MAX PLANCK INSTITUTE
FOR SOFTWARE SYSTEMS

Internship report – Second year of master's degree

January 31 – June 16, 2017

Certificate Checking in Coq and HOL4 for Static Analyses of Mixed-Precision Floating-Point Arithmetic

Author:
Raphaël Monat
ENS de Lyon
Lyon, France

Supervisor:
Eva Darulová
Tenure-track faculty
Max Planck Institute for Software Systems
Saarbrücken, Germany

Abstract

Designing correct numerical programs using floating-point arithmetic is a challenge for non-experts. As a finite set of numbers is representable, computations can have roundoff errors. Accumulating these errors in a program can lead to numerical instabilities and bugs. Thus, having tools being able to bound roundoff errors in programs helps assessing the reliability of a given program. Exactly bounding these errors is undecidable, so one must resort to approximations. Static analyzers can bound the difference between the ideal, real-valued behavior of a program and its actual behavior. Most static analyzers are not formally verified, so that bugs in the analysis may exist and yield wrong results. Another approach would be to ask the programmer to describe numerical programs in terms of real-valued computations, and then let a compiler handle the translation into an optimal floating-point program. In particular, the compiler can perform mixed-precision tuning: it can choose to strike a given balance between the precision and the computational cost of the computations using different floating-point formats. This approach has been implemented in a tool called Daisy and written in Scala. Daisy also provides guarantees on the behavior of the generated programs by giving error bounds. To strengthen these guarantees, a certificate checker has been developed in two theorem provers – Coq and HOL4 – to formally prove that the interval-based analysis of Daisy is sound. It was able to check analyses where the floating-point format is uniform. During my internship, I worked on extending the certificate checker to the mixed-precision, interval-based analysis of Daisy. I also worked on a formalization of affine arithmetic in Coq. This report presents an overview of Daisy and its certificate checker, and a detailed presentation of the extensions I worked on.

1 Introduction

Floating-point numbers are a common choice when designing numerical programs. The hardware implementation of most operations makes them easy to use. They strike a good balance between precision and computational cost compared to arbitrary precision computations. In the past, implementations of floating-point arithmetic were not unified and yielded different results on different platforms. Now, thanks to the IEEE 754 standardization of floating-point arithmetic, numerical programs are more portable and mathematical reasoning about floating-point arithmetic is now possible. As floating-point numbers cannot exactly represent every real number, computations may not be exact and have roundoff errors. Using the IEEE 754 standard, it is possible to bound these roundoff errors. This standard also lead the way to subtle floating-point numerical computations ensuring minimal losses of precision, such as the one presented by Ogita et al. [17], using compensations to gain back accuracy. However, even if this standard is now well-known by a specialized community, floating-point computations are not well understood by most developers who may write unstable numerical programs.

1.1 Designing numerical programs is still a difficult task

Goldberg [9] warns developers that designing numerical programs working on floating-point numbers is a difficult task that should not be neglected. Two main features of floating-point arithmetic are mentioned. First, roundoff errors entailed by the base 2 representation of floating-point numbers and the finiteness of the precision can add up and create catastrophic errors. Second, floating-point arithmetic does not satisfy most algebraic properties valid on real numbers.

Even in really specialized domains, the design of numerical programs has been overlooked. An example of a disastrous consequence is the Patriot missile accident that happened in Dhahran in 1991[1]: 0.1 was not exactly representable in the base 2 floating-point system used, so that roundoff errors were accumulated over time, resulting in a missile deviating from its intended target and hitting US barracks.

Another example of catastrophic errors resulting in a counterintuitive behavior is the computation of the sequence defined below, and taken from [16]. For all $n \geq 2$, $u_n \geq 6$, so there is no risk of division by zero, and this sequence is well defined. The limit of this sequence is 6. However, due to roundoff errors accumulating at each step, a program computing this sequence using floating-point numbers converges to 100, even if an extended precision is used.

$$\begin{cases} u_0 = 2 \\ u_1 = -4 \\ u_{n+2} = 111 - \dfrac{1130}{u_{n+1}} + \dfrac{3000}{u_{n+1} \cdot u_n} \end{cases}$$

Developers aware of the consequences of roundoff errors usually use the most precise floating-point format available rather than the optimal precision, to reduce the risk of catastrophic errors. However, this results in a loss of performance.

To design safe, and yet efficient numerical programs, it is necessary to resort to automatic tools. These tools should be automatic, as the user usually has not a deep understanding of the technical details concerning floating-point arithmetic. One way to assist developers is to design a static analyzer being able to compare the ideal, real-valued behavior of a program with its floating-point based behavior, by bounding the value and roundoff error of each variable in a program. This will be

[1] `https://web.archive.org/web/20100702180720/http://mate.uprh.edu/~pnm/notas4061/patriot.htm`

described in the next subsection. Static analyzers can also be used to help choose the right precision for each variable in a program. Recently, the idea of automatically choosing the right precision for the right performance has been extended to mixed-precision computations [18, 4], where the goal is to achieve the fastest computations given a maximal precision error.

A similar approach, described in subsection 1.3 is to ask the developer to describe numerical programs using real numbers, and then let a tool perform a translation to floating-point numbers. This approach also requires a static analyzer to perform the translation. On the contrary to a static analyzer, a compiler manipulating real-based programs is also able to perform aggressive code optimizations, using algebraic identities valid on real numbers. Such a compiler also chooses the optimal precision for each variable.

1.2 Static analysis of floating-point programs

To ensure safety of numerical programs, one approach is to bound the difference between the ideal, real-valued behavior of a program, and its floating-point-based behavior. It is possible to find such upper bounds using static analyzers. These analyzers are fully automatic, and do not require to run the analyzed program, but only to read its source code. As problems related to program analysis are mostly undecidable, static analyzers usually compute an over-approximation of the analyzed program behavior.

Soundness This over-approximation property is fundamental and should be guaranteed by any static analysis: if only over-approximations are performed, every behavior of the program will be captured by the analysis. In that case, the analysis is said to be sound. In particular, if a static analyzer does not find any bugs in a program, then this program does not have any bugs. The goal of static analysis developers is to find a good balance between the precision of the analysis and its performance. If a static analyzer is precise but really slow, it will probably be unused. If an analyzer is too imprecise, it can then declare too many safe programs unsafe due to over-approximations, and there would be no point in running it.

Implementation bugs By definition, a sound static analysis provides theoretical guarantees on the results it computes. However, the proof of soundness done on paper may contain invalid reasoning, yielding a false result. More importantly, a static analyzer implemented on a machine can also differ from the designed static analysis, due to implementation errors and bugs. This is due to the fact that there is no formal proof that the static analyzer actually performs a given static analysis. An exception to this argument is the Verasco static analyzer. In that case, Jourdan et al. [14] used the Coq proof assistant to implement a static analyzer and prove its soundness. As such, Verasco gives much stronger guarantees concerning the soundness of both the static analysis and its implementation, compared to other static analyzers. A downside to this approach is that it takes much more development time and may trade performance for easier proofs.

Floating-point analysis Concerning floating-point programs, a well-known analysis consists in approximating each variable by an interval. This will be presented in section 2.2. Analyses based on the interval domain have a low computational cost, but their precision can be improved. In Fluctuat – a state-of-the-art static analyzer of floating-point programs – Goubault and Putot [11] use a more expressive numerical domain called "affine real form plus error term domain". This domain uses affine arithmetic to keep the real-value range of each variable. Affine arithmetic is a numerical domain usually more precise than interval arithmetic. To gain more precision, this domain also tracks the roundoff errors separately.

Static analyzers are thus able to warn the user of possible errors in the programs they design. These analyzers must be sound, but this soundness property might be violated by some error, either in the paper proof of soundness or due to an implementation error. The use of a static analyzer does not overcome the design issues mentioned before. It cannot perform optimizations either. Rather, it may help the developer find and correct flaws in his code *a posteriori*.

1.3 A compiler from reals to floats

As we have seen in the last subsection, static analysis of floating-point programs relates the result of an ideal, real-valued computation with its actual, floating-point counterpart. By definition, these static analyzers take as input floating-point programs. A slightly different approach is to change the input language, so that the user would give real-valued programs in input. Then, a tool performs the translation into a floating-point program. This tool will still contain a static analysis phase, to guarantee that the compiled floating-point program behaves similarly to the input program. This approach has some really interesting properties: as the input programs are described in terms of real numbers, performing aggressive optimizations is now possible. Such a compiler can also choose the precision of each variable, and strike a given balance between precision and performance.

This concept of compiler for real numbers has been developed by my advisor in [6], and is available as a tool called *Daisy*. Daisy is written in Scala, and will be presented in more depth in section 2.5. A recent development has been to formally verify the analysis made by Daisy, and prove that this analysis is sound. It would have been possible to rewrite Daisy's analysis into a proof assistant, and show that this implementation is sound. This design would have been similar to Verasco's. However, this approach would have been complicated here, as Daisy's framework is used to implement and test a few different ideas. Instead, the formal verification of Daisy relies on certificates, and consists into two parts. The core of Daisy has been extended to include a certificate generator. When an analysis is performed, Daisy describes the result of its analysis into a file called a certificate. This certificate is then run through a certificate checker, checking that the analysis performed by Daisy was right. The certificate checker was written in both Coq [5] and HOL4 [10], two common theorem provers. In both languages, it is formally proved that if the checker validates a certificate, then the analysis performed was sound.

