrow \wp(E^{*\infty}) \qquad \text{inquiry function}$$
$$\mathbb{I}(\mathcal{S}^{Max}, \mathcal{L}^{Max}, \sigma) \triangleq \bigcap\{\mathcal{P} \in \mathcal{L}^{Max} \mid \sigma \in \alpha_{\text{Pred}}[\![\mathcal{S}^{Max}]\!](\mathcal{P})\}$$

This definition relies on the prediction abstraction from Section 4.2. The formulation in [DC19] goes on to define a cognizance and observation function but since all floating-point entities have omniscient cognizance they are no longer needed here.

The results of the constrained affine sets analysis is a sound over-approximation of the inquiry function. This is due to the Galois connection above where each prefix trace is soundly abstracted by the domain and the fact that the affine sets domain is sound with respect to the type of program behaviors we are interested in. Firstly, we detect when the error assertion statement fails as soon as the statement is evaluated which means that all prefix traces that are erroneous are detected at that point. The inquiry function may associate an error with some traces before we get to the point of the assertion but since we associate such traces with $\top^{Max}$ this is sound. Similarly, for conditional divergence we attach $\mathfrak{B}$ only at join points meaning all prefix traces at program points before are also marked wiht $\top^{Max}$. A caveat to note here is that we may identify some traces to be $\mathfrak{B}$ earlier than when the affine analysis detects the behavior. This is sound as we assume these invariants are calculated by an under-approximating sufficient condition analyzer. So, we also have the Galois connection

$$(\wp(E^{*\infty}) \to (\mathbb{L} \to \wp(E^{*\infty}), \ddot{\subseteq}) \underset{\alpha'}{\overset{\gamma'}{\longleftrightarrow}} (\mathbb{D}^\sharp_R \to (\mathbb{L} \to \mathbb{D}^\sharp_R), \ddot{\sqsubseteq}_{\mathbb{D}^\sharp_R})$$

where α' can be defined in terms of α and we can infer $\mathcal{P}$ from the constrained affine sets. γ' is simply equal to γ where we drop the property before concretizing. Since our actual partitioning is on this connection, our abstract interpreter is sound.

4.4.6 Widening for Loops

To compute the fixpoint for the semantics of while statements we need to introduce a widening operator that speeds up the fixpoint computation. Otherwise, we are not guaranteed to compute a fixpoint in a finite amount of time as the constrained affine sets domain does not have a finite height. Furthermore, we define the fixpoint in terms of both partitions and the underlying values the partitions define. There may be degenerate cases that introduce partitioning tokens at each iterate of the loop. We must prevent the computation from introducing an arbitrary amount of partitions. Thus, we use a widening operator to reach a post-fixpoint of the semantics of while statements. This is a sound approximation of the least fixpoint of the same function.

From Section 3.3.5 we know that a widening $\nabla_{\mathbb{D}^\sharp}$ exists for constrained affine sets. Recall that this widening works by loop unrolling. Let N be the number of unrollings. Then, $\nabla_{\mathbb{D}^\sharp}$ is not used for N iterates and if a fixpoint is not reached beyond this point we start using $\nabla_{\mathbb{D}^\sharp}$ for every iterate. In this case, the convergence is accelerated by collapsing noise symbols that do not have equal coefficients. We can take inspiration from this widening along with the widening for partitioning used in [RM07] to define a widening over the partitioning.

Firstly, if we start noticing that the loop is continously generating divergence tokens in the loop body then this probably means that we are going to encounter an exponential blow-up in partitioning tokens if we let the partitioning continue. In this case, we stop the generation of partition tokens after some O amount of iterates if this behavior is observed. Secondly, after the N iterates as defined for $\nabla_{\mathbb{D}^\sharp}$ occurs, we should stop the generation of new partitioning tokens to stabilize the partitioning before starting the widening on the underlying abstract domain. Finally, we can define a threshold P amount of partition tokens that we will allow to be created before we start merging tokens and disallowing any new partitions to occur. The last idea can actually be used throughout the whole analysis and not just only loops. At some point it may make sense to stop partition generation and let the analysis finish as doing otherwise could be computationally costly.

Definition 4.4.18. Let $\nabla_{\mathbb{D}^\sharp_R} = (\nabla_P^{N,O,P}, \nabla_{\mathbb{D}^\sharp}^N)$ be a widening for abstract environments $\mathcal{E} \in \mathbb{E}$. $\nabla_{\mathbb{D}^\sharp}^N$ unrolls a loop N times before starting to apply the widening operation $\nabla_{\mathbb{D}^\sharp}$ onto the constrained affine forms. Similarly, $\nabla_P^{N,O,P}$ unrolls a loop N times before disallowing the creation of any new partitions. Furthermore, if for any consecutive O iterates of partitioning, the transition systems P_i as defined by the partitioning leads to $P_i \preceq_\tau P_{i+1} \preceq_\tau \cdots \preceq_\tau P_{i+O}$ then again the partitioning forbids the creation of any new partitioning tokens. Finally, for any iterate that causes the partitioning to exceed P tokens, then the widening disallows the creation of any new partitioning token in the next iterates. To disallow the creation of tokens we stop using all partitioning functions and revert to join_P when computing joins.

Since no known narrowings exist for both the trace partitioning domain and the constrained affine sets domain, we do not define a narrowing for $\mathbb{D}^\sharp_R$.

4.4.7 Running Example

The forward semantics of programs will refine the system from Figure 4.6 using the information obtained by the affine analysis. Recall from Section 2.1 that we have an if-else discontinuity if $-u_x < \hat{r}^r_{[2]}$ and an else-if discontinuity if $-1 \leq \hat{r}^r_{[3]} < -u_y$. This means that the analysis detects the conditions of divergence but, as we noted above, does not discriminate between the cases which in turn limits the precision of inferring the right bound. So, we deal with this issue by explicitly considering each case on its own. This results in the transition system given in Figure 4.7. As noted, the partitioning occurs at the join point, i.e. exit, of the if-then-else statement. This system's transitions are also explicilty labelled with the events that cause the transition. For example, the event $\neg$`(x <= 2 && y >= 1)`$\wedge\neg\rho$`(x <= 2 && y >= 1)` corresponds to the event where the program interpreted in real semantics takes the else branch and the program interpreted in floating-point semantics also takes the same branch. Finally, the right and left bounds are again labelled as `R` and `L` and we will ignore the left bound.

Again, each node of Figure 4.7 is labelled above with the strongest property that can be guaranteed after that point where the properties come from the lattice given in Figure 3.2 and below we have constraints on the noise symbols. The nodes of the transition system are now marked with a label and a corresponding token. We see that besides label [9] no partitioning on the system has been performed and at [9] 4 new tokens were introduced. Node $[9], {}^1t^\ell_{\mathbf{val}}$ corresponds to the transition where the real program takes the if branch and the error symbols are constrained to the case where the floating-point program does not diverge. Node $[9], {}^1t^\ell_{\mathbf{val}}$ is the transition where the real program still takes the if branch but now the error symbols are constrained to the case where the floating-point program diverges towards the

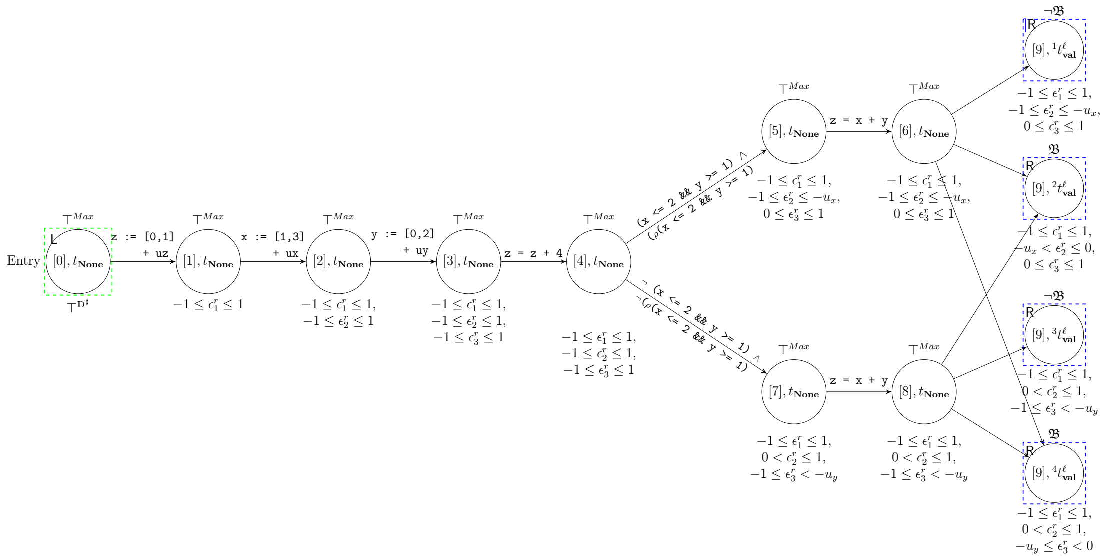


Figure 4.7: Partitioned transition system of Program 2.1.

if branch. Similar reasoning follows with nodes $[9], {}^3t^{\ell}_{\mathbf{val}}$ meaning no divergence and $[9], {}^4t^{\ell}_{\mathbf{val}}$ meaning divergence when the real program takes the else branch. The refinement we have performed to obtain Figure 4.7 is a trace partitioning of Figure 4.6 which we computed with $\overrightarrow{\mathcal{S}}$.

In this example, the trace partitioning of the forward semantics $\overrightarrow{\mathcal{S}}$ inserted partition directives at the join point. We split a 2-way conditional into a 4-way one but in a somewhat intuitive way as the splitting occurs where normally the conditional would collapse into a single path. The computations of the non-divergent cases occur with a single transition such as from [6] to $[9], {}^1t^{\ell}_{\mathbf{val}}$. On the other hand, the computation of the divergent cases must use two transitions as the independent computations of the real and floating-point values can only be joined at the branch exit. For example, the node $[9], {}^2t^{\ell}_{\mathbf{val}}$ corresponds to the case of if divergence and is computed using nodes [6] and [8]. The results of the affine analysis gives the invariants for each branch including for discontinuity. We note that if the analysis at any point returns $\bot$ then we know that a specific partition, or more generally node, is unreachable.

The basis transition system given in Figure 4.6 corresponds closer to the source code of Figure 2.1. However, as we previously noted, this system is much less informative than Figure 4.7 as all nodes' labels are $\top$, meaning the best that can be inferred is that any behavior is possible at all nodes. This is because at nodes [6] and [8] of this system we took a join over all possible behaviors (i.e. divergence vs non-divergence) at these nodes such that this system is unaware of the fact that conditional agreement or conditional divergence occurred. On the other hand, the partitioned system given in Figure 4.7 partitions the original system into more precise behaviors which leads to a more informative system with the splitting of the conditional branch at node [9]. While this does not give us a better right bound for this program we now have sufficient conditions for the behaviors $\mathfrak{B}$ and $\neg\mathfrak{B}$. Looking at Figure 4.7 we see that the right bounds are $[9], {}^{2}t^{\ell}_{\mathbf{val}}$ (which corresponds to program location 9) and $[9], {}^{4}t^{\ell}_{\mathbf{val}}$ (which again corresponds to program location 9). This makes sense, as any point after the divergent case is taken clearly guarantees $\mathfrak{B}$ from that point on. If we had more nodes from this point onwards then clearly our right bound would be more accurate compared to the right bound of Figure 4.6 which is always the very last node of the transition system. In general, the forward trace partitioning of the original system using the constraints derived by the affine analysis leads to a more informative transition system.

The right bounds for Figure 4.7 are not better than the right bounds for Figure 4.6, but we did gain some useful information to discern between the traces that lead to divergence and those that do not. So, our partitioning strategy in the forward pass proved to be useful. We see that a forward affine analysis of a floating-point program leads to a partitioning strategy that may improve the accuracy of the right bound. However, this is still unsatisfactory as we know the real right bound of responsibility is either line 2 or 3. We can improve these right bounds by refining the system further using another partitioning of Figure 4.7. For this we will consider the backward semantics of program Program 2.1 with respect to the transition system Figure 4.7 which we discuss in section Section 4.5.

4.4.8 Remarks

This section has introduced the trace partitioning forward semantics of floating-point programs. The goal of partitioning is to separate prefix traces where behaviors $\mathfrak{B}$ occur from those where $\mathfrak{B}$ does not occur. To that end, partitioning directives are either introduced by the results of the constrained affine analysis or using partition hints. We expect the analysis to almost always improve the results of a standard semantics that abstracts the set of reachable states and in the worst case should not lead to a loss in precision. Furthermore, the generic design of the analysis allows for the definition of different partitioning functions without changing other aspects of the analyzer. In fact, the analysis can be made generic with respect to the underlying abstract domain $\mathbb{D}^\sharp$. However, for responsibility analysis the partitioning functions of Section 4.4.3 is not enough and we must generate partitioning hints using the backward semantics of programs. This will be introduced in the next section.

