Floating-Point Format Inference in Mixed-Precision

Matthieu Martel(✉)

Laboratoire de Mathématiques et Physique (LAMPS),
Université de Perpignan Via Domitia, Perpignan, France
matthieu.martel@univ-perp.fr

Abstract. We address the problem of determining the minimal precision on the inputs and on the intermediary results of a program containing floating-point computations in order to ensure a desired accuracy on the outputs. The first originality of our approach is to combine forward and backward static analyses, done by abstract interpretation. The backward analysis computes the minimal precision needed for the inputs and intermediary values in order to have a desired accuracy on the results, specified by the user. The second originality is to express our analysis as a set of constraints made of first order predicates and affine integer relations only, even if the analyzed programs contain non-linear computations. These constraints can be easily checked by an SMT Solver. The information collected by our analysis may help to optimize the formats used to represent the values stored in the floating-point variables of programs. Experimental results are presented.

1 Introduction

Issues related to numerical accuracy are almost as old as computer science. An important step towards the design of more reliable numerical software was the definition, in the 1980's, of the IEEE754 Standard for floating-point arithmetic [2]. Since then, work has been carried out to determine the accuracy of floating-point computations by dynamic [3,17,29] or static [11,13,14] methods. This work has also been motivated by a few disasters due to numerical bugs [1,15].

While existing approaches may differ strongly each other in their way of determining accuracy, they have a common objective: to compute approximations of the errors on the outputs of a program depending on the initial errors on the data and on the roundoff of the arithmetic operations performed during the execution. The present work focuses on a slightly different problem concerning the relations between precision and accuracy. Here, the term *precision* refers to the number of bits used to represent a value, i.e. its format, while the term *accuracy* is a bound on the absolute error $|x - \hat{x}|$ between the represented $\hat{x}$ value and the exact value x that we would have in the exact arithmetic.

We address the problem of determining the minimal precision on the inputs and on the intermediary results of a program performing floating-point computations in order to ensure a desired accuracy on the outputs. This allows compilers

C. Barrett et al. (Eds.): NFM 2017, LNCS 10227, pp. 230–246, 2017.
DOI: 10.1007/978-3-319-57288-8_16

to select the most appropriate formats (for example IEEE754 half, single, double or quad formats [2,23]) for each variable. It is then possible to save memory, reduce CPU usage and use less bandwidth for communications whenever distributed applications are concerned. So, the choice of the best floating-point formats is an important compile-time optimization in many contexts. Our approach is also easily generalizable to the fixed-point arithmetic for which it is important to determine data formats, for example in FPGAs [12,19].

The first originality of our approach is to combine a forward and a backward static analysis, done by abstract interpretation [8,9]. The forward analysis is classical. It propagates safely the errors on the inputs and on the results of the intermediary operations in order to determine the accuracy of the results. Next, based on the results of the forward analysis and on assertions indicating which accuracy the user wants for the outputs at some control points, the backward analysis computes the minimal precision needed for the inputs and intermediary results in order to satisfy the assertions. Not surprisingly, the forward and backward analyses can be applied repeatedly and alternatively in order to refine the results until a fixed-point is reached.

The second originality of our approach is to express the forward and backward transfer functions as a set of constraints made of propositional logic formulas and relations between affine expressions over integers (and only integers). Indeed, these relations remain linear even if the analyzed program contains non-linear computations. As a consequence, these constraints can be easily checked by a SMT solver (we use Z3 in practice [4,21]). The advantage of the solver appears in the backward analysis, when one wants to determine the precision of the operands of some binary operation between two operands a and b, in order to obtain a certain accuracy on the result. In general, it is possible to use a more precise a with a less precise b or, conversely, to use a more precise b with a less precise a. Because this choice arises at almost any operation, there is a huge number of combinations on the admissible formats of all the data in order to ensure a given accuracy on the results. Instead of using an ad-hoc heuristic, we encode our problem as a set of constraints and we let a well-known, optimized solver generate a solution.

This article is organized as follows. We briefly introduce some elements of floating-point arithmetic, a motivating example and related work in Sect. 2. Our abstract domain as well as the forward and backward transfer functions are introduced in Sect. 3. The constraint generation is presented in Sect. 4 and experimental results are given in Sect. 5. Finally, Sect. 6 concludes.

2 Preliminary Elements

In this section we introduce some preliminary notions helpful to understand the rest of the article. Elements of floating-point arithmetic are introduced in Sect. 2.1. Further, an illustration of what our method does is given in Sect. 2.2. Related work is discussed in Sect. 2.3.