1.4 Extension to mixed-precision floating-point arithmetic

The IEEE 754 standard specifies three different floating-point formats (32, 64 and 128 bits), from a low-cost, low-precision format to a higher-cost, higher-precision format. Lately, tuning automatically floating-point programs using mixed-precision floating-point arithmetic has become increasingly popular [18, 4]. This mixed-precision tuning lets users choose a better balance between the precision of a program and its computational cost. For example, mixed-precision tuning allows trading some precision for a lower computation time and a lower energy consumption. In other cases, this tuning can result in more precise results having a low computational overhead. Recently, Daisy gained support for mixed-precision tuning.

However, this mixed-precision tuning was not supported in the formal verification of Daisy. When I arrived, the certificate generation and certificate checking was already developed for single-precision floating-point arithmetic in both Coq and HOL4. This formalization was done by Heiko Becker (a PhD student in the lab), Eva Darulovà (my supervisor) and Magnus Myreen (Associate Professor at Chalmers University, Sweden). My main work during this internship has been to extend this formalization in both Coq and HOL4 to mixed-precision arithmetic. From a conceptual point of view, such an extension seems to be a fairly simple task. However, understanding the formal proofs

in two different theorem provers and extending these formal proofs turned out to be challenging.

1.5 Outline

The next section introduces background material. Section 3 describes my main contribution during this internship, which is the extension of the formal checking of Daisy's analyses in mixed-precision. In Section 4, I describe another ongoing work, which is a formalization of affine arithmetic in Coq. Section 5 provides a brief related work. The conclusion of my report also describes other side projects I have carried out during my internship.

2 Background on Daisy

In this section, I first provide a brief overview on floating-point arithmetic. I describe two ways to do range analysis, using either interval or affine arithmetic. Then, I introduce the Coq and HOL4 proofs assistants, before giving an overview of Daisy and of its formalization.

2.1 Floating-point arithmetic

A much more comprehensive coverage of floating-point arithmetic can be found in [16]. The IEEE 754 standard is defined in [1].

Floating-point numbers are defined with respect to a format, consisting in four integers: a base number β (sometimes called a radix), a precision p, a minimal exponent e_{min} and a maximal exponent e_{max} (with $e_{min} < e_{max}$). For example, the format of single-precision floating-point numbers defined by the IEEE 754 standard is $\phi_{32} = (2, 24, -126, 127)$, and the format of double-precision numbers is $\phi_{64} = (2, 53, -1022, 1023)$. Once a format $\phi = (\beta, p, e_{min}, e_{max})$ is fixed, the set of finite floating-point numbers is:

$$\mathbb{FP}_\phi = \{(-1)^s \cdot M \cdot \beta^{e-p+1} \mid s \in \{0,1\}, M \in \mathbb{N}, e \in \mathbb{Z},\, 0 \leq M < \beta^p,\, e_{min} \leq e \leq e_{max}\}$$

The number M is the integral significand and usually named the mantissa. The precision p of the format corresponds to the maximal number of digits of the number M written in base β.

Normal and subnormal numbers The set of floating-point numbers defined above can be normalized, and split into a set of normal numbers and a set of subnormal numbers. The set of subnormal numbers is the one having the exponent fixed to $e = e_{min}$.

$$\begin{aligned}\mathbb{FP}_\phi =& \{(-1)^s \cdot M \cdot \beta^{e-p+1} \mid s \in \{0,1\}, M \in \mathbb{N}, e \in \mathbb{Z},\ e_{min} \leq e \leq e_{max},\, \beta^{p-1} \leq M < \beta^p\} \\ & \cup \{(-1)^s \cdot M \cdot \beta^{e_{min}-p+1} \mid s \in \{0,1\}, M \in \mathbb{N},\, 0 \leq M < \beta^{p-1}\}\end{aligned}$$

Subnormal numbers complicate operator implementations and proofs, but their presence ensures more stable numerical programs, due to the fact that there is no gap between 0 and β^e_{min} (the underflow is gradual and not abrupt). In the following, we assume that ϕ is fixed and that $\beta = 2$.

Overflows and underflows An operation overflows if its real-valued computation results in a floating-point number having an exponent $e > e_{max}$. There are two definitions of underflow: an operation underflows if its result is a subnormal number, or it underflows if its result is rounded to zero. We assume that the first definition is used in what follows.

Rounding modes The IEEE 754 format specifies that common operations on floating-point numbers (such as addition, multiplication, square root, ...) should behave as if they were computed on real numbers and then rounded to a given floating-point number. In the case of the addition, this means that for a binary operator $\circ \in \{+, -, \times, /, \}$ on real numbers and its counterpart $\circ_{\mathbb{FP}}$ on floating-point numbers, the following equality should hold for a given rounding function rnd, and for every pair (x, y) of floating-point numbers:

$$x +_{\mathbb{FP}} y = rnd(x + y)$$

Several rounding modes are specified in the IEEE standard. The most common rounding mode is called rounding to nearest with ties to even; this function will be written RN in what follows. This basically means that a real number r is rounded to the nearest representable floating-point number. If r is equidistant from two floating-point numbers, there is a tie. In that case, the result of the rounding is the floating-point number having an even integral significand. In the following, rounding to nearest with ties to even will be used by default. However, we will also use rounding to $-\infty$ and rounding to $+\infty$, written respectively $RND_{-\infty}$ and $RND_{+\infty}$. The rounding to the left (or $-\infty$) of a real number r is the closest floating-point number less or equal to r. Similarly, rounding to $+\infty$ of a real number r is the closest floating-point number greater or equal to r.

$1 + \delta$ **abstraction** If x is a real number satisfying $2^{e_{min}} \leq |x| \leq (2^p - 1) \cdot 2^{e_{max}-p+1}$, it is possible to prove the following relative error bound: $|x - RN(x)| \leq |x|2^{-p}$. This relative error bound can be simplified and rewritten into what is called the $1 + \delta$ abstraction: $RN(x) = x \cdot (1 + \delta)$ with $|\delta| \leq 2^{-p}$. Moreover, if $\circ \in \{+, -, \times, /\}$, $|\delta| \leq 2^{-p}$ and if $x \circ_{\mathbb{FP}} y$ is a correct operation involving no underflow and no overflow:

$$x \circ_{\mathbb{FP}} y = (x \circ y) \cdot (1 + \delta)$$

The value 2^{-p} is called the machine epsilon of a floating-point format, and is usually written ϵ_M.

Special cases In addition to floating-point numbers, the IEEE 754 norm adds a few constructors:

- $+\infty$ and $-\infty$ are defined in the standard, as well as signed zeroes: there is $+0$ and -0.
- In the case of invalid operations such as $0/0$, the Not a Number (NaN) constructor is used.

On the contrary to real numbers, binary operations are not associate and not distributive anymore. Examples can be found in [16]. Moreover, Boldo et al. [3] points out that in floating-point arithmetic, even naïve optimizations such as replacing $x+0$ by x are not valid anymore (for example, $-0 + 0$ evaluates to $+0$).

In the following, we will only consider normal floating-point numbers and assume that the operations are valid, and do not underflow or overflow.

2.2 Interval analysis

This subsection presents how interval analysis works for arithmetic expressions. The idea is to soundly bound expressions by an interval. To do so, we need to define operators on intervals that will capture the behavior of computations on expression. For example, to analyze the sum of two expressions, we will need to define a similar sum operator on intervals.

Interval arithmetic In the case of interval arithmetic, we can define the addition, subtraction and multiplication of real-numbered intervals as:

$$[a;b] +^{\#} [c;d] = [a+c; b+d] \qquad [a;b] -^{\#} [c;d] = [a-d; b-c]$$
$$[a;b] \times^{\#} [c;d] = [\min(ac, ad, bc, bd), \max(ac, ad, bc, bd)]$$

The definition of the division of intervals is similar, but particular attention should be devoted to the denominator to avoid division by zero. As we want our analysis to be sound, we need to have sound interval operators. In this case, it means that for $\circ \in \{+, -, \times\}$, $\circ^{\#}$ should over-approximate $\circ$, i.e:

$$x \in [a;b] \wedge y[c;d] \implies x \circ y \in [a;b] \circ^{\#} [c;d]$$

Interval analysis Let us consider arithmetic expressions consisting in constants, variables, or binary operations between two expressions:

$$e ::= e_1 \circ e_2 \mid c \in \mathbb{R} \mid X \in \mathcal{V} \qquad \circ \in \{+, -, *, /\}$$

Here $\mathcal{V}$ is a set of variables. In the case of real interval analysis, if e is an expression, it is simple to define an interval analysis function. This function would take as argument an expression e and a map ρ from variables to intervals, and would compute an interval bounding expression e. In the following, this is written as $[\![e]\!](\rho)$, and defined recursively as follows:

$$[\![c]\!](\rho) = [c;c] \qquad [\![X]\!](\rho) = \rho(X) \qquad [\![e_1 \circ e_2]\!](\rho) = [\![e_1]\!](\rho) \circ^{\#} [\![e_2]\!](\rho)$$

The case of constants is simple, and the case of variables just uses the map ρ. The definition of the analysis of binary operations is defined in a general setting, and relies on the abstract operators $\circ^{\#}$ defined in interval arithmetic. The fundamental property that the analysis should possess is that every case that may happen when evaluating the expression to a real numbers should be captured by the interval analysis. This is the soundness of the analysis. This property can be formally stated as follows: if there is a map $\alpha : \mathcal{V} \to \mathbb{R}$ from variables to real numbers, such that for all variable v, $\alpha(v) \in \rho(v)$, and if an expression e evaluates to a value x using the mapping α, then $x \in [\![e]\!](\rho)$.