There are also a few future directions for the forward semantics. Firstly, and most importantly, the representation of complement sets for constrained affine sets should be further explored as partitioning based on partitioning hints will be crucial for the analysis. While the notion of a partitioning set seems to be intuitive, experimental verification will be useful in verifying that this definition works in practice. Besides complementation, further experimentation on different types of floating-point programs including those that could possibly lead to too many partitionings should be considered. This will allow for the refinement of the partitioning strategies described in this section to strategies that balance the trade-off between accuracy and performance. The partitioning strategy here is very expressive but it might actually not be necessary to perform so many partitions. Furthermore, this kind of analysis will be useful in identifying merge points in programs where partitions may be thrown away to reduce the computational cost. Finally, the analysis proposed here only utilizes partitions based on values. However, trace partitioning in its most general form performs partitions based on many factors. Future work should consider the possibility of such partitions and how gains in precision may be made by introducing more sophisticated partitions. Finally, since we are performing a value-based partitioning it may prove useful to take a look at constraint solving techniques [PMR12] to refine the results of our partitioning.

4.5 Backward Semantics of Programs

The forward semantics of Section 4.4 detects errors but does not refine the transition system enough such that we can compute accurate bounds of responsibility. A backward semantics that does not incur any loss in precision would compute both the necessary and sufficient conditions for a given post-condition. However, such an analysis is not computable so we rely on sound under-approximations of sufficient conditions. The reason for the use of under-approximations of sufficient conditions is that the forward analysis will compute a set of invariants that describe traces that may lead to either $\mathfrak{B}$ or $\neg\mathfrak{B}$. So, the backward semantics should compute invariants that describe concrete traces that we know always lead to these behaviors such that another pass of the forward analysis may eliminate traces that are guaranteed to cause a behavior earlier on in the analysis. If instead the analysis were to compute over-approximations then such a partitioning cannot be done as there may be too many so-called phantom concrete traces, i.e. those that do not actually exist in the concrete semantics of the program but are introduced by our over-approximating abstraction, in the concretization of abstract values. Under-approximations allow us to mitigate this issue.

The backward semantic function will take as input each terminal state of a transition system produced by the forward semantics and assume each to be a post-condition that either causes $\mathfrak{B}$ or $\neg\mathfrak{B}$. Then, the analysis will compute the backward semantics of programs for each post-condition independently. The result will be a function from program labels to the invariants the backward semantics associates with each label. This function can then be used as the partitioning hints function $\mathcal{H}$ for another iteration of the forward semantics which should lead to some gains in accuracy due to earlier partitioning of the program.

We formalize the backward semantics of programs. First, we focus on the backward semantics of assignment statements and Boolean expressions. Then, we describe the backward semantics and present the result of the analysis on the transition system of Figure 4.7.

4.5.1 Backward Semantics of Expressions

Before formalizing the backward semantics of programs we must define the backward semantics of assignments and Boolean expressions. Instead of assuming the existence of a general abstract domain $D^{\sharp}_{R}$, the semantics presented here will only work for numeric domains such as constrained affine sets. For the backward semantics of assignments, a first thought may be to define the backward semantics as the inverse of the affine forms as all affine forms are linear, thus invertible. However, this will not work because the interpretation of tests is carried out by either boxes, octagons or polyhedra where there is a loss of information during the approximation on the bounds of the noise symbols. Additionally, the inverse multiplication is affine division which will introduce terms for its linearization and we will have to check for division by 0. It is simpler to follow the schemes as described in Section 3.6.

Definition 4.5.1 (Definition 3.6.7)**.** The backward projection operator for constrained affine sets is defined as

$$\overleftarrow{\mathcal{A}}[\![\texttt{v = [-}\infty,\infty\texttt{];}]\!]D^{\sharp} \triangleq \begin{cases} D^{\sharp} & \text{if } \gamma_{\mathbb{D}^{\sharp}}(\mathcal{A}[\![\texttt{v = [-}\infty,\infty\texttt{];}]\!]D^{\sharp}) = \gamma(D^{\sharp}), \\ \bot^{\mathbb{D}^{\sharp}} & \text{otherwise.} \end{cases}$$

Definition 4.5.2. The under-approximating backward semantics of arithmetic assignments is given by the function

$$\overleftarrow{\mathcal{A}} : \mathbb{S} \times \mathbb{D}^{\sharp} \to \mathbb{D}^{\sharp}$$

that takes as input an assignment statement either of the form `v = [n,n] + u;` or `v = A;` where $\texttt{A} \in \mathbb{A}$ and a constrained affine set $D^{\sharp}$. If the statement is $\texttt{v}_p$ `= A;`

then the result is computed as

$$\overleftarrow{\mathcal{A}}[\![\texttt{v}_p\ \texttt{= A;}]\!]D^\sharp \triangleq drop_p(\overleftarrow{\mathcal{A}}[\![\texttt{v}_p\texttt{= [-}\infty,\infty\texttt{];}]\!] \circ \mathcal{B}[\![\texttt{v}_{p+1}\ \texttt{= A'}]\!]D^\sharp).$$

where `A'` is equivalent to `A` but all occurrences of $\texttt{v}_p$ have been replaced by a fresh variable $\texttt{v}_{p+1}$. If the statement is `v=[n1,n2]+u` then the result is computed as

$$\begin{aligned}\overleftarrow{\mathcal{A}}[\![\texttt{v = [n1,n2] + u;}]\!]D^\sharp \triangleq &(\mathcal{A}[\![\texttt{v = [-}\infty,\infty\texttt{];}]\!] \circ \\ &(\mathcal{A}[\![\texttt{v = v - n1 - u;}]\!] \sqcap_{\mathbb{D}^\sharp} \mathcal{A}[\![\texttt{v = v - n2 - u;}]\!]) \circ \\ &\mathcal{B}[\![\texttt{v >= n1 and v <= n2}]\!])D^\sharp.\end{aligned}$$

The first definition is derived from Definition 3.6.8 and the second is derived from Definition 3.6.9. Both are sound under-approximations as constrained affine sets describe closed convex sets.

Note that in the above definition `v = v - n1 - u;` subtracts `n1` from `v`'s real value and subtracts `n1 + u` from `v`'s floating-point value. This also holds for `v = v - n2 - u;`. Furthermore, the test `v>=a and v<=b` is correct as under floating-point interpretation it will be evaluated as `v>=a+u and v<=b+u` as long as we negate the error symbol `u` before interpreting the test. So, we get a backward semantics that does not rely on the fallback projection of assignments.

For the backward semantics of tests there are two choices. Notice that test interpretation with constrained affine sets refines the values of the noise symbols to ranges smaller than $[-1, 1]$. This means that the maximum area any affine form can describe is if its symbols are all set to $[-1, 1]$. Thus, the first option for backward interpretation of tests is to simply expand all noise symbols that were involved in the test to $[-1, 1]$ as this is equivalent to removing the refinement of the test from the set of constraints. After removing the constraints there should be a check on whether the interpretation of the test under these new constraints implies the post-condition using $\sqsubseteq_{\mathbb{D}^\sharp}$. However, for the responsibility analysis of floating-point programs, the constraints on the noise symbols are useful as they can be used to limit the abstract values to the cases where test discontinuity occurs. Thus, as a second option we may choose to use the fallback test from Section 3.6 instead. Without evaluating the two strategies empirically it is hard to say which one leads to more accurate results but the second strategy seems to be intuitively better. So, we choose a slight variant of this fallback test and note that we require empirical verification of this choice.

Definition 4.5.3. The under-approximating backward semantics of arithmetic assignments is given by the function

$$\overleftarrow{\mathcal{B}} \;:\; \mathbb{B} \times \mathbb{D}^\sharp \rightarrow \mathbb{D}^\sharp$$

which takes a Boolean expression and outputs the meet of the given abstract element with its interpretation under the forward semantics of the Boolean expression.

$$\overleftarrow{\mathcal{B}}[\![\mathtt{B}]\!]D^\sharp \triangleq \mathcal{B}[\![\mathtt{B}]\!]D^\sharp \sqcap_{\mathbb{D}^\sharp} D^\sharp.$$

Definition 4.5.3 will fallback to the identity function if the test interpretation produces an invariant larger than its input. In some cases, it might be that the post-condition came exclusively from one branch so this definition allows us to remove the information of that branch in the backward direction. With these sound backward operators we can now define the backward semantics of programs. Before we do so, we lift these definitions to the domain $\mathbb{D}^\sharp_R$.

Definition 4.5.4. The lifting of $\overleftarrow{\mathcal{A}}$ to $\mathbb{D}^\sharp_R$ is defined as

$$\overleftarrow{\mathcal{A}}[\![\texttt{S = ...}]\!](D^\sharp, \mathcal{P}) = \begin{cases} (\overleftarrow{\mathcal{A}}[\![\texttt{S = ...}]\!]D^\sharp, \mathcal{P}) & \text{if } \overleftarrow{\mathcal{A}}[\![\texttt{S = ...}]\!]D^\sharp \sqsubseteq_{\mathbb{D}^\sharp} D^\sharp. \\ (\overleftarrow{\mathcal{A}}[\![\texttt{S = ...}]\!]D^\sharp, \top^{Max}) & \text{otherwise.} \end{cases}$$

The lifting of $\overleftarrow{\mathcal{B}}$ is much simpler and is defined as

$$\overleftarrow{\mathcal{B}}[\![\texttt{B}]\!](D^\sharp, \mathcal{P}) = (\overleftarrow{\mathcal{B}}[\![\texttt{B}]\!]D^\sharp, \mathcal{P})$$

Definition 4.5.4 ensures the soundness of the backward analysis. Recall that the abstract domain $\mathbb{D}^\sharp_R$ is a pair where the first element describes an abstract invariant while the second describes a property that can be inferred from that invariant. Our post-condition assigns to an invariant a property for which we know with certainty that it occurs and this is an over-approximation of the inquiry function $\mathbb{I}$. But, when we compute the backward semantics, the set of values described by the abstract invariants may start getting larger due to projections. In this case, this means that we can no longer guarantee that the behavior in the post-condition holds so we must revert to $\top^{Max}$, i.e. anything can happen. With this definition, the backward semantics is now sound with respect to both constrained affine sets and the system behaviors of interest. The reduced product between the two domains still holds where if one reverts to bottom then the other must become bottom as well.

4.5.2 Backward Semantics of Programs

The definition of the backward semantics is straightforward as it is identical to the generic backward abstract interpreters given in [Min14] and [Cou19]. We simply perform the backward analysis on each leaf of the partitioning tree that is computed for the very end of the program by the forward semantics. Tree nodes that guarantee the property $\top^{Max}$ may be marked as $\neg\mathfrak{B}$ no errors have been detected and the analysis has reached the end of the program. However, care must be taken to not mark all such nodes as $\neg\mathfrak{B}$ as some might be the result of partition merging which has collapses the divergent and non-divergent invariants together. For example, the widening $\nabla_{\mathbb{D}^\sharp_R}$ may start merging partitions which in turn will lose information regarding which abstract invariants are associated with $\mathfrak{B}$. So, any leaf node that has a parent which is a partitioning directive of the form $t^m_{\textbf{None}}$ is not marked as $\neg\mathfrak{B}$ and the rest can be safely marked as such. This processing leaves us with the set of leaves which we know either guarantee $\mathfrak{B}$ if they were marked so or $\neg\mathfrak{B}$ if they are marked as $\top^{Max}$. Note that the backward semantics will still be computed for all post-conditions so that the notion of partitioning set may be utilized.

The function $\overleftarrow{\mathbb{S}}$, defined as

$$\overleftarrow{\mathbb{S}} \;:\; \mathbb{Pc} \times (\mathbb{L} \to \mathbb{D}^\sharp_R) \times (\mathbb{L} \to \mathbb{D}^\sharp_R),$$

takes as input the syntax of programs, a mapping from program labels to elements of the responsibility abstract domain $\mathcal{R}$ and produces a new mapping from labels to abstract elements. The first function allows us to define post-conditions or assumptions at multiple program points. In our case, we will only be attaching a post-condition to the last program point and every other label is mapped to $\bot_{\mathbb{D}^\sharp_R}$.

For each post-condition $D^\sharp$ the following is computed.

The backward semantics of programs.