2.1 Elements of Floating-Point Arithmetic

We introduce here some elements of floating-point arithmetic [2,23]. First of all, a *floating-point number* x in base β is defined by

$$x = s \cdot (d_0.d_1 \dots d_{p-1}) \cdot \beta^e = s \cdot m \cdot \beta^{e-p+1} \tag{1}$$

where $s \in \{-1, 1\}$ is the sign, $m = d_0 d_1 \dots d_{p-1}$ is the *significand*, $0 \leq d_i < \beta$, $0 \leq i \leq p-1, p$ is the *precision* and e is the exponent, $e_{min} \leq e \leq e_{max}$.

A floating-point number x is *normalized* whenever $d_0 \neq 0$. Normalization avoids multiple representations of the same number. The IEEE754 Standard also defines denormalized numbers which are floating-point numbers with $d_0 = d_1 = \dots = d_k = 0, k < p-1$ and $e = e_{min}$. Denormalized numbers make underflow gradual [23]. The IEEE754 Standard defines binary formats (with $\beta = 2$) and decimal formats (with $\beta = 10$). In this article, without loss of generality, we only consider normalized numbers and we always assume that $\beta = 2$ (which is the most common case in practice). The IEEE754 Standard also specifies a few values for p, e_{min} and e_{max} which are summarized in Fig. 1. Finally, special values also are defined: `nan` (Not a Number) resulting from an invalid operation, $\pm\infty$ corresponding to overflows, and $+0$ and -0 (signed zeros).

Format	Name	p	e bits	e_{min}	e_{max}
Binary16	Half precision	11	5	-14	$+15$
Binary32	Single precision	24	8	-126	$+127$
Binary64	Double precision	53	11	-1122	$+1223$
Binary128	Quadruple precision	113	15	-16382	$+16383$

Fig. 1. Basic binary IEEE754 formats.

The IEEE754 Standard also defines five rounding modes for elementary operations over floating-point numbers. These modes are towards $-\infty$, towards $+\infty$, towards zero, to the nearest ties to even and to the nearest ties to away and we write them $\circ_{-\infty}, \circ_{+\infty}, \circ_0, \circ_{\sim_e}$ and $\circ_{\sim_a}$, respectively. The semantics of the elementary operations $\diamond \in \{+, -, \times, \div\}$ is then defined by

$$f_1 \diamond_\circ f_2 = \circ(f_1 \diamond f_2) \tag{2}$$

where $\circ \in \{\circ_{-\infty}, \circ_{+\infty}, \circ_0, \circ_{\sim_e}, \circ_{\sim_a}\}$ denotes the rounding mode. Equation (2) states that the result of a floating-point operation $\diamond_\circ$ done with the rounding mode $\circ$ returns what we would obtain by performing the exact operation $\diamond$ and next rounding the result using $\circ$. The IEEE754 Standard also specifies how the square root function must be rounded in a similar way to Eq. (2) but does not specify the roundoff of other functions like sin, log, etc.

We introduce hereafter two functions which compute the *u*nit in the *f*irst *p*lace and the *u*nit in the *l*ast *p*lace of a floating-point number. These functions

are used further in this article to generate constraints encoding the way roundoff errors are propagated throughout computations. The ufp of a number x is

$$\mathsf{ufp}(x) = \min\left\{i \in \mathbb{N} \ : \ 2^{i+1} > x\right\} = \lfloor \log_2(x) \rfloor. \tag{3}$$

The ulp of a floating-point number which significand has size p is defined by

$$\mathsf{ulp}(x) = \mathsf{ufp}(x) - p + 1. \tag{4}$$

The ufp of a floating-point number corresponds to the binary exponent of its most significant digit. Conversely, the ulp of a floating-point number corresponds to the binary exponent of its least significant digit. Note that several definitions of the ulp have been given [22].

2.2 Overview of Our Method

Let us consider the program of Fig. 2 which implements a simple linear filter. At each iteration $\mathtt{t}$ of the loop, the output $\mathtt{y}_t$ is computed as a function of the current input $\mathtt{x}_t$ and of the values $\mathtt{x}_{t-1}$ and $\mathtt{y}_{t-1}$ of the former iteration. Our program contains several annotations. First, the statement $\mathtt{require\_accuracy(y}_t\mathtt{,10)}$ on the last line of the code informs the system that the programmer wants to have 10 accurate binary digits on $\mathtt{y}_t$ at this control point. In other words, let $\mathtt{y}_t = d_0.d_1 \ldots d_n \cdot 2^e$ for some $n \geq 10$, the absolute error between the value v that $\mathtt{y}_t$ would have if all the computations where done with real numbers and the floating-point value $\hat{v}$ of $\mathtt{y}_t$ is less than $2^{e-11} : |v - \hat{v}| \leq 2^{e-9}$.