A simple example of this real interval analysis is the following: if $\rho(X) = [0;10]$ and $\rho(Y) = [0;5]$, then $[\![X+Y]\!](\rho) = [0;15]$.

Floating-point analysis In most settings, static analyzers of floating-point programs will use themselves floating-point arithmetic to perform the analysis. In that case, performing a sound interval analysis using floating-point numbers is a bit more complicated. For example, the addition of two floating-point intervals $[a,b]$ and $[c,d]$ should be defined as $[RND_{-\infty}(a+c); RND_{+\infty}(b+d)]$.

2.3 Affine arithmetic

Performing range analysis using interval arithmetic is simple and efficient. However, interval arithmetic is sometimes really imprecise: if $x \in [0;10]$, interval arithmetic can only infer that $x - x \in [-10;10]$. This is due to the fact that intervals do not keep track of relations between variables. To have a more precise analysis in that case, one way is to replace interval arithmetic by affine arithmetic [20], as affine arithmetic keeps track of linear correlations between variables.

Definition and concretization An affine form is defined as $\hat{x} = x_0 + \sum_{i=1}^{n} x_i \cdot \epsilon_i$, with the coefficients x_i being real numbers or floating-point numbers. Each epsilon represents an independent noise term, ranging between -1 and 1, so that an affine form represents the sets of numbers being computable when each noise term moves between -1 and 1. More formally, the set of values represented by an affine form can be represented using a concretization function γ:

$$\gamma(\hat{x}) = \{v \mid \exists (n_1, \ldots, n_n) \in [-1; 1]^n, v = x_0 + \sum_{i=1}^{n} x_i \cdot n_i\}$$

In the following, $v \in \hat{x}$ is a shortcut meaning $v \in \gamma(\hat{x})$. For example, the affine form $5+\epsilon_1$ represents the set of real numbers ranging from 4 to 6, so $4.5 \in 5 + \epsilon_1$.

Comparison with intervals We define the radius of an affine form $\hat{x} = x_0 + \sum_{i=1}^{n} x_i \cdot \epsilon_i$ as: $\mathrm{rad}(\hat{x}) = \sum_{i=1}^{n} |x_i|$. We can then notice that every value of this affine form is contained in the interval $[x_0 - \mathrm{rad}(\hat{x}); x_0 + \mathrm{rad}(\hat{x})]$, and that the concretization function γ could also be defined as $\gamma'(\hat{x}) = [x_0 - \mathrm{rad}(\hat{x}); x_0 + \mathrm{rad}(\hat{x})]$. Conversely, an interval $[a; b]$ is represented by the following affine form: $\frac{a+b}{2} + \frac{b-a}{2}\epsilon_1$.

As mentioned before, the advantage of affine arithmetic over interval arithmetic is its ability to keep relations between variables. If we have $x \in [0; 10]$, the only property that we can infer using interval arithmetic about $x - x$ is that $x - x \in [-10; 10]$. On the contrary, using an affine form, we can write that $x \in 5 + 5\epsilon_1$. Then, we are able to infer that $x - x \in 0 + 0\epsilon_1$, and have an exact result.

However, affine arithmetic is not strictly more expressive than intervals. For example, if $x \in [-1; 3]$ and $y \in [0; 10]$, then $x \cdot y \in [-10; 30]$. Using affine arithmetic, we can write $x \in 1 + 2\epsilon_1$ and $y \in 5 + 5\epsilon_2$. Thus, $x \cdot y \in 5 + 10\epsilon_1 + 5\epsilon_2 + 10\epsilon_1\epsilon_2$, so $x \cdot y \in [-20; 30]$.

Addition of two affine forms Let $\hat{x} = x_0 + \sum_{i=1}^{n} x_i \cdot \epsilon_i$ and $\hat{y} = y_0 + \sum_{i=1}^{n} y_i \cdot \epsilon_i$ be two affine forms. Here, the noise terms are supposed to be the same for $\hat{x}$ and $\hat{y}$. This is not a restriction since we could extend affine forms with noise terms having a coefficient 0. We define the addition of two affine forms as follows:

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

A simple induction can be used to check that the addition of two affine forms is sound:

Lemma (Soundness of $+^{\#}$)**.** *If there is an n-tuple* $(v_1, \ldots, v_n) \in [-1; 1]^n$ *with* $x = x_0 + \sum_{i=1}^{n} x_i v_i$ *and* $y = y_0 + \sum_{i=1}^{n} y_i v_i$*, then* $x + y \in \gamma(\hat{x} +^{\#} \hat{y})$.

Multiplication of two affine forms One could define the multiplication of two affine forms as: $A \times^{\#} B = x_0 \cdot y_0 + \sum_{i=1}^{n} (x_0 \cdot y_i + x_i \cdot y_j)\epsilon_i + (\sum_{i=1}^{n} x_i\epsilon_i) \cdot (\sum_{j=1}^{n} y_j\epsilon_j)$. However, this adds a lot of new noise terms (the $\epsilon_i\epsilon_j$), and prevents the design of scalable analyses. It is still possible to trade some precision to have fewer noise terms. A simple bound for the last sum is: $|\sum_{i=1}^{n} x_i\epsilon_i| \cdot |\sum_{i=1}^{n} y_i\epsilon_i| \leq \mathrm{rad}(A)\,\mathrm{rad}(B)$. Thus, a more efficient definition of the product of affine forms is: $A \times^{\#} B = x_0 \cdot y_0 + \sum_{i=1}^{n} (x_0 \cdot y_i + x_i \cdot y_j)\epsilon_i + \mathrm{rad}(A) \cdot \mathrm{rad}(B) \cdot \epsilon_{n+1}$, where ϵ_{n+1} is a fresh noise symbol. The soundness of $\times^{\#}$ is also proved by induction on the affine forms.

Inversion of an affine form When an affine form $\hat{x}$ represents only positive numbers, it is possible approximate the inverse of an affine form $\hat{x}$ by an affine form $\hat{y} = \alpha\hat{x} + \beta + \delta\epsilon_{n+1}$, where ϵ_{n+1} is a fresh noise term. An explanation is provided in appendix B. Otherwise, if $\hat{x}$ contains only negative numbers, a similar approximation is possible. If 0 is contained in $\hat{x}$, the result is undefined.

Affine arithmetic and roundoff errors Another interest of affine arithmetic is its power when analyzing roundoff errors in floating-point computations. Affine arithmetic is really precise on small ranges, such as the one found when analyzing roundoff errors [6].

2.4 Coq and HOL4, two proof assistants

Coq [5] and HOL4 [10] are two mainstream theorem provers. In both cases, the user proves interactively a goal under some given hypotheses. Theses hypotheses can be manipulated using tactics. Proving a result in a theorem prover gives a strong confidence in it, as it is impossible to skip some details, or omit some cases. However, these guarantees come at a cost: the proofs are much more involved than on paper, because each step has to be explained in the proof assistant's language. It is usually more difficult and time consuming to prove theorems in a proof assistant than on paper, but the confidence in the established result is much higher. Of course, these guarantees depend on the reliability of a theorem prover: in the presence of bugs, it might be possible to prove false results. In both Coq and HOL4, the kernel that is used to perform the proofs is small, so the chance there is a bug is rather small.

This paragraph tries to highlight some discrepancies between Coq and HOL. A more detailed comparison can be found in [21]. Coq is based on intuitionistic logic, where as HOL4 uses classical logic. This means that in Coq and contrary to HOL4, the law of excluded middle $p \vee \neg p$ does not hold. Coq also distinguishes proposition from booleans. This means that sometimes we needed to define a boolean equality function for new defined objects. On the contrary to HOL4, Coq supports dependent types, which I used in some of my prototypes of formalization of affine arithmetic. This allows for example to define a subset type B of rationals between -1 and 1. Every element $b \in B$ then carries a value and a proof that this value is between -1 and 1.

I always started proving theorems in Coq, and then porting them to HOL4. Coq usually involves more complex designs and proofs, that are then easily portable to HOL4. Currently, the development is really similar.