- Abstract semantics of a program `P ::= Sl` :

$$\overleftarrow{\mathcal{S}}\llbracket \texttt{P} \rrbracket \triangleq \overrightarrow{\mathcal{S}}\llbracket \texttt{Sl} \rrbracket$$

- Abstract semantics of a statement list `Sl ::= Sl' S`:

$$\begin{aligned}\overleftarrow{\mathcal{S}}\llbracket \texttt{Sl} \rrbracket \mathcal{R} \triangleq \overleftarrow{\mathcal{S}}\llbracket \texttt{Sl'} \rrbracket (\lambda(\ell \in \mathit{labs}\llbracket \texttt{Sl'} \rrbracket).\\ \textbf{if } \ell = \mathit{after}\llbracket \texttt{Sl'} \rrbracket \textbf{ then } \mathcal{R}(\ell) \sqcap_{\mathbb{D}^\sharp_R} \overleftarrow{\mathcal{S}}\llbracket \texttt{S} \rrbracket \mathcal{R}\\ \textbf{else } \mathcal{R}(\ell))\end{aligned}$$

- Abstract semantics of an empty statement list `Sl ::=` ϵ:

$$\overleftarrow{\mathcal{S}}\llbracket \texttt{Sl} \rrbracket \mathcal{R} \triangleq \mathcal{R}(\mathit{at}\llbracket \texttt{Sl} \rrbracket)$$

- Abstract semantics of assignment `S ::=` **`v = [n1,n2] + u ;`** with uncertainty:

$$\overleftarrow{\mathcal{S}}\llbracket \texttt{S} \rrbracket \mathcal{R} \triangleq \mathcal{R}(\mathit{at}\llbracket \texttt{S} \rrbracket) \sqcap_{\mathbb{D}^\sharp_R} \overleftarrow{A}\llbracket \texttt{\textbf{v} = [\textbf{n1},\textbf{n2}] + u ;} \rrbracket \mathcal{R}(\mathit{after}\llbracket \texttt{S} \rrbracket)$$

- Abstract semantics of assignment `S ::=` $\mathbf{v}$ `= A ;`:

$$\overleftarrow{\mathcal{S}}[\![\mathtt{S}]\!]\mathcal{R} \triangleq \mathcal{R}(at[\![\mathtt{S}]\!]) \sqcap_{\mathbb{D}^\sharp_R} \overleftarrow{A}[\![\mathbf{v}\ \mathtt{=\ A;}]\!]\mathcal{R}(\mathit{after}[\![\mathtt{S}]\!])$$

- Abstract semantics of an error assertion `S ::= assert(`$\mathbf{v}$`,`$\mathbf{n}$`)`;:

$$\overleftarrow{\mathcal{S}}[\![\mathtt{S}]\!]\mathcal{R} \triangleq \mathcal{R}(at[\![\mathtt{S}]\!]) \sqcap_{\mathbb{D}^\sharp_R} \mathcal{R}(\mathit{after}[\![\mathtt{S}]\!])$$

- Abstract semantics of a conditional `S ::= if (B) St else Sf`:

$$\overleftarrow{\mathcal{S}}[\![\mathtt{S}]\!]\mathcal{R} \triangleq \mathcal{R}(at[\![\mathtt{S}]\!]) \sqcap_{\mathbb{D}^\sharp_R} \overleftarrow{\mathcal{B}}[\![\mathtt{B}]\!](\overleftarrow{\mathcal{S}}[\![\mathtt{St}]\!]\mathcal{R}) \sqcap_{\mathbb{D}^\sharp_R} \overleftarrow{\mathcal{B}}[\![\neg\mathtt{B}]\!](\overleftarrow{\mathcal{S}}[\![\mathtt{Sf}]\!]\mathcal{R})$$

- Abstract semantics of a while loop `S ::= while` $^\ell$ `(B) St`:

$$\overleftarrow{\mathcal{S}}[\![\mathtt{while}\ ^\ell\ \mathtt{(B)\ St}]\!]\mathcal{R} \triangleq \mathrm{gfp}^{\sqsubseteq_{\mathbb{D}^\sharp_R}}(\overleftarrow{\mathcal{F}}[\![\mathtt{while\ (B)\ St}]\!]\mathcal{R})$$

  where $\overleftarrow{\mathcal{F}}[\![\mathtt{while}\ ^\ell\ \mathtt{(B)\ St}]\!]\mathcal{R} \in \mathbb{P}\mathbb{C} \times (\mathbb{L} \to \mathbb{D}^\sharp_R) \to (\mathbb{L} \to \mathbb{D}^\sharp_R)$ is defined as

$$\overleftarrow{\mathcal{F}}[\![\mathtt{while}\ ^\ell\ \mathtt{(B)\ St}]\!]\mathcal{R}\,\mathcal{X}^{\ell'} \triangleq \mathcal{R}(\ell) \sqcap_{\mathbb{D}^\sharp_R} \overleftarrow{\mathcal{B}}[\![\mathtt{B}]\!](\mathcal{R}(\mathit{after}[\![\mathtt{S}]\!]))\sqcap_{\mathbb{D}^\sharp_R}$$
$$\overleftarrow{\mathcal{B}}[\![\mathtt{B}]\!](\overleftarrow{\mathcal{S}}[\![St]\!](\lambda(\ell' \in \mathit{labs}[\![\mathtt{S}]\!]).\ \mathbf{if}\ \ell' = \ell\ \mathbf{then}\ \mathcal{R}(\ell) \sqcap_{\mathbb{D}^\sharp_R} \mathcal{X}\ \mathbf{else}\ \mathcal{R}(\ell')))$$

- Abstract semantics of a skip `S ::= ;`:

$$\overleftarrow{\mathcal{S}}[\![\mathtt{S}]\!]\mathcal{R} \triangleq \mathcal{R}(at[\![\mathtt{S}]\!]) \sqcap_{\mathbb{D}^\sharp_R} \mathcal{R}(\mathit{after}[\![\mathtt{S}]\!])$$

- Abstract semantics of a compound statement `S ::= { Sl }`:

$$\overleftarrow{\mathbb{S}}[\![\texttt{S}]\!] \triangleq \overleftarrow{\mathbb{S}}[\![\texttt{Sl}]\!]$$

Since there does not exist any lower widenings for constrained affine sets we accelerate loop convergence for greatest fixpoint computations by performing unrollings for N iterations and afterwards returning $\bot_{\mathbb{D}^\sharp_R}$. Finally, to produce the partitioning hint function $\mathcal{H}$ we map each label of the program to the abstract tree environment we used to generate the post-conditions and replace each leaf by the invariants for each corresponding location.

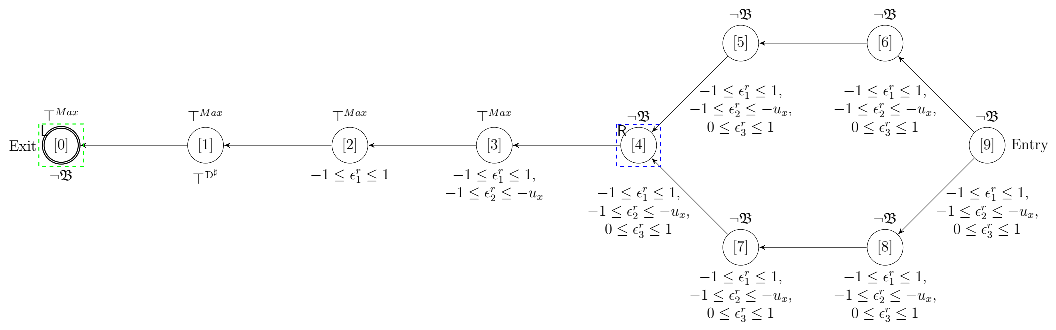
Proposition 4.5.1. The backward semantics $\overleftarrow{\mathbb{S}}$ is sound.

There is not much to explain here as the soundness follows from [Min14] and [Cou19]. The only key issue is how to deal with program properties when going backwards and this was addressed with Definition 4.5.4.

4.5.3 Running Example

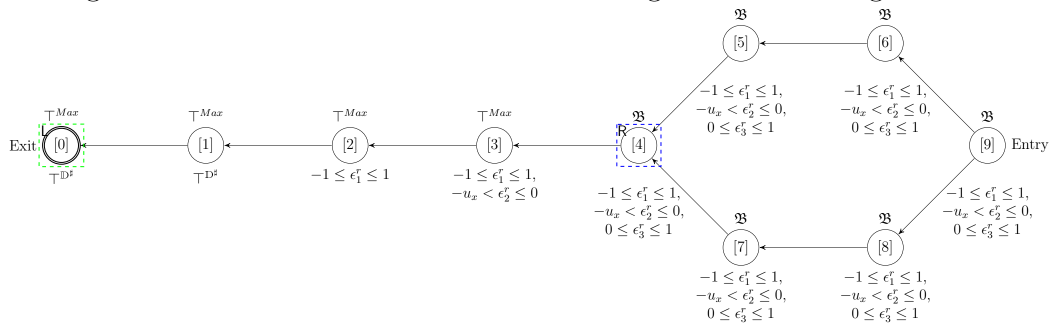
Doing a backward analysis on Figure 4.7 yields the transition systems given in Figures 4.8 through 4.11. We have 4 systems now as we performed the analysis for 4 post-conditions; one for each of the accepting states of Figure 4.7. Each node of each system is again labelled below with constraints on the noise symbols and above with the strongest guaranteed behavior.

For the transition from [6] to [5] we know that there is an assignment of the form `z = x + y` or `z = x - y`. The inverse of linear affine forms is exact for these



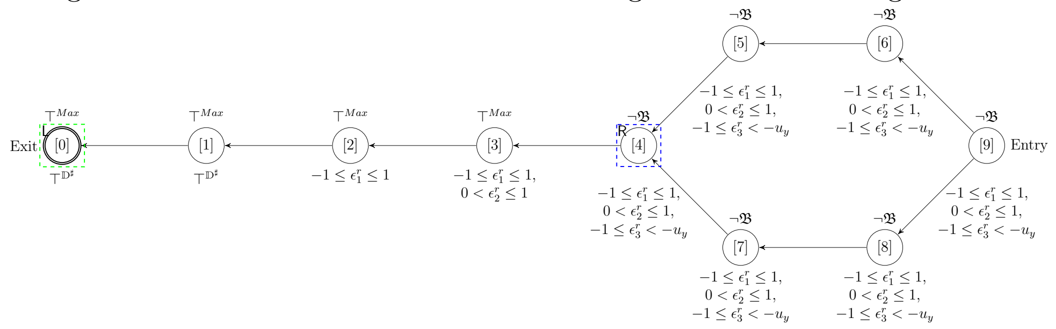
$\neg\mathfrak{B}$ [5]
$\neg\mathfrak{B}$ [6]
$-1 \le \epsilon_1^r \le 1$, $-1 \le \epsilon_2^r \le -u_x$, $0 \le \epsilon_3^r \le 1$
$-1 \le \epsilon_1^r \le 1$, $-1 \le \epsilon_2^r \le -u_x$, $0 \le \epsilon_3^r \le 1$
Exit [0] $\top^{Max}$ L $\neg\mathfrak{B}$
[1] $\top^{Max}$ $\top^{\mathbb{D}^\sharp}$
[2] $\top^{Max}$ $-1 \le \epsilon_1^r \le 1$
[3] $\top^{Max}$ $-1 \le \epsilon_1^r \le 1$, $-1 \le \epsilon_2^r \le -u_x$
[4] $\neg\mathfrak{B}$ R $-1 \le \epsilon_1^r \le 1$, $-1 \le \epsilon_2^r \le -u_x$, $0 \le \epsilon_3^r \le 1$
[9] Entry $\neg\mathfrak{B}$ $-1 \le \epsilon_1^r \le 1$, $-1 \le \epsilon_2^r \le -u_x$, $0 \le \epsilon_3^r \le 1$
$\neg\mathfrak{B}$ [7] $-1 \le \epsilon_1^r \le 1$, $-1 \le \epsilon_2^r \le -u_x$, $0 \le \epsilon_3^r \le 1$
$\neg\mathfrak{B}$ [8] $-1 \le \epsilon_1^r \le 1$, $-1 \le \epsilon_2^r \le -u_x$, $0 \le \epsilon_3^r \le 1$

Figure 4.8: The backward semantics of Program 2.1 for if agreement.