Note that accuracy is not a property of a number but a number that states how closely a particular floating-point number matches some ideal true value.

```
x_{t-1}:=[1.0,3.0]#16;
x_t:=[1.0,3.0]#16;
y_{t-1}:=0.0;
while(c) {
  u:=0.3 * y_{t-1};
  v:=0.7 * (x_t + x_{t-1});
  y_t:=u + v;
  y_{t-1}:=y_t;
};
require_accuracy(y_t,10);
```

```
x_{t-1}^{|9|}:=[1.0,3.0]^{|9|}; x_t^{|9|}:=[1.0,3.0]^{|9|};
y_{t-1}^{|10|}:=0.0^{|10|};
while(c) {
  u^{|10|}:=0.3^{|10|} *^{|10|} y_{t-1}^{|10|};
  v^{|10|}:=0.7^{|11|} *^{|10|} (x_t^{|9|} +^{|10|} x_{t-1}^{|9|});
  y_t^{|10|}:=u^{|10|} +^{|10|} v^{|10|};
  y_{t-1}^{|10|}:=y_t^{|10|}; };
require_accuracy(y_t,10);
```

```
x_{t-1}^{|16|}:=[1.0,3.0]^{|16|};
x_t^{|16|}:=[1.0,3.0]^{|16|};
y_{t-1}^{|52|}:=0.0^{|52|};
u^{|52|}:=0.3^{|52|} *^{|52|} y_{t-1}^{|52|};
v^{|15|}:=0.7^{|52|} *^{|15|} (x_t^{|16|} +^{|16|} x_{t-1}^{|16|});
y_t^{|15|}:=u^{|52|} +^{|15|} v^{|15|};
y_{t-1}^{|15|}:=y_t^{|15|};
```

```
x_{t-1}^{|9|}:=[1.0,3.0]^{|9|}; x_t^{|9|}:=[1.0,3.0]^{|9|};
y_{t-1}^{|8|}:=0.0^{|8|};
u^{|10|}:=0.3^{|8|} *^{|10|} y_{t-1}^{|8|};
v^{|10|}:=0.7^{|11|} *^{|10|} (x_t^{|9|} +^{|10|} x_{t-1}^{|9|});
y_t^{|10|}:=u^{|10|} +^{|10|} v^{|10|};
y_{t-1}^{|10|}:=y_t^{|10|};
require_accuracy(y_t,10);
```

Fig. 2. Top left: initial program. Top right: annotations after analysis. Bottom left: forward analysis (one iteration). Bottom right: backward analysis (one iteration).

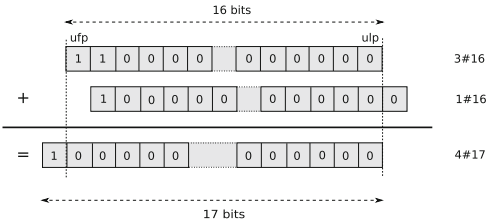

Fig. 3. Example of forward addition: `3.0#16` + `1.0#16` = `4.0#17`.

For example, using the basis $\beta = 10$ for the sake of simplicity, the floating-point value 3.149 represents π with an accuracy of 3. It itself has a precision of 4. It represents the real number 3.14903 with an accuracy of 4.

An abstract value $[a, b]_p$ represents the set of floating-point values with p accurate bits ranging from a to b. For example, in the code of Fig. 2, the variables $\mathtt{x}_{t-1}$ and $\mathtt{x}_t$ are initialized to the abstract value $[1.0, 3.0]_{16}$ thanks to the annotation `[1.0,3.0]#16`. Let $\mathbb{F}_p$ be the of set of all floating-point numbers with accuracy p. This means that, compared to exact value v computed in infinite precision, the value $\hat{v} = d_0.d_1 \dots d_n \cdot 2^e$ of $\mathbb{F}_p$ is such that $|v - \hat{v}| \leq 2^{e-p+1}$. By definition, using the function ufp introduced in Eq. (3), for any $x \in \mathbb{F}_p$ the roundoff error $\varepsilon(x)$ on x is bounded by $\varepsilon(x) < 2^{\mathsf{ulp}(x)} = 2^{\mathsf{ufp}(x)-p+1}$. Concerning the abstract values, intuitively we have the concretization function