2.5 Daisy, a compiler for reals

Daisy[2] is a framework developed around the concept of compiler for real-valued program mentioned before. Input programs are also written in Scala, using a real-valued specification language. A program is a collection of functions, and each function consists in a precondition, a body and a postcondition. Preconditions give mandatory ranges for the function parameters. Without these ranges, it would be impossible to compute the relative errors used in the $1 + \delta$ abstraction. Postconditions enforcing a maximal roundoff error on a computation can also be specified, but are not mandatory. A function body consists in a succession of variable bindings, returning an expression. An example of input is given in Fig. (1a). The output consists in a program using floating-point arithmetic and written in Scala (but the backend could be ported to another language easily); an example of mixed-precision output is given in Fig. (1b). In addition to this program, Daisy displays an upper bound on the difference of the results returned by the ideal function and the floating-point one. That is, given a function $f : \mathbb{R} \to \mathbb{R}$, and an input domain $D \subset \mathbb{R}$ (defined by the precondition), if $\tilde{f}$ is the function generated, Daisy returns an upper bound of $\max_{x \in D} \left| f(x) - \tilde{f}(RN(x)) \right|$.

The analysis performed by Daisy is actually split into two parts:

- One is the real range analysis, computing the real-valued range of every expression.

[2] `https://gitlab.mpi-sws.org/AVA/daisy-public`

```
import daisy.lang._
import Real._

object Sine {
  def sine(a:Real) = {
    require(-1 <= a && a <= 1)
    val three = - a*a*a / 6
    val five = a*a*a*a*a / 120
    a + three + five
  }
}
```

(a) Input code

```
object Sine {
  /*@pre : ((-1. <= a) && (a <= 1.))*/
  def sine(a : Double) : DoubleDouble = {
    val three : DoubleDouble = (((-(a) * a) * a) / 6)
    val five : Double = (((((a * a) * a) * a) * a) / 120)
    ((a + three) + five)
  }
}
```

(b) Generated code (in mixed-precision)

Figure 1: Running Daisy on a Taylor expansion of sine

– Given the result of the previous analysis, the roundoff errors of every expressions are then computed using the $1 + \delta$ abstraction mentioned in Section 2.1.

Splitting this analysis in two parts yields more precise results than doing an analysis keeping track of only the floating-point range, because the computation of the error bound of an expression actually depends on the real ranges and roundoff errors of its subexpressions. To compute the ranges, the analysis can rely on interval arithmetic or affine arithmetic.

Limitations For now, Daisy does not support control-flow statements such as `if` and `while`. This is due to the fact that if a boolean condition compares two floating-point numbers, roundoff errors may create a divergence in the flow of the program: the real-valued program may enter the `if` branch, but the floating-point program may enter the `else` branch. Concerning `while` statements, Daisy would also need to find an upper-bound on the number of iterations. This is a well-known issue, and Darulova and Kuncak [6] explain why tackling divergence is challenging. Daisy does not support inter-procedural analysis of functions.

In Daisy's current state, overflows, NaNs, and ranges containing only subnormals are considered to be faults, and stop the analysis. Daisy assumes that operations do not overflow. However, thanks to the real range analysis, checking that no overflow happen should be simple to implement. The use of the $1 + \delta$ abstraction is sound on ranges of floating-point numbers containing at least one normal number.

Daisy uses rationals in its implementation. This avoids having to deal with different rounding modes to ensure the soundness of the analysis, but it is quickly inefficient on big computations.

Mixed-precision Daisy supports some mixed-precision tuning of programs. It is based on a delta-debugging algorithm used in Precimonious [18]: at first, all variables use the highest available precision. Then, some chosen variables are given a lower precision, and the algorithm checks if the modified program is precise enough. After this, the algorithm backtracks or continues. For debugging purposes, I implemented a small parser within Daisy that can read a (maybe partial) assignment of precision for variables.

2.6 Verifying Daisy's analyses

The formal verification of Daisy consists in a verified result checker. When Daisy is run, it generates a certificate encoding the result of its analysis. Then, the certificate checker (written in both Coq

and HOL4) is run on this certificate. It has been formally proved that if the certificate checker accepts a certificate then the performed analysis was sound.

Following the execution of the checker, we will first see what is encoded in a certificate, before seeing what is checked, and how these checks relate to the soundness of the analysis.

Certificates When Daisy has finished the analysis of a given function, it stores the results in a certificate. Each certificate consists in:

- definitions of the analyzed expressions (and variable bindings) used in the input function;
- a map from these expressions to their corresponding real ranges and roundoff errors;
- preconditions giving for each input variable of the function, the range of this variable.

The certificate ends with a theorem stating that the checking functions should return true. When a proof assistant is run on a certificate, it uses the definitions of the expressions and the maps to compute the checking function. The example shown in Fig. (3) will be detailed in the next section.

Checking functions The certificate checker consists in two main parts, similar to the analysis performed by Daisy. That is, the certificate checker first verifies that the real ranges (this has been called interval checker) and that the error ranges (called error checker) have been computed correctly in Daisy. Currently, both interval and error checkers analyze again the expressions, and return true if their results are stricter or equal to what was obtained by Daisy. Each checker function runs recursively on expressions to check the validity of the ranges or the error bounds.

Deducing the soundness of the analysis In this part, we show how to deduce that an analysis was sound given the positive result of the checking functions.

Proving the soundness property requires being able to capture the behavior of computations in Scala. To do this, an inductive predicate `eval_exp` was defined. This relation states that a given expression e under an environment E mapping variables to values may evaluate to a floating-point number f (assuming a fixed floating-point precision of 53 bits). It uses the $1 + \delta$ abstraction mentioned before and is thus non-deterministic. In both Coq and HOL4, this would have been written as `eval_exp e E f`. A simpler inductive predicate `eval_exp_real` was also defined to express real-valued computations.

We also need to use interval arithmetic. In both Coq and HOL4, real-valued interval arithmetic has been formalized, and its operations have been proved sound.

Then, the soundness proof concerning the interval analysis is written as:

```
Theorem validIntervalbounds_sound e absenv P E:
  validIntervalbounds e absenv P = true →  eval_exp_real E e vR →
  fst (fst (absenv f)) <= vR <= snd (fst (absenv f)).
```

In that theorem, `e` is an arithmetic expression, the analysis result `absenv` maps expressions to a range containing the real evaluated expression and a computation error. `P` is the precondition on the ranges of the input variables and `E` is an environment mapping variables to real numbers. The theorem states that if the range checker returns true and if the expression `e` evaluates to `vR` in the reals, then `vR` is contained in the interval given by the analysis result. We can see that this is a soundness theorem: every possible real-valued computation is captured by the result of the analysis.

The soundness statement of the error analysis is:

```
Theorem validErrorbound_sound e absenv P E1 E2:
  approxEnv E1 absenv E2 →  validIntervalbounds e absenv P = true →
  validErrorbound e absenv = true →  eval_exp_real E1 e vR →  eval_exp E2 e vF →
  | vR − vF| <= snd (absenv e).
```

It states that if the range checker and the error checker return true, and if the expression e evaluates to vR in real-valued precision and may evaluate to vF in floating-point arithmetic, then the absolute value of $vR - vF$ is less than the error bound found by Daisy, and given in the second component of the analysis result `absenv`.

Variable bindings The soundness results presented above showed the soundness of the range and error checker for expressions. However, Daisy also supports `let`-bindings, binding variables with expressions into either another let-binding or an expression. The certificate checker also supported the checking of these let-bindings.

Due to the presence of the variable bindings, two different environments `E1` and `E2` were needed in the soundness theorem of the error analysis: in the soundness proof of the error validation for commands, the `let`-bound variables do not have the same values depending on the real or floating-point evaluation, and thus the environments are just approximating each other.

3 Contribution: extending the formal checking to mixed-precision

When I arrived, this formalization was already developed for single-precision floating-point arithmetic analyses using only intervals. My main task during this internship has been to extend the certificate checking to handle mixed-precision computations. To perform this task, I had to implement new features and port a few others:

- I encoded the different floating-point formats into what I called machine types.
- I generalized the definition of arithmetic expressions.
- I updated the semantics of `eval_exp` and its variable-bindings counterpart.
- I designed a typing phase needed to know the floating-point precision used in each expression.
- Then, I generalized the checkers of ranges and errors, and their soundness proofs.
- The final step was to update the certificate generator within Daisy, to support the updates of the definitions.

Following the previous development, I first focused on generalizing the proofs in Coq, before porting everything to HOL4.

In the following, I will use as a running example a computation of the Taylor expansion of sine around zero. The code given in input to Daisy is presented in Fig. (1a), and the output is in Fig. (1b). In this example, I suppose that the floating-point format of each variable is fixed to double-precision, except variable `three` which has a quadruple-precision format. Parts of the certificate (for the Coq checker) are presented in Fig. (3), and will be explained in the coming sections.