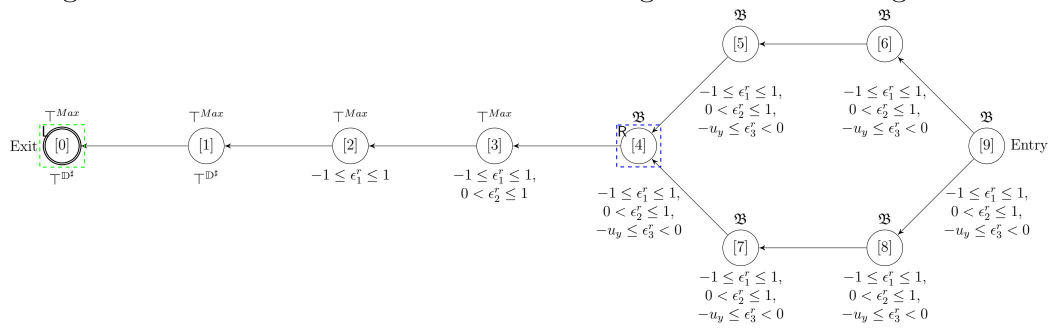
$\mathfrak{B}$ [5]
$\mathfrak{B}$ [6]
$-1 \le \epsilon_1^r \le 1$, $-u_x < \epsilon_2^r \le 0$, $0 \le \epsilon_3^r \le 1$
$-1 \le \epsilon_1^r \le 1$, $-u_x < \epsilon_2^r \le 0$, $0 \le \epsilon_3^r \le 1$
Exit [0] $\top^{Max}$ L $\top^{\mathbb{D}^\sharp}$
[1] $\top^{Max}$ $\top^{\mathbb{D}^\sharp}$
[2] $\top^{Max}$ $-1 \le \epsilon_1^r \le 1$
[3] $\top^{Max}$ $-1 \le \epsilon_1^r \le 1$, $-u_x < \epsilon_2^r \le 0$
[4] $\mathfrak{B}$ R $-1 \le \epsilon_1^r \le 1$, $-u_x < \epsilon_2^r \le 0$, $0 \le \epsilon_3^r \le 1$
[9] Entry $\mathfrak{B}$ $-1 \le \epsilon_1^r \le 1$, $-u_x < \epsilon_2^r \le 0$, $0 \le \epsilon_3^r \le 1$
$\mathfrak{B}$ [7] $-1 \le \epsilon_1^r \le 1$, $-u_x < \epsilon_2^r \le 0$, $0 \le \epsilon_3^r \le 1$
$\mathfrak{B}$ [8] $-1 \le \epsilon_1^r \le 1$, $-u_x < \epsilon_2^r \le 0$, $0 \le \epsilon_3^r \le 1$

Figure 4.9: The backward semantics of Program 2.1 for if disagreement.



$\neg\mathfrak{B}$ [5]
$\neg\mathfrak{B}$ [6]
$-1 \le \epsilon_1^r \le 1$, $0 < \epsilon_2^r \le 1$, $-1 \le \epsilon_3^r < -u_y$
$-1 \le \epsilon_1^r \le 1$, $0 < \epsilon_2^r \le 1$, $-1 \le \epsilon_3^r < -u_y$
Exit [0] $\top^{Max}$ L $\top^{\mathbb{D}^\sharp}$
[1] $\top^{Max}$ $\top^{\mathbb{D}^\sharp}$
[2] $\top^{Max}$ $-1 \le \epsilon_1^r \le 1$
[3] $\top^{Max}$ $-1 \le \epsilon_1^r \le 1$, $0 < \epsilon_2^r \le 1$
[4] $\neg\mathfrak{B}$ R $-1 \le \epsilon_1^r \le 1$, $0 < \epsilon_2^r \le 1$, $-1 \le \epsilon_3^r < -u_y$
[9] Entry $\neg\mathfrak{B}$ $-1 \le \epsilon_1^r \le 1$, $0 < \epsilon_2^r \le 1$, $-1 \le \epsilon_3^r < -u_y$
$\neg\mathfrak{B}$ [7] $-1 \le \epsilon_1^r \le 1$, $0 < \epsilon_2^r \le 1$, $-1 \le \epsilon_3^r < -u_y$
$\neg\mathfrak{B}$ [8] $-1 \le \epsilon_1^r \le 1$, $0 < \epsilon_2^r \le 1$, $-1 \le \epsilon_3^r < -u_y$

Figure 4.10: The backward semantics of Program 2.1 for else agreement.



$\mathfrak{B}$ [5]
$\mathfrak{B}$ [6]
$-1 \le \epsilon_1^r \le 1$, $0 < \epsilon_2^r \le 1$, $-u_y \le \epsilon_3^r < 0$
$-1 \le \epsilon_1^r \le 1$, $0 < \epsilon_2^r \le 1$, $-u_y \le \epsilon_3^r < 0$
Exit [0] $\top^{Max}$ L $\top^{\mathbb{D}^\sharp}$
[1] $\top^{Max}$ $\top^{\mathbb{D}^\sharp}$
[2] $\top^{Max}$ $-1 \le \epsilon_1^r \le 1$
[3] $\top^{Max}$ $-1 \le \epsilon_1^r \le 1$, $0 < \epsilon_2^r \le 1$
[4] $\mathfrak{B}$ R $-1 \le \epsilon_1^r \le 1$, $0 < \epsilon_2^r \le 1$, $-u_y \le \epsilon_3^r < 0$
[9] Entry $\mathfrak{B}$ $-1 \le \epsilon_1^r \le 1$, $0 < \epsilon_2^r \le 1$, $-u_y \le \epsilon_3^r < 0$
$\mathfrak{B}$ [7] $-1 \le \epsilon_1^r \le 1$, $0 < \epsilon_2^r \le 1$, $-u_y \le \epsilon_3^r < 0$
$\mathfrak{B}$ [8] $-1 \le \epsilon_1^r \le 1$, $0 < \epsilon_2^r \le 1$, $-u_y \le \epsilon_3^r < 0$

Figure 4.11: The backward semantics of Program 2.1 for else disagreement.

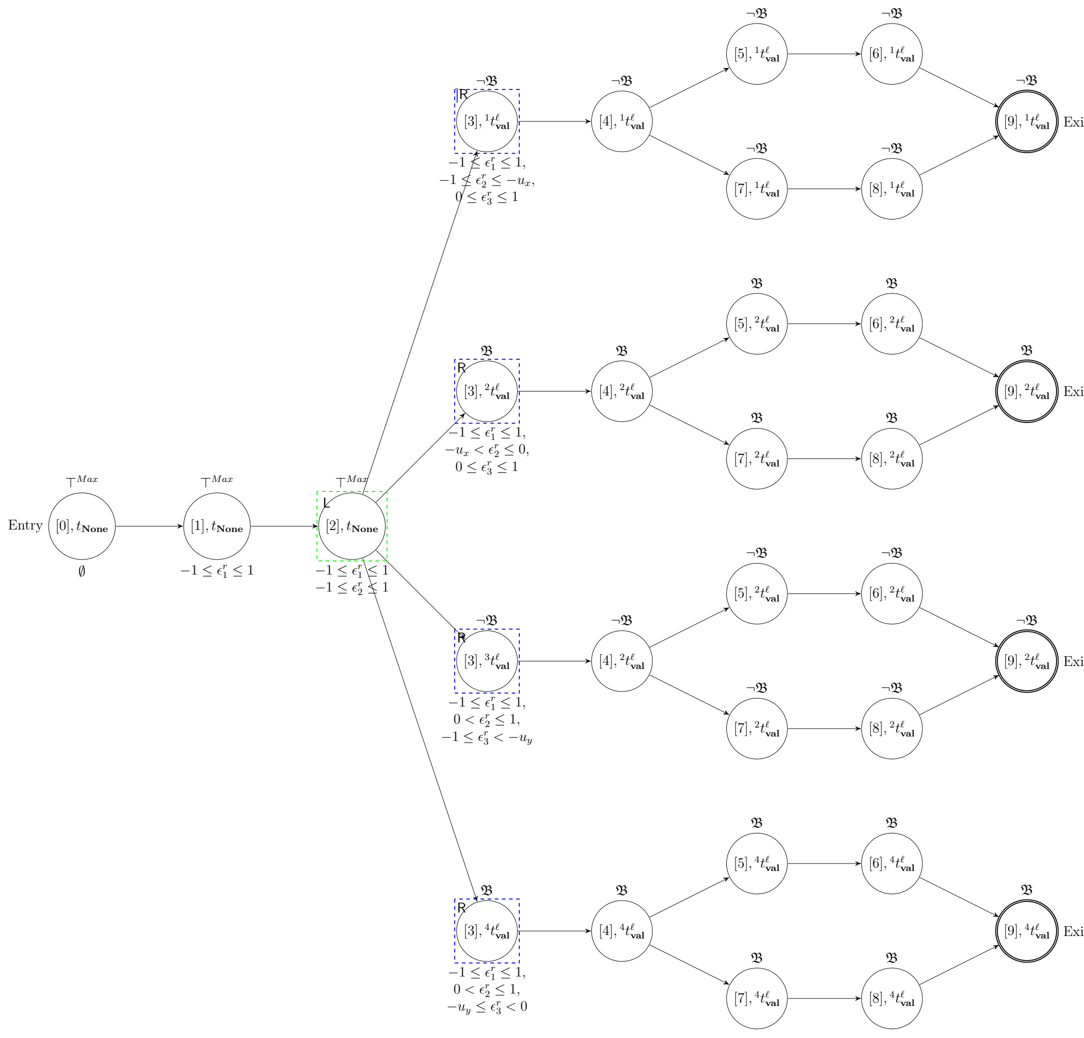
assignments since we only use linear operations so we can actually compute the old value algebraically. This is why the noise symbol for z, ϵ_r^1, does not change in terms of how it is constrained. Thus, the results of these backward analyses are exact assuming we lose no precision in terms of the noise symbols. The meet at node 4 is trivial as both of its operands will be the same constrained affine set.

We see that after node [4] we can no longer ensure that the invariants we compute guarantee either $\mathfrak{B}$ or $\neg\mathfrak{B}$. This is because line 3 of Program 2.1 is an assignment to the variable y which we project to $[-\infty, \infty]$. However, we still do get an improvement of the right bound for all transition systems. In Figure 4.7 we had a right bound node [9] while the right bound for these systems are node [4]. We know that the actual responsible entities are either nodes [2] or [3] so the new right bound of [4] is a definite improvement. There are no improvements in the left bound as it remains at [0].

Figure 4.8 suggest to us that at node [3] we perform a partitioning. The set of noise symbols from the forward semantics at line 3 of Program 2.1 is $\Phi_r^3 = \{-1 \leq \epsilon_1^r \leq 1, -1 \leq \epsilon_2^r \leq 1, 1 \leq \epsilon_3^r \leq 1\}$. Looking at node [4] we see that the partitioning of Φ_r^3 into $\Phi_r^{[4]} = \{-1 \leq \epsilon_1^r \leq 1, -1 \leq \epsilon_2^r \leq -u_x, 0 \leq \epsilon_3^r \leq 1\}$ and $\Phi_r^3 \setminus \Phi_r^{[4]}$ should let us split the cases where we guarantee $\mathfrak{B}$ occurs from the cases where we only know $\top^{Max}$ is guaranteed. This complement could be represented as disjunctive completion but notice that our partitioning set definition works here. Let's consider Figures 4.9 through 4.11. The union of these constraints and the constraint at node [4] of Figure 4.8 are exactly equal to Φ_r^3. Furthermore, if we split the cases into Figure 4.8 and Figure 4.9 we see that the range of ϵ_3^r is fixed and for Figure 4.10 and Figure 4.11 the range of ϵ_2^r is fixed. Thus, the intersection of Figure 4.8 and Figure 4.9 produces bottom for variable ϵ_2^r and the intersection of Figure 4.10 and Figure 4.11 produces bottom for variable ϵ_3^r. Furthermore, the other pair-wise complements will also lead to bottom in either ϵ_3^r or ϵ_2^r. This implies that in the forward semantics we can split node [3] perfectly using the definition of partitioning set such that we know now that there are conditions at line 3 of Program 2.1 that guarantee either $\mathfrak{B}$ or $\neg\mathfrak{B}$ occurs. Finally, the forward analysis need not perform any further partitions after this point as there is no longer any partitions in the abstract environment that are associated with $\top^{Max}$. Thus, the hints produced by our backward semantics has managed to push the right bound of responsibility to [3] which is exactly where some responsible entities for $\mathfrak{B}$ are in the concrete semantics.

Carrying out another forward analysis using the hints from the backward analysis produces a new transition system which is given in Figure 4.12. The transition event labels are again omitted and the above labels describe the strongest possible behavior. The below labels are omitted beyond nodes $[3], t$ as they are easy to infer. The labels L and R are for the left and right bounds for the behaviors $\mathfrak{B}$ or $\neg\mathfrak{B}$ which may be inferred by checking the strongest guaranteed property above node $[3], t$.