$$\gamma([a, b]_p) = \{x \in \mathbb{F}_p \ : \ a \leq x \leq b\}. \tag{5}$$

These abstract values are special cases of the values used in other work [18] in the sense that, in the present framework, the errors attached to floating-point numbers have form $[-2^u, 2^u]$ for some integer u instead of arbitrary intervals with real bounds. Restricting the form of the errors enables one to simplify drastically the transfer functions for the backward analysis and the generation of constraints in Sect. 4. In this article, we focus on the accuracy of computations and we omit other problems related to runtime-errors [3,5]. In particular, overflows are not considered and we assume that any number with p accurate digits belongs to $\mathbb{F}_p$. In practice, a static analysis computing the ranges of the variables and rejecting programs which possibly contain overflows is done before our analysis.

In our example, $\mathtt{x}_t$ and $\mathtt{x}_{t-1}$ belong to $[1.0, 3.0]_{16}$ which means, by definition, that these variables have a value $\hat{v}$ ranging in $[1.0, 3.0]$ and such that the error between $\hat{v}$ and the value v that we would have in the exact arithmetic is bounded by $2^{\mathsf{ufp}(x)-15}$. Typically, in this example, this information would come from the specification of the sensor related to `x`. By default, the values for which no accuracy annotation is given (for instance the value of $\mathtt{y}_{t-1}$ in the example of Fig. 2) are considered as exact numbers rounded to the nearest in double precision. In this format numbers have 53 bits of significand (see Fig. 1). The last bit being rounded, these numbers have 52 accurate bits in our terminology

and, consequently, by default values belong to $\mathbb{F}_{52}$ in our framework. Based on the accuracy of the inputs, our forward analysis computes the accuracy of all the other variables and expressions. The program in the left bottom corner of Fig. 2 displays the result of the forward analysis on the first iteration of the loop. Let $\overrightarrow{\oplus}$ denote the forward addition (all the operations used in the current example are formally defined in Sect. 3). For example, the result of $\mathtt{x}_t + \mathtt{x}_{t-1}$ has 16 accurate digits since

$$\overrightarrow{\oplus}(\texttt{1.0\#16}, \texttt{1.0\#16}) = \texttt{2.0\#16}, \quad \overrightarrow{\oplus}(\texttt{1.0\#16}, \texttt{3.0\#16}) = \texttt{4.0\#17},$$
$$\overrightarrow{\oplus}(\texttt{3.0\#16}, \texttt{1.0\#16}) = \texttt{4.0\#17}, \quad \overrightarrow{\oplus}(\texttt{3.0\#16}, \texttt{3.0\#16}) = \texttt{6.0\#16}.$$

This is illustrated in Fig. 3 where we consider the addition of these values at the bit level. For the result of the addition $\overrightarrow{\boxplus}$ between intervals, we take the most pessimistic accuracy: $\overrightarrow{\boxplus}(\texttt{[1.0,3.0]\#16},\texttt{[1.0,3.0]\#16}) = \texttt{[2.0,6.0]\#16}$.

The backward analysis is performed after the forward analysis and takes advantage of the accuracy requirement at the end of the code (see the right bottom corner of Fig. 2 for an unfolding of the backward analysis on the first iteration of the loop). Since, in our example, 10 bits only are required for $\mathtt{y}_t$, the result of the addition u+v also needs 10 accurate bits only. By combining this information with the result of the forward analysis, it is then possible to lower the number of bits needed for one of the operands. Let $\overleftarrow{\oplus}$ be the backward addition. For example, for $\mathtt{x}_t+\mathtt{x}_{t-1}$ in the assignment of v, we have:

$$\overleftarrow{\oplus}(\texttt{2.0\#10}, \texttt{1.0\#16}) = \texttt{1.0\#8}, \quad \overleftarrow{\oplus}(\texttt{2.0\#10}, \texttt{3.0\#16}) = \texttt{-1.0\#8},$$
$$\overleftarrow{\oplus}(\texttt{6.0\#10}, \texttt{1.0\#16}) = \texttt{5.0\#9}, \quad \overleftarrow{\oplus}(\texttt{6.0\#10}, \texttt{3.0\#16}) = \texttt{3.0\#8}.$$

Conversely to the forward function, the interval function now keeps the largest accuracy arising in the computation of the bounds:

$$\overleftarrow{\boxplus}(\texttt{[2.0,6.0]\#10},\texttt{[1.0,3.0]\#16}) = \texttt{[1.0,3.0]\#9}.$$