3.1 Machine Type

In the uniform-precision formalization, the floating-point format of each variable was fixed to double-precision and its corresponding machine epsilon was hard-coded. To change this, I added a representation of floating-point formats called machine types. It currently encodes the machine epsilon, as the other parameters defined by a floating-point format are not used currently. In both Coq and HOL4, the machine type is described as a sum type representing either real-based precision, 32, 64, 128 or 256 bits long floating-point format, respectively written `M0, M32, M64, M128, M256`. The real-based precision machine type is needed because the `eval_exp_real` predicate is just a special case of `eval_exp`. The mapping from machine types to their corresponding machine epsilon is performed by a function called `meps`. In Scala, computations involving two different machine precision yields a result having the highest precision of the two. To specify this behavior, I defined a lattice on the machine types, consisting in total order $\sqsupseteq$ and a join operator $\sqcup$. The order is: `M256` $\sqsupseteq$ `M128` $\sqsupseteq$ `M64` $\sqsupseteq$ `M32` $\sqsupseteq$ `M0`. Then, the join of two elements is the maximum element with respect to the order defined above. This ensures the crucial property that if the join of m_1 and m_2 is `M0`, then both m_1 and m_2 are also `M0`. As mentioned in section 2.4, I also needed to define a boolean equality operator in Coq.

At first, I defined the machine type to be just a positive rational, but this created much more difficult proofs. Using a sum type is much easier, as some proofs can then be done automatically by case analysis. For example, the property $m_1 \sqcup m_2 = \texttt{M0} \implies m_1 = m_2 = \texttt{M0}$ can be proved quickly in Coq by using the `destruct` tactic on m_1 and m_2. This tactic creates subgoals for each constructor of m_1 and m_2, and most of them will entail a contradiction.

3.2 Generalizing arithmetic expressions

In the previous formalization, arithmetic expressions did not explicitly depend on a machine type. Now, constants are defined as pair, specifying the value of the constant and its precision.

I also needed to add a new constructor to the expressions, called `Downcast`: this operator forces the cast of an expression to a lower precision. On the contrary to an increase in precision, a `Downcast` has to be explicit as it may introduce new roundoff errors.

Thus, expressions can now be:

- a variable whose identifier is a natural number;
- a rational constant in a given floating-point format;
- a unary expression consisting in a unary operator ($-$ or $/$) and an expression;
- a binary expression;
- a downcast of an expression, consisting in a machine type and an expression.

The formal definitions of the expressions are presented in Fig. (2). Fig. (3a) shows how $-a^3$ is defined in a certificate, where the natural number 0 represents variable a.

3.3 Generalizing eval_exp

The `eval_exp` predicate is used to state how an expression is computed in Scala. As mentioned in the previous section, this predicate is required to express the soundness of the certificate checker. I updated this predicate to take into account mixed-precision computations.

```
Inductive exp: Type :=
  Var: nat →  exp
| Const: mType →  Q →  exp
| Unop: unop →  exp →  exp
| Binop: binop →  exp →  exp →  exp
| Downcast: mType →  exp →  exp.
```

(a) In Coq

```
val _ = Datatype ‘
  exp = Var num
      | Const mType ’v
      | Unop unop exp
      | Binop binop exp exp
      | Downcast mType exp‘
```

(b) In HOL4

Figure 2: Formal definition of the expressions

```
Definition ExpVara0 :exp Q := Var Q 0.
Definition UMinExpVara0 :exp Q := Unop Neg ExpVara0.
Definition MultUMinExpVara0ExpVara0 :exp Q := Binop Mult UMinExpVara0 ExpVara0.
Definition MultMultUMinExpVara0ExpVara0ExpVara0 :exp Q := Binop Mult
    MultUMinExpVara0ExpVara0 ExpVara0.
```

(a) Defining $-a^3$

```
Definition defVars_sine :(nat →  option mType) := fun n ⇒
if n =? 0 then Some M64 else if n =? 3 then Some M128 else if n =? 4 then Some M64 else None.
```

(b) Defining the type map

```
Definition absenv_sine :analysisResult :=
fun (e: exp Q) ⇒
if (expEqBool e ExpVara0) then (((−1) /(1) ,  (1) /(1) ),  (1) /(9007199254740992) )
else if (expEqBool e UMinExpVara0) then (((−1) /(1) ,  (1) /(1) ),  (1) /(9007199254740992) )
else if (expEqBool e MultUMinExpVara0ExpVara0) then
  (((−1) /(1) ,  (1) /(1) ),(2433889152438200721089647796551 69)
    /(730750818665451459101842416358141509827966271488) )
else if (expEqBool e MultMultUMinExpVara0ExpVara0ExpVara0) then
  (((−1) /(1) ,  (1) /(1) ),(32910091146424128150607570305662412414463026960948689432960040961)
    /(59285549689505892056868344324448208820874232148807968788202283012051522375647232) )
else ((0/1,0/1)  ,0/1) .
```

(c) Defining the analysis result (in the Coq code, the / are actually #)

```
Theorem ErrorBound_sine_Sound :
  CertificateCheckerCmd final_command absenv_sine thePrecondition_sine defVars_sine = true.
Proof.
  cbv; auto.
Qed.
```

(d) Final result

Figure 3: Excerpts of Daisy's certificate when run on the example of Fig. 1a

The updated `eval_exp` predicate takes into account two environments: one mapping variables to values, and one mapping variables to machine types. For the sake of readability, the predicate `eval_exp` is renamed as $\Downarrow$, so that `eval_exp Env e v m` is equivalent to `e`, $Env \Downarrow v, m$ (here, e is an expression, Env is consists in the two environments, v is a number and m is a machine type). The main rules are presented as inference rules in Fig. (4). Let us phrase the case of a

$$\frac{E\ n = (v,\ m)}{\texttt{Var n}, E \Downarrow v, m}\ \text{Var} \qquad \frac{|\delta| \leq \text{meps}(m)}{\texttt{Const m n}, E \Downarrow n \cdot (1+\delta), m}\ \text{Const}$$

$$\frac{\texttt{f1}, E \Downarrow v1, m1 \qquad \texttt{f2}, E \Downarrow v2, m2 \qquad |\delta| \leq \text{meps}(m1 \sqcup m2) \qquad ((op = /) \implies v2 \neq 0)}{\texttt{Binop op f1 f2}, E \Downarrow (v1\ \text{op}\ v2) \cdot (1+\delta), m1 \sqcup m2}\ \text{Binop}$$

$$\frac{\texttt{f1}, E \Downarrow v1, m1 \qquad m1 \sqsupseteq m \qquad |\delta| \leq \text{meps}(m)}{\texttt{Downcast m f1}, E \Downarrow v1 \cdot (1+\delta), m}\ \text{Downcast} \qquad \frac{\texttt{f}, E \Downarrow v, m}{-\texttt{f}, E \Downarrow -v, m}\ \text{Unary} -$$

Figure 4: The new definition of eval_exp

binary operator: given an environment E (mapping each variable to a value and a machine type), two expressions f_1 and f_2 and a binary operator op $\in \{+, -, \times, /\}$, the expression f_1 op f_2 may evaluate to $(v_1\ \text{op}\ v_2) \cdot (1+\delta)$ in machine precision $m_1 \sqcup m_2$ if the following conditions are satisfied:

- given the same environment E, f_1 evaluates to value v_1 in machine precision m_1;
- and similarly for f_2, but in the case of a division we should have $v_2 \neq 0$;
- $|\delta|$ is less than the machine epsilon of the new machine precision $m_1 \sqcup m_2$.

As mentioned before, the machine type of f_1 op f_2 is the highest precision among $m1$ and $m2$, i.e. $m1 \sqcup m2$. The other inference rules can be read similarly.

3.4 Generalizing variable bindings

Variable bindings are also updated: they now specify the machine type used for each bound variable. Then, every function and theorem presented in the next sections will be lifted to a command-based version, ultimately relying on its expression-based counterpart. The extension is mostly technical, I will not detail it in the next sections.

3.5 Typing

The mixed-precision error checker requires to know the maximal roundoff error that can be created by each expression. This roundoff error is bounded by the machine epsilon, itself depending on the machine type. Thus, we need to know for each expression, its corresponding machine type. To that end, I implemented a new typing validation phase in the certificate checker, creating a map from expressions to machine types, and checking that this map is correct. The current version of this typing validation follows the same structure as the interval and error validation files: first, checking functions are defined and check that a given type map from expressions to machine types is valid. Second, soundness theorems are proved: they state that if an expression e and typing map Γ are validated, and if e evaluates to a floating-point v in machine precision m (i.e. $\texttt{e}, E \Downarrow v, m$), then the type of e should be m (i.e. $\Gamma(e) = m$).

In the current version, I also defined a function (in Coq and HOL4) computing the type of each expression, given the type of each variable. No explicit typing map is currently stored during Daisy's execution, but it would be more efficient to compute it during the analysis, store this typing map in the certificate and check it. Currently, the machine type of each variable is given by the certificate

using a partial map. An example is given in figure (3b): we can see that the initial variable a and variable `five` have machine precision `M64`, but that variable `three` has machine precision `M128` corresponding to Scala's `DoubleDouble` type.

This part of the generalization has been the most difficult one. Starting to design a new validation phase (similar to both the interval and error validation phases) did not seem to be too difficult at first, as I was starting to feel sufficiently at ease with Coq. However, my first design involved a function computing the typing map directly. I then wanted to prove that the computed typing map was valid. This entailed reasoning on types of subexpressions, and it turned out to be too complicated to prove. I spent a few weeks trying to sidestep this difficulty, without success: when the typing function was sound, I was unable to prove the soundness of the error validation and conversely when the properties needed in the error validation proofs where met by the typing function, I was unable to prove its soundness. Finally, the solution was to use a structure similar to the range and error validation phases, with a checker function, that does not try to compute a typing map, but only checks if this typing map is valid.