We see that similar to Figure 4.7 we introduce partitions for the 4 possible cases after an if-statement (if agreement, if disagreement, else agreement and else disagreement). However, due to the hints produced by the backward analysis we now perform this partitioning at node [3] and not node [9]. This vastly improves the precision of the analysis as now all right bounds of responsibility are at nodes of the form $[3], t$. This is a good result. We may actually push the left bound of responsibility to node [2] which gives us the most accurate responsibility bounds for Program 2.1 though we do not need outline how to do this here (see Section 4.6).



Entry
$[0], t_{\mathbf{None}}$
$\top^{Max}$
$\emptyset$
$[1], t_{\mathbf{None}}$
$-1 \leq \epsilon_1^r \leq 1$
$[2], t_{\mathbf{None}}$
L
$-1 \leq \epsilon_1^r \leq 1$
$-1 \leq \epsilon_2^r \leq 1$
R
$[3], {}^1t^\ell_{\mathbf{val}}$
$\neg\mathfrak{B}$
$-1 \leq \epsilon_1^r \leq 1,$
$-1 \leq \epsilon_2^r \leq -u_x,$
$0 \leq \epsilon_3^r \leq 1$
$[3], {}^2t^\ell_{\mathbf{val}}$
$\mathfrak{B}$
$-1 \leq \epsilon_1^r \leq 1,$
$-u_x < \epsilon_2^r \leq 0,$
$0 \leq \epsilon_3^r \leq 1$
$[3], {}^3t^\ell_{\mathbf{val}}$
$-1 \leq \epsilon_1^r \leq 1,$
$0 < \epsilon_2^r \leq 1,$
$-1 \leq \epsilon_3^r < -u_y$
$[3], {}^4t^\ell_{\mathbf{val}}$
$-1 \leq \epsilon_1^r \leq 1,$
$0 < \epsilon_2^r \leq 1,$
$-u_y \leq \epsilon_3^r < 0$
$[4], {}^1t^\ell_{\mathbf{val}}$
$[5], {}^1t^\ell_{\mathbf{val}}$
$[6], {}^1t^\ell_{\mathbf{val}}$
$[7], {}^1t^\ell_{\mathbf{val}}$
$[8], {}^1t^\ell_{\mathbf{val}}$
$[9], {}^1t^\ell_{\mathbf{val}}$
Exit
$[4], {}^2t^\ell_{\mathbf{val}}$
$[5], {}^2t^\ell_{\mathbf{val}}$
$[6], {}^2t^\ell_{\mathbf{val}}$
$[7], {}^2t^\ell_{\mathbf{val}}$
$[8], {}^2t^\ell_{\mathbf{val}}$
$[9], {}^2t^\ell_{\mathbf{val}}$
$[4], {}^4t^\ell_{\mathbf{val}}$
$[5], {}^4t^\ell_{\mathbf{val}}$
$[6], {}^4t^\ell_{\mathbf{val}}$
$[7], {}^4t^\ell_{\mathbf{val}}$
$[8], {}^4t^\ell_{\mathbf{val}}$
$[9], {}^4t^\ell_{\mathbf{val}}$

Figure 4.12: Forward partitioning of Figure 4.7 using hints from backward analysis.

The forward-backward analysis will not be able to refine Figure 4.12 any further due to the backward assignment at node [3].

We could consider splitting the case of backward assignment into those that assign to ranges of the form `v = [a,b] + u` and those that assign to ranges of the form `v = expr`. For the latter we keep our current backward operator and for the former we use the identity function which will be sound. However, this will have the unintended conseqeuence of pushing the partitioning hint all the way to node [0] and we know that there does not exist any transition from node [0] that guarantees $\mathfrak{B}$ unless we fix the value of variables a priori. So this type of backward operator will not work as we do not let variables' choose values which goes against the idea of responsibility.

4.5.4 Remarks

In this section we have defined the hint producing backward semantics of floating-point programs. The goal of this analysis is to provide hints to the partitioning forward semantics so that its result may be refined. This refinement should cause the forward analysis to perform a better partitioning such that we may obtain more accurate left and right bounds of responsibility. From [RM07] we know that trace partitioning will never produce less accurate information. Thus, in the worst case the results of this analysis will not help with forward partitioning but we expect this to not occur often. The overall idea is to iterate back and forth between the forward semantics and the backward semantics where each will improve the results of the other. We stop when no new refinements can be produced. This process will be formalized in the next section.

One limitation of our backward semantics is that assignments to intervals will always project the values to infinity. There are two ways to fix this issue. The first is to only allow a single uncertain assignment to a variable. All further assignments will either be arithmetic assignments and for re-assignments to that variable the user should simply declare a new variable. This is somewhat restrictive so there is also another solution. We can simply introduce new temporary variables before the analysis begins that keep track of the variable's old value before a new assignment occurs. In this case, this tracking should happen after the first uncertain assignment occurs so that the last uncertain assignmnet we encounter during the backward pass may project the variable back to infinity safely. We haven't outlined this here but a realistic implementation of this domain should probably use either of these proposals. Either one is simple and does not require much changing to the semantics as they can either be implemented using synctactic addition of variables or checking the program for re-declarations before starting the analysis.

For future work we are considering whether it is possible to design backward partitioning semantics. We do not want to partition the system any further but it might be useful to design a way such that it is easy for Definition 4.4.15 to identify candidate partitions quickly and Definition 4.4.16 may use its hint partitioning more than once in a forward pass. Another direction is the design of non-trivial backward operators for Boolean expressions. We know that our backward assignment operators are exact because affine operations are always invertible which we can express in programs by re-writing expressions. However, we could still explore alternatives that may lead to an increase in the precision of the analysis. For Boolean expressions we theorized that any non-trivial operator should not grow the constraints on the affine symbols by too much as this may lead to a loss of precision. Referring back to the systems given in Figures 4.8 through 4.11, consider expanding all noise symbols to larger ranges. Assuming such an operation is sound, we will encounter two issues. The first is that the invariants at node [4] of each system no longer perfectly partition the set Φ_r^3 as these invariants will no longer form a partitioning set. So, we will only be able to partition according to a subset of the invariants instead of all of them and as of now there is no systematic way to go about this. Secondly, the right bounds of responsibility will have to be pushed to node [5] and [7] as increasing the ranges on the noise symbols leads to larger concretizations compared to the concretization of the post-condition. Clearly, not performing such an operation and sticking to the trivial backward function produces better results. Finally, we note that experimental evaluation of the hints generated by the backward semantics may allow us to evaluate whether a complementation based approach works better compared to partitioning sets. For example, in the preceeding section the partitioning set worked perfectly but this may not always be the case.

4.6 Responsibility Analysis

From Section 4.4 and Section 4.5 we know that we wish to go back and forth between the two semantics and refine the results of each one. This process is a forward-backward static analysis with iterated intermediate reduction [Cou19].

Definition 4.6.1. The abstract responsibility analysis of floating-point programs is defined by the following iterates.

$$
\begin{aligned}
X^0 &\triangleq \mathcal{E} \\
Y^0 &\triangleq \mathcal{H} \\
X^1 &\triangleq \overrightarrow{\mathcal{S}}[\![\mathsf{S}]\!]Y^0\mathcal{E} \\
Y^1 &\triangleq \overleftarrow{\mathcal{S}}[\![\mathsf{S}]\!]X^1 \\
&\vdots \\
X^{n+1} &\triangleq \overrightarrow{\mathcal{S}}[\![\mathsf{S}]\!]Y^n\mathcal{E} \\
Y^{n+1} &\triangleq \overleftarrow{\mathcal{S}}[\![\mathsf{S}]\!]X^n \\
&\vdots
\end{aligned}
$$

where $\mathcal{E} = \text{leaf}[\top^{\mathbb{D}^\sharp_R}]$ is the initial environment for the forward analysis and $\mathcal{H}$ is the function that maps all program locations to $\text{leaf}[\bot^{\mathbb{D}^\sharp_R}]$.

Ideally, we would like to continue the iterates from Definition 4.6.1 until we reach a fix-point. That is, if we cannot further refine the forward semantics and cannot produce any new partition hints from one iterate to another then we have reached a stable transition system. However, characterizing this is difficult and we would have to design a widening operator. Fortunately, these iterates are a reduction which means that the iteration may be safely stopped at any point and the results will be sound [Cou19]. Thus, the abstract responsibility analysis of floating-point programs consists of iterating Definition 4.6.1 some N many times and then stopping the process. From the resulting transition system defined by the final iterates of the forward semantics we extract the left and right bounds of responsibility.

4.6.1 Concrete Responsibility Semantics

We begin by stating the concrete responsibility analysis from [DC19]. Since we only have an omniscient cognizance for floating-point responsibility analysis, we drop the definition of the cognizance and observation functions from our presentation. Given a behavior of interest $\mathcal{P}$ and a set of traces to be analyzed $\mathcal{T}$, every trace in $\sigma \in \mathcal{T}$ can be split into $\sigma = \sigma_H\sigma_R\sigma_F$ where H means history, R means responsible part and F means future. If $\bot^{Max} \subsetneq \mathbb{I}(\mathcal{S}^{Max}, \mathcal{L}^{Max}, \sigma_H\sigma_R) \subseteq \mathcal{P} \subsetneq \mathbb{I}(\mathcal{S}^{Max}, \mathcal{L}^{Max}, \sigma_H)$ holds then σ_H cannot guarantee $\mathcal{P}$ as it must be strictly greater than $\mathcal{P}$, which in our case becomes $\top^{Max}$. Furthermore, $\sigma_H\sigma_R$ guarantees that a behavior at least as strong as $\mathcal{P}$ occurs as it cannot be $\bot^{Max}$ due to the strict inclusion. In the trace $\sigma_H\sigma_R\sigma_F$, σ_R is said to be the responsible entity for $\mathcal{P}$ and we require $|\sigma_R| = 1$ such that only a single entity may be responsible for $\mathcal{P}$.

Note that for Figure 3.2 we have that $\mathbb{I}(\mathcal{S}^{Max}, \mathcal{L}^{Max}, \sigma_H\sigma_R)$ must equal $\mathcal{P}$ while $\mathbb{I}(\mathcal{S}^{Max}, \mathcal{L}^{Max}, \sigma_H)$ must equal $\top^{Max}$. In a more refined lattice such as Figure 3.3 this is no longer the case.

The function

$$\begin{aligned}
&\alpha_R \in \wp(E^{*\infty}) \to \wp(\wp(E^{*\infty})) \to \wp(E^{*\infty}) \to \wp(E^{*\infty}) \to \wp(E^{*\infty} \times E \times E^{*\infty})\\
&\alpha_R(\mathcal{S}^{Max}, \mathcal{L}^{Max}, \mathcal{P}, \mathcal{T}) \triangleq \{(\sigma_H, \sigma_R, \sigma_F) \mid \sigma_H\sigma_R\sigma_F \in \mathcal{T} \wedge |\sigma_R| = 1 \wedge\\
&\qquad\qquad \bot^{Max} \subsetneq \mathbb{I}(\mathcal{S}^{Max}, \mathcal{L}^{Max}, \sigma_H\sigma_R) \subseteq \mathcal{P} \subsetneq \mathbb{I}(\mathcal{S}^{Max}, \mathcal{L}^{Max}, \sigma_H)\}
\end{aligned}$$

defines the responsible entities for $\mathcal{P}$ for any given trace. Since $\alpha_R(\mathcal{S}^{Max}, \mathcal{L}^{Max}, \mathcal{P})$ preserves joins on traces $\mathcal{T}$ we also have the Galois connection

$$(\wp(E^{*\infty}), \subseteq) \underset{\alpha_R(\mathcal{S}^{Max}, \mathcal{L}^{Max}, \mathcal{P})}{\overset{\gamma_R(\mathcal{S}^{Max}, \mathcal{L}^{Max}, \mathcal{P})}{\leftrightarrows}} (\wp(E^{*\infty} \times E \times E^{*\infty}), \subseteq).$$

Lemma 4.6.1 ([DC19]). *If σ_R is a responsible entity for a behavior $\mathcal{P}$ in a valid trace $\sigma_H\sigma_R\sigma_F$ then $\sigma_H\sigma_R$ guarantees the occurrence of behavior $\mathcal{P}$ and there must exist another valid prefix trace $\sigma_H\sigma'_R$ such that behavior $\mathcal{P}$ is not guaranteed.*

4.6.2 Abstract Responsibility Semantics

In the abstract responsibility semantics we define the right and left bounds of responsibility. For the right bound of responsibility the idea is quite simple: any node that is the first one marked as $\mathfrak{B}$ on a path from the beginning of the transition system to an ending state must be the right bound of responsibility. This follows from the definition of α_R. For the left bound we will rely on Lemma 4.6.1.