By processing similarly on all the elementary operations and after computation of the loop fixed point, we obtain the final result of the analysis displayed in the top right corner of Fig. 2. This information may be used to determine the most appropriate data type for each variable and operation, as shown in Fig. 4. To obtain this result we generate a set of constraints corresponding to the forward and backward transfer functions for the operations of the program. There exists several ways to handle a backward operation: when the accuracy on the inputs x and y computed by the forward analysis is too large wrt. the desired accuracy on the result, one may lower the accuracy of either x or y or both.

```
volatile half x_{t-1}, x_t;
half u, v, y_t;
float y_{t-1}, tmp;
y_{t-1}:=0.0;
while(c) {
  u:=0.3 * y_{t-1};
  tmp:=x_t + x_{t-1};
  v:=0.7 * tmp;
  y_t:=u + v;
  y_{t-1}:=y_t;
};
```

Fig. 4. Final program with generated data types for the example of Fig. 2.

Since this question arises at each binary operation, we would face to a huge number of combinations if we decided to enumerate all possibilities. Instead, we generate a disjunction of constraints corresponding to the minimization of the accuracy of each operand and we let the solver search for a solution. The control flow of the program is also encoded with constraints. For a sequence of statements, we relate the accuracy of the former statements to the accuracy of the latter ones. Each variable x has three parameters: its forward, backward and final accuracy, denoted $\mathsf{acc}_F(x)$, $\mathsf{acc}_B(x)$ and $\mathsf{acc}(x)$ respectively. We must always have

$$0 \leq \mathsf{acc}_B(x) \leq \mathsf{acc}(x) \leq \mathsf{acc}_F(x). \tag{6}$$

For the forward analysis, the accuracy of some variable may decrease when passing to the next statement (we may only weaken the pre-conditions). Conversely, in the backward analysis, the accuracy of a given variable may increase when we jump to a former statement in the control graph (the post-conditions may only be strengthened). For a loop, we relate the accuracy of the variables at the beginning and at the end of the body, in a standard way.

The key point of our technique is to generate simple constraints made of propositional logic formulas and of affine expressions among integers (even if the floating-point computations in the source code are non-linear). A static analysis computing safe ranges at each control point is performed before our accuracy analysis. Then the constraints depend on two kinds of integer parameters: the ufp of the values and their accuracies $\mathsf{acc}_F, \mathsf{acc}_B$ and acc. For instance, given control points ℓ_1, ℓ_2 and ℓ_3, the set C of constraints generated for $\texttt{3.0\#16}^{\ell_1} +^{\ell_3} \texttt{1.0\#16}^{\ell_2}$, assuming that we require 10 accurate bits for the result are:

$$C = \left\{ \begin{array}{l} \mathsf{acc}_F(\ell_1) = 16,\ \mathsf{acc}_F(\ell_2) = 16,\ r^{\ell_3} = 2 - \max(\mathsf{acc}_F(\ell_1) - 1, \mathsf{acc}_F(\ell_2)), \\ (1 - \mathsf{acc}_F(\ell_1)) = \mathsf{acc}_F(\ell_2) \Rightarrow i^{\ell_3} = 1,\ (1 - \mathsf{acc}_F(\ell_1)) \neq \mathsf{acc}_F(\ell_2) \Rightarrow i^{\ell_3} = 0, \\ \mathsf{acc}_F(\ell_3) = r^{\ell_3} - i^{\ell_3},\ \mathsf{acc}_B(\ell_3) = 10 \\ \mathsf{acc}_B(\ell_1) = 1 - (2 - \mathsf{acc}_B(\ell_3)),\ \mathsf{acc}_B(\ell_2) = 1 - (2 - \mathsf{acc}_B(\ell_3)) \end{array} \right\}.$$

For the sake of conciseness, the constraints corresponding to Eq. (6) have been omitted in C. For example, for the forward addition, the accuracy $\mathsf{acc}_F(\ell_3)$ of the result is the number of bits between $\mathsf{ufp}(3.0 + 1.0) = 2$ and the ufp u of the error which is

$$\begin{aligned} u &= \max\left(\mathsf{ufp}(3.0) - \mathsf{acc}_F(\ell_1), \mathsf{ufp}(1.0) - \mathsf{acc}_F(\ell_2)\right) + i \\ &= \max\left(1 - \mathsf{acc}_F(\ell_1), 0 - \mathsf{acc}_F(\ell_2)\right) + i, \end{aligned}$$

where $i = 0$ or $i = 1$ depending on some condition detailed later. The constraints generated for each kind of expression and command are detailed in Sect. 4.