3.6 Generalizing the interval validation

The interval validation, checking that the real-valued ranges of each expression are sound, was quite easy to port. The ranges are specified in certificate using a map. An excerpt of this map is shown in Fig. (3c). This map associates expressions to pairs of intervals and roundoff errors. As mentioned in Section 2.4, the boolean equality between expressions is defined in Coq by a function called `expEqBool`. The interval checker will only use the intervals given by the map, and proceeds recursively on the expression it received.

The extension of the interval checker consists in adding the case of the `Downcast`. By definition, this analysis expresses real-valued computations, so checking the range of an expression `Downcast m f` consists in checking that the range found for `Downcast m f` and `f` are the same, and recursively checking the ranges for `f`.

The proof of soundness is done by induction on the expressions. I had to prove the new case of `Downcast`, but the rest of the proof cases were mostly unchanged.

3.7 Generalizing the error validation

The error checker generalization was more involved: every function and result is about roundoff errors, and had to be updated to use the good machine epsilon, depending on the machine type of each expression. The function checking the error bounds for expressions is changed in two ways:

- The checking function now takes one more argument, a type map mapping every expression to a machine type.
- The checking function proceeds by recursion on a given expression. Compared to the uniform-precision formalization, the expression type has one more constructor being `Downcast`. The implementation of this new case is described in what follows. We start by stating a theorem, that will be used to check the case of `Downcast`. This theorem links the error bounds of an expression e with the `Downcast` of this expression.

Theorem 1. *Assuming that:*

$$e_R, E_1 \Downarrow n_R, M0 \qquad \texttt{e}, E_2 \Downarrow n_F, m \qquad \texttt{(Downcast m' (Var 1))}, E_2[1 \mapsto (n_F, m)] \Downarrow r_F, m'$$

The following bound holds: $|n_R - r_F| \leq |n_R - n_F| + |n_F| \cdot \text{meps}(m')$.

The last assumption of the theorem cannot be as simple as `(Downcast m' e)`, $E_2 \Downarrow r_F, m'$: by non-determinism of $\Downarrow$, it would not be possible to relate this hypothesis with the evaluation of `e` in floating-point. Thus, I followed the solution used in similar cases in the uniform-precision certificate checker, storing the result of the evaluation of `e` into a variable.

The proof of this theorem has been formalized in Coq (in 18 lines) and HOL4 (in 14 lines). A paper proof is presented in appendix A.

Now, we can state the condition that will be used in the error checker. Suppose we have an expression `f = Downcast m' e`. We also suppose that according to the certificate, `analysisResult e = ([le, he], erre)`. This means that the real-valued expression `e` is contained in the interval $[le; he]$, and that `e` evaluated in floating-point arithmetic is contained in $[le - erre; he + erre]$. If `analysisResult f = ([lf; hf], errf)`, then the condition that is checked is:

$$erre + \max(|le - erre|, |he + erre|) \cdot \text{meps}(m') \leq errf$$

This bound is really similar to what was stated in the theorem before. Moreover, this condition looks reasonable, because the soundness theorem is still provable. As described in the last section, the soundness theorem of the error validation claims that when both checker function return true, then the error found is sound. This theorem is proved by induction on the expressions. In the case of `Downcast`, we can now see why it holds. We know that:

- $|nR - nF| \leq erre$ (by induction hypothesis);
- $nF \in [le - erre, he + erre]$, so that $|nF| \leq \max(|le - erre|, |he + erre|)$ (using the interval validation and the bound of the previous point).
- $erre + \max(|le - erre|, |he + erre|) \cdot \text{meps}(m') \leq errf$ (using the fact that the error checker returns true)

Thus, we can conclude that $|nR - rF| \leq errf$.

3.8 Final checking function

The last part of a certificate is a theorem stating that the certificate checker validates the defined certificate. It takes as argument the `final_command`, which represents the whole analyzed function, the analysis result `absenv_sine`, the precondition of the function `thePrecondition_sine` and the type map for the variables `defVars_sine`. One example is given in Fig. (3d). The only part of the certificate not presented in Fig. (3) is the precondition which maps parameter variables of the function (in our case, a), to constraints on these variables (here, $a \in [-1; 1]$). If a certificate is validated, then we know by the soundness proofs done before that the analysis performed was sound.

3.9 Implementation details

The formalization of mixed-precision certificate-checking is now finished, although some optimizations could be implemented. I also modified the certificate generator in Daisy to handle mixed-precision certificate generation. One of the difficulties in this work was to understand each proof and function, as the formalization starts to be quite big: when I arrived, the Coq formalization took roughly 5000 lines of code, and roughly 6000 lines of code for the HOL4 formalization. Another was to use theorem provers: I knew a bit the Coq proof assistant, but I discovered many of its features and subtleties during my internship. I had no previous knowledge of HOL4, and learning to use a

new theorem prover always takes some time for me. Finally, the difficult point of the generalization in itself was to introduce the typing validation, and get a correct and usable formalization of this typing. As of mid-May 2017, the extension of the formalization to mixed-precision takes 6500 lines of code in Coq and 7100 lines in HOL4.

The formalization is still limited for now: interval arithmetic is the only supported format, but Daisy is capable of analyzing programs using affine arithmetic as well. Another issue is that currently, the certificate checking has a computational cost similar to performing the analysis.

4 Formalization of affine arithmetic

After the formalization of mixed-precision, interval-based certificate checking, we decided to start formalizing affine-arithmetic [20]. This would allow more precise analyses to be also verified, because affine arithmetic is able to track relationships between variables. Currently, the formalization of affine arithmetic consists in:

- a definition of what an affine form is;
- a definition of the concretization function γ mentioned in section 2.3;
- a function computing the addition of two affine forms;
- a function computing the multiplication of two affine forms;
- proofs that these functions are sound, as defined in section 2.3.

For now, the formalization is only written in Coq, although the port to HOL4 should not involve any issues. The formalization is also not yet linked with the certificate checker.

I started by defining in Coq what an affine form is. An affine form is encoded as a list of coefficients, starting by the constant term. The noise terms are each represented by an index and a coefficient (corresponding to the i and x_i in the definitions of the affine forms above). For example, the affine form $5 + 3\epsilon_2$ is written `Noise_t 2 (3/1) (Const_t 5)`. The current development implicitly assumes that the noise terms are sorted by decreasing index, and it maintains this property through the functions I defined. This index property has not been explicitly defined yet, but it possible to define normalized affine forms, where each noise coefficient is non-zero (or even positive), and where the indices of the noise terms are sorted.

To encode the concretization function, we need to enforce that the n-tuples or the map used to evaluate an affine form into a real number has elements between -1 and 1 only. In Coq, I used a subset type that I called `bounded_values`. It consists of a rational q between -1 and 1, and a proof that q is between -1 and 1.

The concretization function then takes an affine form $\hat{a} = a_0 + \sum_{i=1}^{n} a_i\epsilon_i$, a rational v and a partial map M from indices to `bounded_values`. It is defined as follows: v is in the concretization of $\hat{a}$ when $v = \text{eval}(\hat{a}, M)$, with $\text{eval}(\hat{a}, M) == a_o + \sum_{i=1}^{n} a_i M[i]$. That is, v is a value represented by $\hat{a}$ if the evaluation of $\hat{a}$ using the noise terms of M is v.

Defining the addition of two affine forms uses an approach really similar to the merging of two sorted lists (as we want to keep the indices of the new affine form in decreasing order). However, I encountered some difficulties, because Coq was not able to infer that the definition of the addition of affine forms terminates. By default, Coq only accepts definition of functions that terminate according to its heuristics. Every recursive function that is defined in Coq should call itself back with a decreasing parameter. For example, if the recursive function is manipulating a list, Coq will accept the definition only if the function progresses in the list during its recursive call. To continue

Tool	Daisy	Fluctuat	FPTaylor	Gappa
Automation	Full	Full	Full	Partial
Formal verification	Certificates in Coq/HOL4	No	Certificates in HOL Light	Outputs proofs in Coq
Numerical domain	Intervals	Zonotopes	N/A	Intervals
Transcendental functions	No	No	Yes	Yes
Mixed-precision	Yes	Yes	No	No
Input language	Scala, using an abstract class	Subset of C	Custom	Custom
Variable bindings	Yes	Yes	No (inlining)	Yes
Control-flow support	No	Yes, divergence in option	No	No
Overflow/underflow support	Partial underflow support	Yes	Subnormals	Subnormals

Table 1: Summary of different frameworks

with lists, when merging two sorted lists l_1 and l_2, the merging function may recurse with the tail of l_1 and l_2, or l_1 and the tail of l_2, or the tail of both l_1 and l_2. This merging function terminates, but Coq is unable to infer that automatically, because only one out of two arguments is decreasing (and currently Coq cannot infer by itself that the pair (l_1, l_2) is a decreasing parameter). The same problem was happening with the addition of affine forms. I experimented with different approaches to prove the termination. In the end I chose to use the *Function* module. In that case, I wrote the function `plus_aff` I wanted, and I defined a termination parameter. Then, the module generates proof obligations the user should fulfill to get a well-defined function. These proof obligations consisted in proving that each recursive call of the function decreases the termination parameter. One issue I had with this approach is that the *Function* module was annotating the function a lot. This was an issue in the proof of soundness, because the unfolding of a computation `plus_aff` was really difficult. Later, I discovered that during proofs, it was possible to remove the annotations created by the *Function* module. Then, I was able to prove the soundness of the defined addition using functional induction: this simplified my proofs a lot.