Definition 4.6.2. The right bound of responsibility for property $\mathcal{P}$ in a given path in a transition system which starts at the starting state and ends at an ending state is the left-most node from the ending state that guarantees the behavior $\mathcal{P}$. Alternatively, the right bound of responsibility for a given path in a transition system is the first node that guarantees $\mathcal{P}$ when traversing that path. If no such node exists then the ending state of the system is the right bound.

Definition 4.6.3. The left bound of responsibility for property $\mathcal{P}$ in a given path in a transition system is the first node on that path that after a single transition any of its outgoing nodes no longer guarantees $\top^{Max}$. If no such node exists then the starting state of the system is the left bound and all nodes up the right bound are responsible candidates for $\mathcal{P}$.

The intuition for Definition 4.6.2 has already been given and it is the application of α_R. For Definition 4.6.3 we rely on the fact that if in the abstract we can show all outbound nodes from a given node lead to either $\mathfrak{B}$ or $\neg\mathfrak{B}$ then it must be that all concrete traces from that point on also are either responsible for $\mathfrak{B}$ or $\neg\mathfrak{B}$. By Lemma 4.6.1 this node must be the left bound of responsibility. Otherwise, if there are some outgoing nodes that are marked with $\top^{Max}$ then we cannot be sure in the concrete what the current node's choices will always cause a single behavior to occur.

Lemma 4.6.2. *The abstract responsibility analysis for floating-point programs is sound which follows from the soundness of the forward partitioning semantics of programs.*

To actually compute the left and right bounds of responsibility we may do a tree traversal on the final transition system of the forward semantics.

4.6.3 Running Example

As noted in Section 4.5.3 it is not possible to refine the transition system given in Figure 4.12 any further. So, we start finding the responsible entities for the behavior $\mathfrak{B}$ by tree traversal and get the following bounds of responsibility

$$\{[([2], t_{\textbf{None}}), ([3], t^2_{\textbf{val}})], [([2], t_{\textbf{None}}), ([3], t^4_{\textbf{val}})]\}.$$

The bound $[([2], t_{\textbf{None}}), ([3], t^2_{\textbf{val}})]$ describes the bounds of responsibility for the traces where the constraints on the affine noise symbols given by the node $([3], t^2_{\textbf{val}})$ hold. Similar reasoning applies for the pair $[([2], t_{\textbf{None}}), ([3], t^4_{\textbf{val}})]$. For both responsibility bounds [3] is the actual responsible entity as the error happens only after [3] makes its choice on whether to take the if branch or the else branch. This result is somewhat unintuitive as we know that with respect to the concrete semantics the actual responsible entity can either be [2] or [3]. However, this makes sense as [2] can only be responsible for $\mathfrak{B}$ if we know that [3] will pick its value such that the flow of computation goes to the if branch of the program. Furthermore, the choice that [2] makes is actually pushed into [3] such that node [3] makes two choices. This is not a problem in the abstract and is actually caused by the projection of variable `y` which means that `x` can never guarantee in the abstract that $\mathfrak{B}$ occurs even though we know in the concrete that such traces exist. Future work in the abstract backward semantics may lead to interesting partitioning schemes. In the case of Figure 4.12 if we could define a sort of chained partitionings where any partitioning is immediately followed up by another partitioning, then this would remedy the problem as long as we can estabilish that the chain does indeed guarantee $\mathfrak{B}$. Of course, the soundness of such a result needs to be shown. A promising direction is to use abstraction interpretation of CTL (Computation Tree Logic) [CE81] properties as presented in [UUM18]. For more information on CTL see [BK08]. Nevertheless, our current result is sound as both concrete responsible entities lie in the ranges of both bounds.

We can also calculate the responsible entities for the behavior $\neg\mathfrak{B}$. This gets us the set of bounds

$$\{[([2], t_{\mathbf{None}}), ([3], t^1_{\mathbf{val}})], [([2], t_{\mathbf{None}}), ([3], t^3_{\mathbf{val}})]\}$$

which is again a sound result. Overall, we see that the analysis has allowed us to obtain the most accurate bounds of responsibility with respect to both the process of forward-backward refinement and the concrete semantics.

4.6.4 Remarks

The abstract responsibility analysis of floating-point programs allows us to soundly approximate responsible entities for possible floating-point errors. Throughout this chapter we have presented the analysis, discussed limitations and looked at future directions. In terms of Definition 4.6.1, we expect that improvements in the forward semantics, especially of our partitioning strategies, and backward semantics will lead to improvements in the result. An implementation of this analysis could begin by alerting the user to the errors detected but not actually perform too many partitions on the system. This is to prevent a possible blow-up on the number of partitions. Then, the user could select whether they would like to pursue the issue further and the analysis would then perform a more thorough partitioning. If the user cannot still identify what's wrong with the program, then further partitions can be performed at which point the responsibile entity should slowly start becoming apparent. Another idea is to allow the programmer to pick parts of the program that must be correct or allow the programmer to only pick certain parts of the program to partition. These approaches are theoretically justified as the framework allows us to define many different partitioning without breaking soundness and have an ordering of extended transition systems. Furthermore, such practical considerations are also justified by the fact that not every floating-point error is bad, e.g. see the summation algorithms in [Hig93], and floating-point error is unavoidable. The user has also access to the amount of error that occurs as the constrained affine sets computes such values, so they may determine whether it makes sense to further pursue certain numerical issues.

The next step in this work is to implement an analyzer for this framework that is generic in terms of how the analysis should be carried out. We can then evaluate the results of our analysis experimentally and decide on a more specific analysis that works well. All the future directions mentioned throughout this chapter can also be looked into during the implementation and experimental evaluation phase to further improve the analysis.

4.7 Towards an Implementation

An implementation of the proposed analysis is the next step for this thesis. In this section we discuss some strategies and details for the implementation phase.

So far, the constrained affine sets domain we have presented and used throughout the analysis has relied on real intervals. This means that we wish to compute the possible real values for a given program variable. However, representing the coefficients of the affine forms and the intervals of the noise symbols as rational numbers in a computer, perhaps by using symbolic expressions, can be computationally costly and the use of real numbers is infeasible in general. Thus, we need a way to also soundly approximate these values. For real intervals, we say that a floating-point interval is a sound approximation of a real interval if it is a superset of that real interval. Similarly, a floating-point value is a sound approximation of a real value if its absolute value is greater than or equal to the real value. One way to realize a sound implementation of both the constrained affine sets and the analysis in general is to use double-precision floating-point values with outward rounding [GGP09]. If more precision is required then arbitrary precision floating-point libraries such as MPFR [Fou+07] may be used. These libraries allow the user to change the rounding mode and set the precision of the numbers being calculated. Since the Apron [JM09] library for numerical abstract interpretation already provides an interface to arbitrary precision arithmetic libraries, it is an ideal candidate for the implementation of the analysis.

In fact, the library Taylor1+ [GGP09] for constrained affine sets is also provided in the Apron framework as a domain. This library implements aspects of the constrained affine sets domain but does not fully automate the process of using constrained affine sets for floating-point analysis as described in Section 3.3. So, an implementation must take care to track noise symbols belonging to a real computation and the corresponding errors of the computation. Furthermore, new noise symbols that are used to model rounding errors must be generated manually. We may first introduce a new noise symbol to represent the error and then use the rounding error bounds given in Section 3.1 to bound it appropriately. Finally, any errors due to discontinuity must also be detected manually and introduced by the programmer as new noise symbols. Overall, Taylor1+ implements constrained affine sets for computing arbitrary real values and the programmer must take care to actually implement the domains given in [GP11] or [GP13]. This should not be a difficult task as all the information required to realize this implementation is already outlined in Section 3.1 and Section 3.3. However, we must make sure to pay attention to the fact that all aspects of the domain are implemented fully. For example, if we forget to introduce a noise symbol at some point or do not use the outward rounding mode of MPFR then the whole analysis will produce unsound results.

After implementing the domain for floating-point analysis, we must build the abstract interpreter. We may split the development of the tool into a few phases. Firstly, we need to implement the partitioning domain with hints from Section 4.4.3 and Section 4.4.4. This domain needs to define the tree structure from

Section 4.4.2 where each leaf node refers to our implementation of the constrained affine sets. Then, the partitioning functions can be realized using a combination of tree traversals and function calls to the constrained affine set domain.

The next phase involves implementing both the generic forward and generic backward abstract interpreters as presented in Section 4.4.5 and Section 4.5.2. This is a fairly straightforward process as we can produce an abstract syntax tree of programs using off-the-shelf program lexing and parsing tools and then pattern match on this tree. Each node of the tree corresponds to a synctactic component of our program and since we provide the semantics of all such components we may simply implement the provided semantic functions. Each function in turn will use the interface of the partitioning domain that we implemented in the first phase. Finally, we need to write some code that manages both abstract interpreters, performs the abstract responsibility analysis as described in Section 4.6 and extracts the responsible entities from the final result. Overall, while there may be a lot of parts that need be realized, this thesis lays out how each part should be implemented and the structure of the analysis allows us to modularize the implementation into individual components.

The Apron library provides C, C++ and OCaml interfaces which means our choice of programming language is constrained to these three. We plan to use OCaml to implement the domain so that we can take advantage of its powerful module system to split the analysis into parts and functional features such as pattern matching to implement the generic abstract interpreters. Furthermore, we can take advantage of the `ocamllex` (lexer) and `ocamlyacc` (parser) [Smi06] libraries to generate the abtract syntax tree of programs.

Chapter 5

Related Work

This thesis incorporates floating-point analysis with responsibility analysis, backward semantics and trace partitioning. We describe related work in the context of the analysis of floating-point programs.

5.1 Floating-point Verification

Existing work on the analysis and verification of floating-point programs is quite diverse. The area falls under a few broad categories. The Fluctuat tool [GP11] performs an abstract interpretation based analysis which has been shown to verify the numeric precision of embedded systems [Bou+09]. This work consists of a long lineage of ideas and its foundations can be traced to [Gou01] which gives an abstract domain to compute the loss in precision of a floating-point program and [Mar02] which defines a concrete semantics to identify sources of errors. Zonotope abstractions are introduced in [GP06] and further developed in [GGP10]. The constrained affine sets domain used to detect floating-point divergence is formulated in [GP13]. Other directions for this work involves forward under-approximating semantics [Gou+14], the analysis of probabilistic numeric programs [Adj+13] and combined over-under-approximating analyses of floating-point programs [GP19]. Besides Fluctuat, there is also the PRECiSA tool which was introduced in [Mos+17] and then formulated as an abstract interpretation in [Tit+18]. The real values of program variables and their errors are represented as symbolic expressions whose values are then computed using a branch-and-bound optimization algorithm implemented in Kodiak [Smi+15] and the results of the analysis are certified using the interactive theorem prover PVS [ORS92]. Similar to Fluctuat, PRECiSA is also sound under test divergence and computes the bounds of errors without assuming the stable test condition. This means that the symbolic error expressions also capture test divergence. Both Fluctuat and PRECiSA are able to identify the contributions of certain program points to the error of a program but they do not have a notion of responsibility. Other older abstract interpretation approaches such as polyhedra which are used in Astrée [CMC08] and octagons [Min04a] are able to detect floating-point run-time errors for 'real world' programs.

Optimization based tools such as FPTaylor [Sol+18] and Real2Float [MCD17] bound the error a program accrues due to floating-point arithmetic using optimization techniques. FPTaylor uses symbolic Taylor expansions to approximate the round-off error and then applies a global optimization procedure to obtain tight bounds of the error. Furthermore, the tool emits certificates to HOL Light [Har09]. The main drawback of FPTaylor is that its use is restriced to smooth functions and its design focuses on the analysis of arithmetic expressions. FPTuner [Chi+17] is a tool inspired by FPTaylor and focuses on allocating precision to program variables such that the total error of the program does not exceed a threshold while also minimizing the execution time of the program by choosing lower precision floating-point formats. Real2Float is able to compute certified bound of errors by using semi-definite programming and sum of squares certificates.

Floating-point arithmetic has been formalized in proof assistants such as in Coq [Bou+14] and HOL Light. Some examples include [Bol14] and [JSG15]. Gappa [DLM10] automatically computes the bounds of round-off errors for floating-point arithmetic expressions and then provides a proof that may be checked by Coq. However, the generation of these proofs along with the computation of the bounds may require hints by the user. The main challengse of proof assistant based approaches has been the lack of automation and the high barrier to entry due to the domain specific knowledge required to use them. Besides the formalization of errors, there are also formally verified compilers such as [Bol+15] which builds on top of the CompCert framework [Ler09] and provides a way to compile floating-point programs into semantically equivalent programs. Recall that common algebraic identities do not hold in floating-point so the compiler may not soundly perform transformations it may usually use for other data types such as integers. The Rosa [DK14] tool compiles an ideal real-valued implementation of a program to a floating-point version which meets a precision requirement if such a program exists. Rosa deals with test instability and combines SMT-solving with affine arithmetic.