2.3 Related Work

Several approaches have been proposed to determine the best floating-point formats as a function of the expected accuracy on the results. Darulova and Kuncak use a forward static analysis to compute the propagation of errors [11]. If the

computed bound on the accuracy satisfies the post-conditions then the analysis is run again with a smaller format until the best format is found. Note that in this approach, all the values have the same format (contrarily to our framework where each control-point has its own format). While Darulova and Kuncak develop their own static analysis, other static techniques [13,29] could be used to infer from the forward error propagation the suitable formats. Chiang *et al.* [7] have proposed a method to allocate a precision to the terms of an arithmetic expression (only). They use a formal analysis via Symbolic Taylor Expansions and error analysis based on interval functions. In spite of our linear constraints, they solve a quadratically constrained quadratic program to obtain annotations.

Other approaches rely on dynamic analysis. For instance, the Precimonious tool tries to decrease the precision of variables and checks whether the accuracy requirements are still fulfilled [24,27]. Lam *et al.* instrument binary codes in order to modify their precision without modifying the source codes [16]. They also propose a dynamic search method to identify the pieces of code where the precision should be modified. Finally, another related research axis concerns the compile-time optimization of programs in order to improve the accuracy of the floating-point computation in function of given ranges for the inputs, without modifying the formats of the numbers [10,26].

3 Abstract Semantics

In this section, we give a formal definition of the abstract domain and transfer functions presented informally in Sect. 2. The domain is defined in Sect. 3.1 and the transfer functions are given in Sect. 3.2.

3.1 Abstract Domain

Let $\mathbb{F}_p$ be the set floating-point numbers with accuracy p (we assume that the error between $x \in \mathbb{F}_p$ and the value that we would have in the exact arithmetic is less than $2^{\mathsf{ufp}(x)-p+1}$) and let $\mathbb{I}_p$ be the set of all intervals of floating-point numbers with accuracy p. As mentioned in Sect. 2.2, we assume that no overflow arises during our analysis and we omit to specify the lower and upper bounds of $\mathbb{F}_p$. An element $i^\sharp \in \mathbb{I}_p$, denoted $i^\sharp = [\underline{f}, \overline{f}]_p$, is then defined by two floating-point numbers and an accuracy p. We have

$$\mathbb{I}_p \ni [\underline{f}, \overline{f}]_p = \{f \in \mathbb{F}_p : \underline{f} \leq f \leq \overline{f}\} \text{ and } \mathbb{I} = \bigcup_{p \in \mathbb{N}} \mathbb{I}_p. \tag{7}$$

Our abstract domain is the complete lattice $\mathcal{D}^\sharp = \langle \mathbb{I}, \sqsubseteq, \sqcup, \sqcap, \bot_{\mathbb{I}}, \top_{\mathbb{I}} \rangle$ where elements are ordered by $[a,b]_p \sqsubseteq [c,d]_q \iff [a,b] \subseteq [c,d]$ and $q \leq p$. In other words, $[a,b]_p$ is more precise than $[c,d]_q$ if it is an included interval with a greater accuracy. Let $\circ_{r,m}(x)$ denote the rounding of x at precision r using the rounding mode m. Then the join and meet operators are defined by

$$[a,b]_p \sqcup [c,d]_q = [\circ_{r,-\infty}(u), \circ_{r,+\infty}(v)]_r \text{ with } r = \min(p,q),\ [u,v] = [a,b] \cup [c,d] \quad (8)$$

and

$$[a,b]_p \sqcap [c,d]_q = [u,v]_r \text{ with } r = \max(p,q),\ [u,v] = [a,b] \cap [c,d]. \quad (9)$$

In addition, we have $\bot_{\mathbb{I}} = \emptyset_{+\infty}$ and $\top_{\mathbb{I}} = [-\infty, +\infty]_0$ and we have $[a,b]_p \sqcap [c,d]_q = \bot_{\mathbb{I}}$ whenever $[a,b] \cap [c,d] = \emptyset$. Let $\alpha : \wp(\mathbb{F}) \to \mathbb{I}$ be the abstraction function which maps a set of floating-point numbers X with different accuracies p_i, $1 \leq i \leq n$ to a value of $\mathbb{I}$. Let $x_{min} = \min(X)$, $x_{max} = \max(X)$ and $p = \min\{q : x \in X \text{ and } x \in \mathbb{F}_q\}$ the minimal accuracy in X. We have,

$$\alpha(X) = [\circ_{p,-\infty}\big(\min(X)\big), \circ_{p,+\infty}\big(\max(X)\big)]_p \quad \text{where } p = \min\{q : X \cap \mathbb{F}_q \neq \emptyset\}. \quad (10)$$