5 Related work

Static analyzers I am not aware of any other formal development supporting mixed-precision floating-point static analysis. The following tools are compared with Daisy in Table 1.

Fluctuat [7] is a state-of-the-art static analyzer by abstract interpretation. It supports advanced analyses using a combination of domains, one of them being affine arithmetic. On the contrary to the other tools, it has not been formally verified. It is included to see what functionalities are still missing in the current formal developments.

FPTaylor [19] encodes the error analysis as a global optimization problem, and then solves approximately this problem. It usually finds tighter bounds than another fully automatic analyzers, but the analysis time is longer [6].

Gappa [2] is rather a high-level proof assistant than a static analyzer. It can prove (with the

help of the user) precise results. It can also be used as a tactic inside Coq to solve simple goals.

Formal verification of numerical abstract domains Concerning formal verification of numerical abstract domains, a lot of work has been presented in the last few years. The closest to what we want to achieve is the formalization of some operations of affine arithmetic in PVS [15]. However, no Coq or HOL4 formalization of affine arithmetic exist, and [15] does not supports inversion of affine forms, and other functions defined on the zonotope abstract domain [11, 12]. The VPL library [8] provides a library for manipulating convex polyehdra. It is implemented in OCaml, but a Coq fronted is able to check the soundness of the operations performed. In [14], abstract domains such as intervals, and linear congruences are formalized. Recently, the Octagon abstract domain has also been formalized in Coq [13].

6 Conclusion

Daisy is a tool aiming at easily designing numerical programs having no precision nor performance issue due to a misunderstanding of floating-point arithmetic by the end-user. I implemented support of formally-verified certificate-checking for mixed-precision arithmetic, in the case of interval arithmetic based analysis. I also started including support for the formalization of affine arithmetic based analysis.

During my working time, I spent some time on other tasks.

- The lab interviewed 14 candidates for faculty jobs. This was a good opportunity to attend interesting talks ranging over a lot of different topics in computer science.
- I attended reading groups in programming languages and verification.
- I took a German class offered by the institute.
- I helped organize the Girl's Day event, which is a national event in Germany aiming at presenting to high-school girls domains where they are underrepresented[3]. In our case, our goal was to show them what research in computer science is. To this end, we prepared some activities where the pupils would discover and experiment with new concepts related to computer science. In particular, I helped to adapt an activity taken from the computer science field guide[4], introducing the concept of finite-state automaton.
- My supervisor also taught a static analysis course at the Master's level. One of the parts of this course was about abstract interpretation. I helped design and I conducted a practical session on a static analyzer called Interproc[5]. The goal of this session was to show to the students that the analyses they have seen during the lecture can be implemented, and to discover a few key features of static analysis by abstract interpretation.

References

[1] IEEE standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, Aug 2008. doi:10.1109/IEEESTD.2008.4610935.

[3] `https://www.girls-day.de`

[4] `http://csfieldguide.org.nz/`

[5] `https://pop-art.inrialpes.fr/interproc/interprocweb.cgi`

[2] Sylvie Boldo, Jean-Christophe Filliâtre, and Guillaume Melquiond. Combining coq and gappa for certifying floating-point programs. In Jacques Carette, Lucas Dixon, Claudio Sacerdoti Coen, and Stephen M. Watt, editors, *Intelligent Computer Mathematics, 16th Symposium, Calculemus 2009, 8th International Conference, MKM 2009, Held as Part of CICM 2009, Grand Bend, Canada, July 6-12, 2009. Proceedings*, volume 5625 of *Lecture Notes in Computer Science*, pages 59–74. Springer, 2009. doi:10.1007/978-3-642-02614-0_10.

[3] Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. Verified compilation of floating-point computations. *J. Autom. Reasoning*, 54(2):135–163, 2015. doi:10.1007/s10817-014-9317-x.

[4] Wei-Fan Chiang, Mark Baranowski, Ian Briggs, Alexey Solovyev, Ganesh Gopalakrishnan, and Zvonimir Rakamaric. Rigorous floating-point mixed-precision tuning. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 300–315. ACM, 2017.

[5] Thierry Coquand, Gérard Huet Huet, Christine Paulin, et al. The Coq Proof Assistant, 2016. `https://coq.inria.fr`.

[6] Eva Darulova and Viktor Kuncak. Towards a compiler for reals. *ACM TOPLAS*, 39(2):8:1–8:28, March 2017. doi:10.1145/3014426.

[7] David Delmas, Eric Goubault, Sylvie Putot, Jean Souyris, Karim Tekkal, and Franck Védrine. Towards an industrial use of FLUCTUAT on safety-critical avionics software. In María Alpuente, Byron Cook, and Christophe Joubert, editors, *Formal Methods for Industrial Critical Systems, 14th International Workshop, FMICS 2009, Eindhoven, The Netherlands, November 2-3, 2009. Proceedings*, volume 5825 of *Lecture Notes in Computer Science*, pages 53–69. Springer, 2009. doi:10.1007/978-3-642-04570-7_6.

[8] Alexis Fouilhé, David Monniaux, and Michaël Périn. Efficient generation of correctness certificates for the abstract domain of polyhedra. In Francesco Logozzo and Manuel Fähndrich, editors, *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*, volume 7935 of *Lecture Notes in Computer Science*, pages 345–365. Springer, 2013. doi:10.1007/978-3-642-38856-9_19.

[9] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, 1991. doi:10.1145/103162.103163.

[10] Mike Gordon et al. The Hol Theorem Prover, 2017. `https://hol-theorem-prover.org/`.

[11] Eric Goubault and Sylvie Putot. Static analysis of numerical algorithms. In Kwangkeun Yi, editor, *Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006, Proceedings*, volume 4134 of *Lecture Notes in Computer Science*, pages 18–34. Springer, 2006. doi:10.1007/11823230_3.

[12] Eric Goubault and Sylvie Putot. Perturbed affine arithmetic for invariant computation in numerical program analysis. *CoRR*, abs/0807.2961, 2008.

[13] Jacques-Henri Jourdan. Sparsity preserving algorithms for octagons. *Electr. Notes Theor. Comput. Sci.*, 331:57–70, 2017. doi:10.1016/j.entcs.2017.02.004.

[14] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. A formally-verified C static analyzer. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 247–259. ACM, 2015. doi:10.1145/2676726.2676966.

[15] Mariano M. Moscato, César A. Muñoz, and Andrew P. Smith. Affine arithmetic and applications to real-number proving. In Christian Urban and Xingyuan Zhang, editors, *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*, volume 9236 of *Lecture Notes in Computer Science*, pages 294–309. Springer, 2015. doi:10.1007/978-3-319-22102-1_20.

[16] Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser, 2010. doi:10.1007/978-0-8176-4705-6.

[17] Takeshi Ogita, Siegfried M. Rump, and Shin'ichi Oishi. Accurate sum and dot product. *SIAM J. Scientific Computing*, 26(6):1955–1988, 2005. doi:10.1137/030601818.

[18] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H. Bailey, Costin Iancu, and David Hough. Precimonious: tuning assistant for floating-point precision. In William Gropp and Satoshi Matsuoka, editors, *International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13, Denver, CO, USA - November 17 - 21, 2013*, pages 27:1–27:12. ACM, 2013. doi:10.1145/2503210.2503296.

[19] Alexey Solovyev, Charles Jacobsen, Zvonimir Rakamaric, and Ganesh Gopalakrishnan. Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. In Nikolaj Bjørner and Frank S. de Boer, editors, *FM 2015: Formal Methods - 20th International Symposium, Oslo, Norway, June 24-26, 2015, Proceedings*, volume 9109 of *Lecture Notes in Computer Science*, pages 532–550. Springer, 2015. doi:10.1007/978-3-319-19249-9_33.

[20] Jorge Stol and Luiz Henrique De Figueiredo. Self-validated numerical methods and applications. In *Monograph for 21st Brazilian Mathematics Colloquium, IMPA, Rio de Janeiro*. Citeseer, 1997.

[21] Freek Wiedijk. Comparing mathematical provers. In Andrea Asperti, Bruno Buchberger, and James H. Davenport, editors, *Mathematical Knowledge Management, Second International Conference, MKM 2003, Bertinoro, Italy, February 16-18, 2003, Proceedings*, volume 2594 of *Lecture Notes in Computer Science*, pages 188–202. Springer, 2003. doi:10.1007/3-540-36469-2_15.