5.2 Floating-Point Blame Analysis and Tuning

Precimonious [Rub+13] is a tool that uses delta-debugging to search for floating-point type configurations in a program to reduce the precision of floating-point variables. The goal is to find an optimal configuration that does not cause the error of a program to exceed a threshold while also lowering the precision of variables as much as possible. Higher precision floating-point types are both more power hungry and expensive in terms of how long it takes to compute with them. This work has been followed up with blame analysis as introduced in [Rub+16] that defines a set of floating-point variables in a program to blame. The high level idea is to iteratively vary the precisions of variables and decrease them as long as the total error of the program does not exceed a threshold. Then, the analysis builds what it refers to as blame sets and identifies the variables that it may safely lower their precisions. The variables whose precisions cannot be lowered any further can be considered as ‘blamed. This definition of blame somewhat corresponds to responsibility as the analysis considers the variables which would lead to large round-off errors as causes of those errors. However, both the analysis in Precimonious and blame analysis occurs dynamically and the results are normally local minima of an objective function. This means that there is no guarantee that the best configuration is found which in turn implies that the blame set is also not ideal. It might be that under a better configuration we may remove blame from some variables and assign blame to others. Furthermore, this work does not consider conditional divergence and is not sound for all possible inputs. In this thesis, the analysis is sound for all possible inputs and causality is estabilished in terms of a formal definition with respect to program traces. In terms of blame sets, this means that for any given trace we detect the ideal candidate to blame.

The work on Precimonious has demonstrated that it has good performance while our analysis’ performance is yet to be determined. In general, abstract interpretation based analyses are known to be more computationally expensive compared to dynamic analyses due to the expressiveness of the abstract domains.

Another approach to tuning is program re-writing. In [Tan+10] possible expression re-writes are searched from a database of templates with the goal of improving program precision. [Mar09] proposes an operational semantics to define the possible re-writings of program statements such that we obtain lower amounts of round-off error. Herbie [Pan+15] also proposes a heuristic search that selects re-writes of program expressions by sampling floating-point values while selecting re-writes from a pre-defined list of rules. An interesting aspect of Herbie is that it may also synthesize conditionals that can select different re-writes of the program depending on the run-time inputs. Program re-writing is orthogonal to the current focus of this thesis but an interesting direction to explore in the future.

5.3 Root Causes of Floating-Point Errors

Herbgrind [Pan+15] dynamically keeps track of dependencies to identify what it refers to as root causes of floating-point errors. Unlike responsibility analysis, the root cause of errors may be more than one program entity. These causes are determined by using taint analysis where every floating-point number is associated with a set of floating-point intstructions influencing its value. Then, to identify which floating-point operations cause error, Herbgrind uses the notion of local error from Herbie [Pan+15] which measures how much error an operation's output would have even if its inputs were accurately computed and then rounded to floats. This is similar to the notion of choice we use in this thesis where every floating-point variable is allowed to choose either its real or floating-point value which in turn may or may not cause erroneous behaviors. Unlike our analysis, Herbrgrind also identifies errors over function calls and heap allocations.

The limitations of Herbgrind are that the tool requires a set of representative inputs from the programmer and treats the inputs to the program discretely. The analysis in this thesis considers a real interval and detects all errors along with the associated sub-intervals that lead to the error. Herbgrind only requires one run of the program which it dyamically instruments. We require multiple iterations of a sequence of analyses to determine responsible entities.

Besides Herbgrind there are also a few works that attempt to find floating-point inputs that lead to high amounts of error in programs. [Chi+14] performs a random search of possible floating-point input values to locate those that lead to high amounts of error while [Zou+15] proposes a genetic algorithm to find such inputs. These analyses occur dynamically and are not sound but do exhibit good performance in general.

The key difference between Herbgrind and this thesis is that Herbgrind defines responsibility as a set of candidates. The concrete semantics of responsibility will assign to any one trace only a single responsible entity. In the abstraction of this semantics we try to group similar traces together in such a way that we can identify responsibile entities. Anything that cannot be pinpointed becomes a range of responsible entities. For methods that attempt to find inputs that lead to high error there is no notion of responsibility. With floating-point programs there always exists inputs that lead to degenerate outcomes but these outcomes are most of the time caused by very specific instructions or program expressions. For this thesis instead of relying on the values of the inputs themselves we abstract program traces which better characterizes where the floating-point program may go wrong for a wide range of inputs.

5.4 Floating-Point Conditional Divergence

Floating-point error often leads to divergence in the control flow of programs. The IEEE standard sets the expectation that floating-point programs be correctly rounded versions of real numbers. However, this does not exclude the fact that errors introduced by floating-point arithmetic may lead to behaviors that should not happen if the program were computed using real numbers. Continuity, as introduced in [CGL10], attempts to determine if a program represents a continuous function. If very small pertubations in the program's inputs leads to very small changes in the program's outputs then we say that the program is continuous. Such an analysis can then be used to decide whether a program is robust or not as robustness is also defined in terms of small pertubations to inputs and observing changes in outputs. Another notion of robustness is given in [MS09]. Here the authors propose an algorithm that symbolically traverses program paths and collects constraints on the inputs and outputs of the paths. Then, for each pair of program paths the algorithm can determine the values of the input that causes the program to explore these two paths and for which inputs the difference in the output from the two paths is maximized. There also exists work on synthesizing robust programs such as in [MRT11] and [Tab+12].

With respect to floating-point programs, [CFK14] models floating-point discontinuity by considering every branch to be a non-deterministic one. This leads to the detection of such errors but the results can be coarse. The authors of FPTaylor and FPTuner also have another work [CGR15] which proposes search heuristics to find inputs that lead to divergence. [Bag+13] generates test inputs for floating-point programs for a given control flow in a program. Finally, the work given in [GP13] which this thesis uses as its underlying asbtract domain shows how constrained affine sets may be used to both detect conditional divergence and also characterize divergence in terms of the constraints on the noise symbols of the affine forms. This in turn allows the analysis to determine the interval of inputs that lead to a specific divergent path.

As far as we are aware, this thesis is the first work to define responsible program entities for divergence (or non-robust behavior) in floating-point programs. We show not only the set of inputs or the constraints on certain program expression that lead to discontinuity, but also define the cause of such behaviors using counterfactual dependence. This way, programmers can identify under which conditions divergence occurs along with which program expressions they should consider fixing to mitigate the error. Since responsibility analysis in the context of floating-point programs corresponds to whether a program entity chooses its floating-point value, the programmer can implement fixes to programs in terms of lowering the amount of error accrued.

5.5 Backward Semantics and Trace Partitioning

There does not exist much literature for the backward semantics of floating-point programs. The construction given in [Min14] applies broadly to numerical domains and the authors of Fluctuat also provide a forward under-approximating semantics for affine arithmetic [GP07]. This work is based on generalized affine arithmetic [Gol+05] which itself is based on Kaucher arithmetic [Kau80]. The overall idea is to create existentially quantified domains which means that instead of considering all traces that satisfy a property we consider the question of whether there exists a trace that satisfies a property. [GP07] has lead to the development of [Gou+14] which can be used to under-approximate the set of reachable states of a program. An over-under analyses can be combined for dynamical systems [GP19] such that the set of reachable states for both types of analyses can be expressed as an ordinary differential equation. The next step for responsibility analysis of floating-point programs is to formulate an analysis that works on all combinations of under-over-approximating forward-backward analyses. Such an analysis will allow for the discovery of better partitions of traces that may improve both the results of the partitioning and the performance of the analysis. We may begin exploring this direction by first considering the application generalized affine arithmetic for constrained affine sets. This topic is briefly mentioned in [Gou+14] but not formalized.

Trace partitioning is a very general framework and a thorough detail of its history can be found in [RM07]. In terms of floating-point programs, most of the analyses mentioned in that paper form some kind of partitioning on the input space to determine diferent kinds of outputs such as those that lead to high amount of error. The work in this thesis is different compared to this type of partitioning as the partitions we seek are aimed to separate traces based on their hyperproperties [CS10]. We try to delineate as much as we can and as early as we can in the program execution the traces that lead to $\mathfrak{B}$ from those that lead to $\neg\mathfrak{B}$. Such an approach may be also applicable to both static and dynamic analyses and could lead to interesting characterizations of blame, root causes and robustness.

Chapter 6

Conclusion

We have proposed an abstract interpretation based static analysis to identify responsible entities for floating-point analysis. The analysis incorporates both the forward and backward semantics of programs to perform trace partitionings on a transition system of the program being analyzed and identify tight bounds where a responsible entity for a behavior $\mathcal{P}$ must lie in. The presented analysis is very expressive and liberally partitions programs. However, this does not mean an implementation must follow the scheme presented here exactly. More coarse partitions can be plugged in and as long as they are shown to be approximations of the partitioning functions demonstrated here, then new sound analyses can be constructed. This is because we have demonstrated the soundness of this analysis for a general class of partitions. Throughout the presentation we have discussed potential future directions and the possible points that may need to be addressed after implementing the analysis. Thus, the next step in this research is to create an analyzer that implements the analysis presented here and evaluate it on various floating-point programs from toy ones to more realistic examples.

Bibliography

[Tar+55] Alfred Tarski et al. "A lattice-theoretical fixpoint theorem and its applications." In: *Pacific journal of Mathematics* 5.2 (1955), pp. 285–309.

[CC76] Patrick Cousot and Radhia Cousot. "Static determination of dynamic properties of programs". In: *Proceedings of the 2nd International Symposium on Programming, Paris, France*. Dunod. 1976, pp. 106–130.

[CC77] 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*. 1977, pp. 238–252.

[CH78] Patrick Cousot and Nicolas Halbwachs. "Automatic discovery of linear restraints among variables of a program". In: *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 1978, pp. 84–96.

[CC79] Patrick Cousot and Radhia Cousot. "Systematic design of program analysis frameworks". In: *Proceedings of the 6th ACM SIGACT-*

SIGPLAN symposium on Principles of programming languages. 1979, pp. 269–282.

[Kau80] Edgar Kaucher. “Interval analysis in the extended interval space IR”. In: *Fundamentals of Numerical Computation (Computer-Oriented Numerical Analysis)*. Springer, 1980, pp. 33–49.

[CE81] Edmund M Clarke and E Allen Emerson. “Design and synthesis of synchronization skeletons using branching time temporal logic”. In: *Workshop on Logic of Programs*. Springer. 1981, pp. 52–71.

[Gol91] David Goldberg. “What every computer scientist should know about floating-point arithmetic”. In: *ACM Computing Surveys (CSUR)* 23.1 (1991), pp. 5–48.

[CC92] Patrick Cousot and Radhia Cousot. “Abstract interpretation frameworks”. In: *Journal of logic and computation* 2.4 (1992), pp. 511–547.

[ORS92] Sam Owre, John M Rushby, and Natarajan Shankar. “PVS: A prototype verification system”. In: *International Conference on Automated Deduction*. Springer. 1992, pp. 748–752.

[CS93] JLD Comba and J Stolfi. *Affine arithmetic and its applications to computer graphics. Anais do VII SIBGRAPI, 9–18*. 1993.

[Hig93] Nicholas J Higham. “The accuracy of floating point summation”. In: *SIAM Journal on Scientific Computing* 14.4 (1993), pp. 783–799.

[Cor+97] Agostino Cortesi et al. “Complementation in abstract interpretation”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 19.1 (1997), pp. 7–47.

[CC98] Patrick Cousot and Radhia Cousot. “Introduction to abstract interpretation”. In: *Course notes for the “NATO Int. Summer School* (1998).

[Cou99] Patrick Cousot. “The calculational design of a generic abstract interpreter”. In: *Calculational system design.* IOS Press, 1999, pp. 421–505.

[Cou01] Patrick Cousot. “Abstract interpretation based formal methods and future challenges”. In: *Informatics.* Springer. 2001, pp. 138–156.

[Gou01] Eric Goubault. “Static analyses of the precision of floating-point operations”. In: *International Static Analysis Symposium.* Springer. 2001, pp. 234–259.

[Cou02] Patrick Cousot. “Constructive design of a hierarchy of semantics of a transition system by abstract interpretation”. In: *Theoretical Computer Science* 277.1-2 (2002), pp. 47–103.

[Dan02] Roger B Dannenberg. “Danger in Floating-Point-to-Integer Conversion”. In: *Computer Music Journal* 26.2 (2002), pp. 4–4.