Let $\gamma : \mathbb{I} \to \wp(\mathbb{F})$ and $i^\sharp = [a,b]_p$. The concretization function $\gamma(i^\sharp)$ is defined as:

$$\gamma(i^\sharp) = \bigcup_{q \geq p} \{x \in \mathbb{F}_q \ : a \leq x \leq b\}. \quad (11)$$

Using the functions α and γ of Eqs. (10) and (11), we define the Galois connection $\langle \wp(\mathbb{F}), \subseteq, \cup, \cap, \emptyset, \mathbb{F}\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathbb{I}, \sqsubseteq, \sqcup, \sqcap, \bot_{\mathbb{I}}, \top_{\mathbb{I}}\rangle$ [8].

3.2 Transfer Functions

In this section, we introduce the forward and backward transfer functions for the abstract domain $\mathcal{D}^\sharp$ of Sect. 3.1. These functions are defined using the *u*nit in the *f*irst *p*lace of a floating-point number introduced in Sect. 2.1. First, we introduce the forward transfer functions corresponding to the addition $\overrightarrow{\oplus}$ and product $\overrightarrow{\otimes}$ of two floating-point numbers $x \in \mathbb{F}_p$ and $y \in \mathbb{F}_q$. The addition and product are defined by

$$\overrightarrow{\oplus}(x_p, y_q) = (x+y)_r \text{ where } r = \mathsf{ufp}(x+y) - \mathsf{ufp}\big(\varepsilon(x_p) + \varepsilon(y_q)\big), \quad (12)$$

$$\overrightarrow{\otimes}(x_p, y_q) = (x \times y)_r \text{ where } r = \mathsf{ufp}(x \times y) - \mathsf{ufp}\big(y \cdot \varepsilon(x_p) + x \cdot \varepsilon(y_q) + \varepsilon(x_p) \cdot \varepsilon(y_q)\big). \quad (13)$$

In Eqs. (12) and (13), $x + y$ and $x \times y$ denote the exact sum and product of the two values. In practice, this sum must be done with enough accuracy in order to ensure that the result has accuracy r, for example by using more precision than the accuracy of the inputs. The errors on the addition and product may be bounded by $e_+ = \varepsilon(x_p) + \varepsilon(y_q)$ and $e_\times = y \cdot \varepsilon(x_p) + x \cdot \varepsilon(y_q) + \varepsilon(x_p) \cdot \varepsilon(y_q)$, respectively. Then the most significant bits of the errors have weights $\mathsf{ufp}(e_+)$ and $\mathsf{ufp}(e_\times)$ and the accuracies of the results are $\mathsf{ufp}(x+y) - \mathsf{ufp}(e_+)$ and $\mathsf{ufp}(x \times y) - \mathsf{ufp}(e_\times)$, respectively.

We introduce now the backward transfer functions $\overleftarrow{\oplus}$ and $\overleftarrow{\otimes}$. We consider the operation between x_p and y_q whose result is z_r. Here, z_r and y_q are known while x_p is unknown. We have

$$\overleftarrow{\oplus}(z_r, y_q) = (z-y)_p \text{ where } p = \mathsf{ufp}(z-y) - \mathsf{ufp}\big(\varepsilon(z_r) - \varepsilon(y_q)\big), \quad (18)$$

$$\overleftarrow{\otimes}(z_r, y_q) = (z \div y)_p \text{ where } p = \mathsf{ufp}(z \div y) - \mathsf{ufp}\left(\frac{y \cdot \varepsilon(z_r) - z \cdot \varepsilon(y_q)}{y \cdot (y + \varepsilon(y_q))}\right). \tag{19}$$

The correctness of the backward product relies on the following arguments. Let $\varepsilon(x), \varepsilon(y)$ and $\varepsilon(z)$ be the exact errors on x, y and z respectively. We have $\varepsilon(z) = x \cdot \varepsilon(y) + y \cdot \varepsilon(x) + \varepsilon(x) \cdot \varepsilon(y)$ and then $\varepsilon(x) \cdot (y + \varepsilon(y)) = \varepsilon(z) - x \cdot \varepsilon(y) = \varepsilon(z) - \frac{z}{y} \cdot \varepsilon(y)$. Finally, we conclude that $\varepsilon(x) = \dfrac{y \cdot \varepsilon(z_r) - z \cdot \varepsilon(y_q)}{y \cdot (y + \varepsilon(y_q))}$.