A Proof of theorem 1

This is the proof of theorem 1, recalled below. This proof is also formalized in the certificate checker.

Theorem Assuming that:

$e_R, E_1 \Downarrow n_R, M0$ `e`, $E_2 \Downarrow n_F, m$ `(Downcast m' (Var 1))`, $E_2[1 \mapsto (n_F, m)] \Downarrow r_F, m'$

The following bound holds: $|n_R - r_F| \leq |n_R - n_F| + |n_F| \cdot \text{meps}(m')$.

Proof.

$$\begin{aligned} |n_R - r_F| &= |n_R - n_F + n_F - r_F| \\ &\leq |n_R - n_F| + |n_F - r_F| \end{aligned}$$

Now, we only need to prove that $|n_F - r_F| \leq |n_F| \cdot \text{meps}(m')$. By definition of $\Downarrow$ (in Fig. (4)), we know that `(Downcast m' (Var 1))`, $E_2[1 \mapsto (n_F, m)] \Downarrow r_F, m'$ implies the following:

- $m_x \sqsupseteq m'$;
- $|\delta| \leq \text{meps}(m')$;
- `(Var 1)`, $E_2[1 \mapsto (n_F, m)] \Downarrow x, m_x$ with $x \cdot (1 + \delta) = r_F$.

The last point can be further simplified to get that `(Var 1)`, $E_2[1 \mapsto (n_F, m)] \Downarrow n_F, m$, with $n_F \cdot (1 + \delta) = r_F$. Thus, $|n_F - r_F| = |-n_F \cdot \delta| = |n_F \cdot \delta| \leq |n_F| \cdot \text{meps}(m')$ (as $\text{meps}(m') > 0$). By combining the results, we effectively get that:

$$|n_R - r_F| \leq |n_R - n_F| + |n_F| \cdot \text{meps}(m')$$

□

B Inversion of an affine form

This section describes how one can compute the inverse of an affine form $\hat{x}$, when $\gamma(\hat{x})$ contains only positive numbers. This approach is described by Stol and De Figueiredo in [20], but I wanted to write down the geometrical reasoning involved. In our case, we have an affine form $\hat{x}$, and $\gamma(\hat{x}) = [a; b]$ with $0 < a < b$. The goal is to compute an approximation of $1/\hat{x}$. Stol and De Figueiredo [20] chose to search for an approximation of the form $\alpha\hat{x} + \beta + \delta\epsilon_{n+1}$, where ϵ_{n+1} is a fresh noise term.

We will use figure 5 as an example. In that case, we have $a = 2$ and $b = 7$, so that $\gamma(\hat{x}) = [2; 7]$. We can see that the parallelogram defined by points A, B, C and D is a valid over-approximation of $1/x$, for $2 \leq x \leq 7$ (visually, the plot of the inverse function is contained by the parallelogram). Let us see how this parallelogram has been constructed, and how to express this parallelogram into an affine form. It starts with the construction of $(\mathcal{D}_1)$, which is tangent to $y = 1/x$ at $x = b = 7$. Thus, the equation of $(\mathcal{D}_1)$ is $y = \frac{-1}{b^2}(x-b)+\frac{1}{b}$. Then, by definition of a parallelogram, $(\mathcal{D}_2)$ will be parallel to $(\mathcal{D}_1)$. We just need to translate $(\mathcal{D}_1)$ sufficiently upwards so that the point $(a, 1/a) \in (\mathcal{D}_2)$. Then, $(\mathcal{D}_2)$ is uniquely characterized, and its equation is $y = \frac{-1}{b^2}(x - a) + \frac{1}{a}$. From these two lines, we can easily define the parallelogram defined by A, B, C and D. The translation into an affine form itself is simple: the average value represented by the dashed grey segment can be expressed as $\alpha\hat{x} + \beta$. The deviation from this average is handled by the new noise terms. In particular, we can notice that dashed grey line lies is equidistant from $(\mathcal{D}_1)$ and $(\mathcal{D}_2)$, so that the equation of this line is: $y = \frac{-1}{b^2}(x - \frac{a+b}{2}) + \frac{1}{2a} + \frac{1}{2b}$. We can deduce that $\alpha = \frac{-1}{b^2}$, $\beta = \frac{1}{2}(\frac{1}{a} + \frac{1}{b} - \alpha(a + b))$, and $\delta = \frac{1}{2}(\alpha(b - a) + \frac{1}{a} - \frac{1}{b})$.

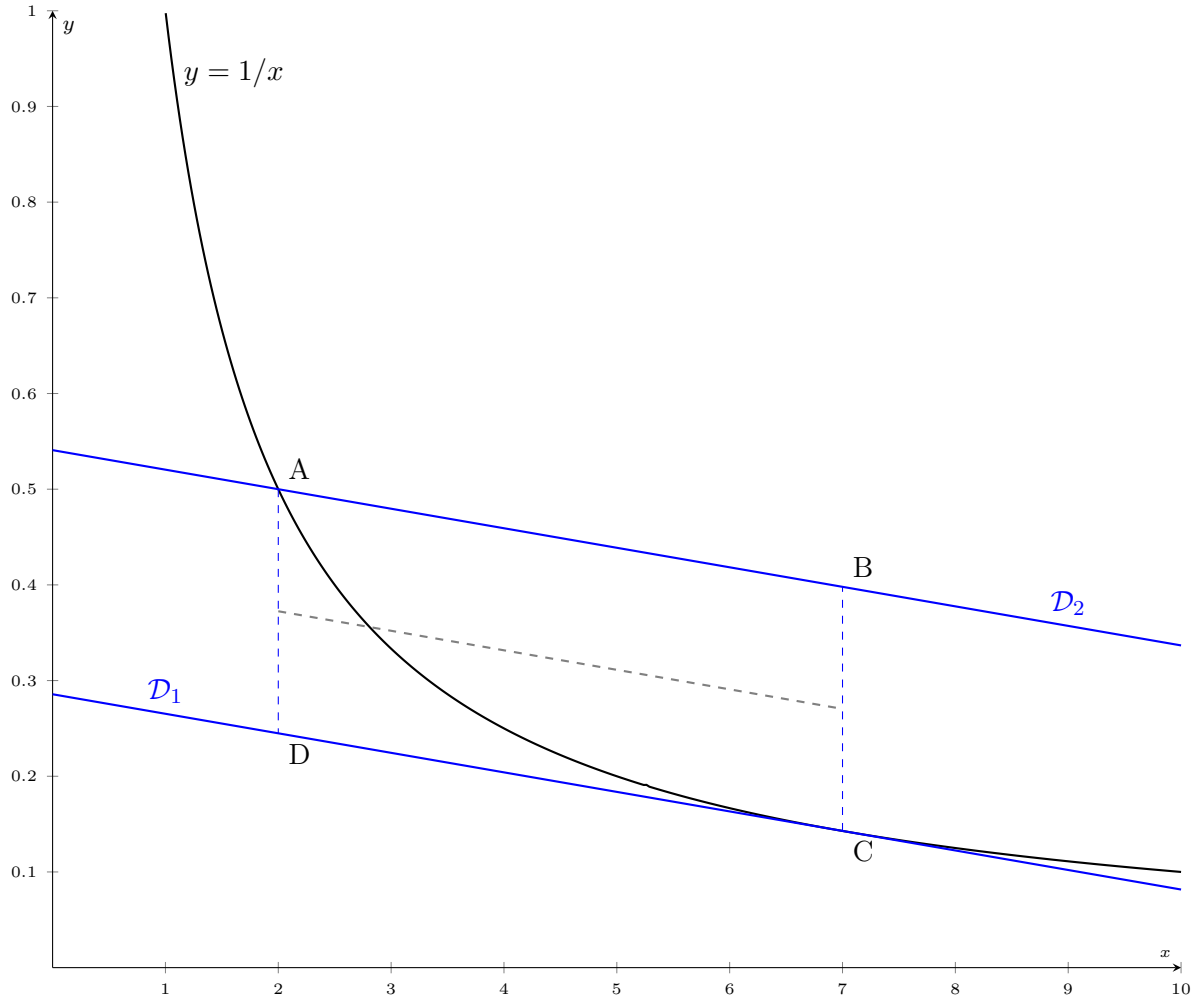

Figure 5: Example of inversion of an affine form

Remark We could have chosen to start by constructing the tangent to $y = 1/x$ at $x = a$, and continue to build the parallelogram this way. While this approach is still valid, it gives worse usually worse results: the created parallelogram would probably contain points having ordinate $y = 0$.

Comparison with intervals Let us compare the size of each representation. Concerning the inverse of an interval, the area used is $I = (b - a) \cdot (a^{-1} - b^{-1})$. Concerning the inverse of an affine form, the area used is $A = (b - a) \cdot 2\delta$. Thus, the ratio is

$$R = \frac{A}{I} = \frac{\alpha(b-a) + a^{-1} - b^{-1}}{a^{-1} - b^{-1}} = 1 + \alpha ab = 1 - \frac{a}{b}$$

As $0 < a < b$, $R < 1$, the approximation using affine forms is strictly better.