[Ham02] Dick Hamlet. “Continuity in software systems”. In: *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis.* 2002, pp. 196–200.

[Mar02] Matthieu Martel. “Propagation of roundoff errors in finite precision computations: a semantics approach”. In: *European Symposium on Programming.* Springer. 2002, pp. 194–208.

[Cou03] Patrick Cousot. “Verification by abstract interpretation”. In: *Verification: Theory and Practice.* Springer, 2003, pp. 243–268.

[CC04] Patrick Cousot and Radhia Cousot. “Basic concepts of abstract interpretation”. In: *Building the Information Society*. Springer, 2004, pp. 359–366.

[Min04a] Antoine Miné. “Relational abstract domains for the detection of floating-point run-time errors”. In: *European Symposium on Programming*. Springer. 2004, pp. 3–17.

[Min04b] Antoine Miné. “Weakly relational numerical abstract domains”. PhD thesis. 2004.

[Gol+05] Alexandre Goldsztejn et al. “Modal intervals revisited: a mean-value extension to generalized intervals”. In: *First International Workshop on Quantification in Constraint Programming*. 2005.

[MR05] Laurent Mauborgne and Xavier Rival. “Trace partitioning in abstract interpretation based static analyzers”. In: *European Symposium on Programming*. Springer. 2005, pp. 5–20.

[GP06] Eric Goubault and Sylvie Putot. “Static analysis of numerical algorithms”. In: *International Static Analysis Symposium*. Springer. 2006, pp. 18–34.

[Min06] Antoine Miné. “The octagon abstract domain”. In: *Higher-order and symbolic computation* 19.1 (2006), pp. 31–100.

[Smi06] Joshua B Smith. *Practical OCaml*. Vol. 3. 1. Springer, 2006.

[Fou+07] Laurent Fousse et al. “MPFR: A multiple-precision binary floating-point library with correct rounding”. In: *ACM Transactions on Mathematical Software (TOMS)* 33.2 (2007), 13–es.

[GP07] Eric Goubault and Sylvie Putot. “Under-approximations of computations in real numbers based on generalized affine arithmetic”. In: *International Static Analysis Symposium*. Springer. 2007, pp. 137–152.

[RM07] Xavier Rival and Laurent Mauborgne. “The trace partitioning abstract domain”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 29.5 (2007), 26–es.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking.* MIT press, 2008.

[CMC08] Liqian Chen, Antoine Miné, and Patrick Cousot. “A sound floating-point polyhedra abstract domain”. In: *Asian Symposium on Programming Languages and Systems*. Springer. 2008, pp. 3–18.

[GP08] Eric Goubault and Sylvie Putot. “Perturbed affine arithmetic for invariant computation in numerical program analysis”. In: *arXiv preprint arXiv:0807.2961* (2008).

[08] “IEEE Standard for Floating-Point Arithmetic”. In: *IEEE Std 754-2008* (2008), pp. 1–70.

[Bou+09] Olivier Bouissou et al. “Space software validation using abstract interpretation”. In: 2009.

[GGP09] Khalil Ghorbal, Eric Goubault, and Sylvie Putot. “The zonotope abstract domain taylor1+”. In: *International Conference on Computer Aided Verification*. Springer. 2009, pp. 627–633.

[Har09] John Harrison. “HOL light: An overview”. In: *International Conference on Theorem Proving in Higher Order Logics*. Springer. 2009, pp. 60–66.

[JM09] Bertrand Jeannet and Antoine Miné. "Apron: A library of numerical abstract domains for static analysis". In: *International Conference on Computer Aided Verification.* Springer. 2009, pp. 661–667.

[Ler09] Xavier Leroy. "Formal verification of a realistic compiler". In: *Communications of the ACM* 52.7 (2009), pp. 107–115.

[MS09] Rupak Majumdar and Indranil Saha. "Symbolic robustness analysis". In: *2009 30th IEEE Real-Time Systems Symposium.* IEEE. 2009, pp. 355–363.

[Mar09] Matthieu Martel. "Program transformation for numerical precision". In: *Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation.* 2009, pp. 101–110.

[CGL10] Swarat Chaudhuri, Sumit Gulwani, and Roberto Lublinerman. "Continuity analysis of programs". In: *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* 2010, pp. 57–70.

[CS10] Michael R Clarkson and Fred B Schneider. "Hyperproperties". In: *Journal of Computer Security* 18.6 (2010), pp. 1157–1210.

[DLM10] Florent De Dinechin, Christoph Lauter, and Guillaume Melquiond. "Certifying the floating-point implementation of an elementary function using Gappa". In: *IEEE Transactions on Computers* 60.2 (2010), pp. 242–253.

[GGP10] Khalil Ghorbal, Eric Goubault, and Sylvie Putot. "A logical product approach to zonotope intersection". In: *International Conference on Computer Aided Verification.* Springer. 2010, pp. 212–226.

[Tan+10] Enyi Tang et al. "Perturbing numerical calculations for statistical analysis of floating-point program (in) stability". In: *Proceedings of the 19th international symposium on Software testing and analysis*. 2010, pp. 131–142.

[Bus11] D Bushnell. "Continuity analysis for floating point software". In: *Proceedings of the 4th workshop on Numerical Software Verification (NSV'11)*. 2011.

[CCM11] Patrick Cousot, Radhia Cousot, and Laurent Mauborgne. "The reduced product of abstract domains and the combination of decision procedures". In: *International Conference on Foundations of Software Science and Computational Structures*. Springer. 2011, pp. 456–472.

[GP11] Eric Goubault and Sylvie Putot. "Static analysis of finite precision computations". In: *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer. 2011, pp. 232–247.

[MRT11] Rupak Majumdar, Elaine Render, and Paulo Tabuada. "A theory of robust software synthesis". In: *arXiv preprint arXiv:1108.3540* (2011).

[GLP12] Eric Goubault, Tristan Le Gall, and Sylvie Putot. "An accurate join for zonotopes, preserving affine input/output relations". In: *Electronic Notes in Theoretical Computer Science* 287 (2012), pp. 65–76.

[PMR12] Olivier Ponsini, Claude Michel, and Michel Rueher. "Refining abstract interpretation based value analysis with constraint programming techniques". In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2012, pp. 593–607.

[Tab+12] Paulo Tabuada et al. “Input-output robustness for discrete systems”. In: *Proceedings of the tenth ACM international conference on Embedded software*. 2012, pp. 217–226.

[Adj+13] Assale Adje et al. “Static analysis of programs with imprecise probabilistic inputs”. In: *Working Conference on Verified Software: Theories, Tools, and Experiments*. Springer. 2013, pp. 22–47.

[Bag+13] Roberto Bagnara et al. “Symbolic path-oriented test data generation for floating-point programs”. In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE. 2013, pp. 1–10.

[GP13] Eric Goubault and Sylvie Putot. “Robustness analysis of finite precision implementations”. In: *Asian Symposium on Programming Languages and Systems*. Springer. 2013, pp. 50–57.

[Rub+13] Cindy Rubio-González et al. “Precimonious: Tuning assistant for floating-point precision”. In: *SC’13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE. 2013, pp. 1–12.

[Bol14] Sylvie Boldo. “Deductive formal verification: how to make your floating-point programs behave”. PhD thesis. 2014.

[Bou+14] Pierre Boutillier et al. *Coq 8.4 Reference Manual*. Research Report. The Coq Development Team. Inria, July 2014. URL: https://hal.inria.fr/hal-01114602.

[CFK14] Swarat Chaudhuri, Azadeh Farzan, and Zachary Kincaid. "Consistency analysis of decision-making programs". In: *ACM SIGPLAN Notices* 49.1 (2014), pp. 555–567.

[Chi+14] Wei-Fan Chiang et al. "Efficient search for inputs causing high floating-point errors". In: *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming.* 2014, pp. 43–52.

[DK14] Eva Darulova and Viktor Kuncak. "Sound compilation of reals". In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* 2014, pp. 235–248.

[Gou14] Frédéric Goualard. "How do you compute the midpoint of an interval?" In: *ACM Transactions on Mathematical Software (TOMS)* 40.2 (2014), pp. 1–25.

[Gou+14] Eric Goubault et al. "Inner approximated reachability analysis". In: *Proceedings of the 17th international conference on Hybrid systems: computation and control.* 2014, pp. 163–172.

[Min14] Antoine Miné. "Backward under-approximations in numeric abstract domains to automatically infer sufficient program conditions". In: *Science of Computer Programming* 93 (2014), pp. 154–182.

[Ber+15] Julien Bertrane et al. "Static analysis and verification of aerospace software by abstract interpretation". In: *Foundations and Trends® in Programming Languages* 2.2-3 (2015), pp. 71–190.

[Bol+15] Sylvie Boldo et al. "Verified compilation of floating-point computations". In: *Journal of Automated Reasoning* 54.2 (2015), pp. 135–163.

[CGR15] Wei-Fan Chiang, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. "Practical floating-point divergence detection". In: *Languages and Compilers for Parallel Computing*. Springer. 2015, pp. 271–286.

[JSG15] Charles Jacobsen, Alexey Solovyev, and Ganesh Gopalakrishnan. "A parameterized floating-point formalizaton in HOL Light". In: *Electronic Notes in Theoretical Computer Science* 317 (2015), pp. 101–107.

[Mac15] James Mackintosh. "Beware lessons of history when dealing with quirky indices". In: *Financial Times* (Aug. 2015). URL: https://www.ft.com/content/40e78992-481e-11e5-af2f-4d6e0e5eda22.

[Pan+15] Pavel Panchekha et al. "Automatically improving accuracy for floating point expressions". In: *ACM SIGPLAN Notices* 50.6 (2015), pp. 1–11.

[Smi+15] Andrew P Smith et al. "A rigorous generic branch and bound solver for nonlinear problems". In: *Proceedings of the 2015 17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. IEEE. 2015, pp. 71–78.

[Zou+15] Daming Zou et al. "A genetic algorithm for detecting significant floating-point inaccuracies". In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. IEEE. 2015, pp. 529–539.

[Rub+16] Cindy Rubio-González et al. "Floating-point precision tuning using blame analysis". In: *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE. 2016, pp. 1074–1085.

[Chi+17] Wei-Fan Chiang et al. "Rigorous floating-point mixed-precision tuning". In: *ACM SIGPLAN Notices* 52.1 (2017), pp. 300–315.

[MCD17] Victor Magron, George Constantinides, and Alastair Donaldson. "Certified roundoff error bounds using semidefinite programming". In: *ACM Transactions on Mathematical Software (TOMS)* 43.4 (2017), pp. 1–31.

[Mar17] Matthieu Martel. "Floating-point format inference in mixed-precision". In: *NASA Formal Methods Symposium.* Springer. 2017, pp. 230–246.

[Min+17] Antoine Miné et al. "Tutorial on static inference of numeric invariants by abstract interpretation". In: *Foundations and Trends® in Programming Languages* 4.3-4 (2017), pp. 120–372.

[Mos+17] Mariano Moscato et al. "Automatic estimation of verified floating-point round-off errors via static analysis". In: *International Conference on Computer Safety, Reliability, and Security.* Springer. 2017, pp. 213–229.

[Mul+18] Jean-Michel Muller et al. *Handbook of Floating-Point Arithmetic.* 2nd. Birkhäuser Basel, 2018. ISBN: 3319765256.

[Sol+18] Alexey Solovyev et al. "Rigorous estimation of floating-point round-off errors with symbolic taylor expansions". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 41.1 (2018), pp. 1–39.

[Tit+18] Laura Titolo et al. "An abstract interpretation framework for the round-off error analysis of floating-point programs". In: *International Conference on Verification, Model Checking, and Abstract Interpretation.* Springer. 2018, pp. 516–537.

[UUM18] Caterina Urban, Samuel Ueltschi, and Peter Müller. "Abstract interpretation of CTL properties". In: *International Static Analysis Symposium*. Springer. 2018, pp. 402–422.

[Cou19] Patrick Cousot. *NYU Computer Science CSCI-GA.3140-001. Lecture notes: Abstract Interpretation*. 2019. URL: https://cs.nyu.edu/~pcousot/courses/spring19/CSCI-GA.3140-001/.

[DC19] Chaoqiang Deng and Patrick Cousot. "Responsibility analysis by abstract interpretation". In: *International Static Analysis Symposium*. Springer. 2019, pp. 368–388.

[GP19] Eric Goubault and Sylvie Putot. "Inner and outer reachability for the verification of control systems". In: *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*. 2019, pp. 11–22.

[19] "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), pp. 1–84.

[Ske] Robert Skeel. *Roundoff Error and the Patriot Missile*. URL: http://mate.uprh.edu/~pnegron/notas4061/patriot.htm.