We end this section by extending the operations to the values of the abstract domain $\mathcal{D}^\sharp$ of Sect. 3.1. First, let $p \in \mathbb{N}$, let $m \in \{-\infty, +\infty, \sim_e, \sim_a, 0\}$ be a rounding mode and let $\circ_{p,m} : \mathbb{F} \to \mathbb{F}_p$ be the rounding function which returns the roundoff of a number at precision p using the rounding mode m. We write $\overrightarrow{\boxplus}$ and $\overleftarrow{\boxplus}$ the forward and backward addition and $\overrightarrow{\boxtimes}$ and $\overleftarrow{\boxtimes}$ the forward and backward products on $\mathcal{D}^\sharp$. These functions are defined in Fig. 5. The forward functions $\overrightarrow{\boxplus}$ and $\overrightarrow{\boxtimes}$ take two operands $[\underline{x}, \overline{x}]_p$ and $[\underline{y}, \overline{y}]_q$ and return the resulting abstract value $[\underline{z}, \overline{z}]_r$. The backward functions take three arguments: the operands $[\underline{x}, \overline{x}]_p$ and $[\underline{y}, \overline{y}]_q$ known from the forward pass and the result $[\underline{z}, \overline{z}]_r$ computed by the backward pass [20]. Then $\overleftarrow{\boxplus}$ and $\overleftarrow{\boxtimes}$ compute the backward value $[\underline{x'}, \overline{x'}]_{p'}$ of the first operand. The backward value of the second operand can be obtained by inverting the operands $[\underline{x}, \overline{x}]_p$ and $[\underline{y}, \overline{y}]_q$. An important point in these formulas is that, in forward mode, the resulting intervals inherit from the minimal accuracy computed for their bounds while, in backward mode, the maximal accuracy computed for the bounds is assigned to the interval.

$$\overrightarrow{\boxplus}\left([\underline{x}, \overline{x}]_p, [\underline{y}, \overline{y}]_q\right) = [\circ_{r,-\infty}(\underline{z}), \circ_{r,+\infty}(\overline{z})]_r \text{ with } \begin{cases} \underline{z}_{r_1} = \overrightarrow{\oplus}(\underline{x}_p, \underline{y}_q), \\ \overline{z}_{r_2} = \overrightarrow{\oplus}(\overline{x}_p, \overline{y}_q), \ r = \min(r_1, r_2). \end{cases} \tag{14}$$

$$\overrightarrow{\boxtimes}\left([\underline{x}, \overline{x}]_p, [\underline{y}, \overline{y}]_q\right) = [\circ_{r,-\infty}(\underline{z}), \circ_{r,+\infty}(\overline{z})]_r \text{ with } \begin{cases} a_{r_1} = \overrightarrow{\otimes}(\underline{x}_p, \underline{y}_q), \ b_{r_2} = \overrightarrow{\otimes}(\underline{x}_p, \overline{y}_q), \\ c_{r_3} = \overrightarrow{\otimes}(\overline{x}_p, \underline{y}_q), \ d_{r_4} = \overrightarrow{\otimes}(\overline{x}_p, \overline{y}_q), \\ \underline{z} = \min(a_{r_1}, \ b_{r_2}, \ c_{r_3}, \ d_{r_4}), \\ \overline{z} = \max(a_{r_1}, \ b_{r_2}, \ c_{r_3}, \ d_{r_4}), \\ r = \min\ (r_1, r_2, r_3, r_4). \end{cases} \tag{15}$$

$$\overleftarrow{\boxplus}\left([\underline{x}, \overline{x}]_p, [\underline{y}, \overline{y}]_q, [, \overline{z}]_r\right) = [\underline{x'}, \overline{x'}]_{p'} \text{ with } \begin{cases} \underline{u}_{r_1} = \overleftarrow{\oplus}(\underline{z}_r, \overline{y}_q), \ \overline{u}_{r_2} = \overleftarrow{\oplus}(\overline{z}_r, \underline{y}_q), \\ \underline{x'} = \max(\underline{u}, \underline{x}), \ \overline{x'} = \min(\overline{u}, \overline{x}), \\ p' = \max(r_1, r_2)\ . \end{cases} \tag{16}$$

$$\overleftarrow{\boxtimes}\left([\underline{x}, \overline{x}]_p, [\underline{y}, \overline{y}]_q, [\underline{z}, \overline{z}]_r\right) = [\underline{x'}, \overline{x'}]_{p'} \text{ with } \begin{cases} a_{r_1} = \overleftarrow{\otimes}(\underline{z}_r, \underline{y}_q), \ b_{r_2} = \overleftarrow{\otimes}(\underline{z}_r, \overline{y}_q), \\ c_{r_3} = \overleftarrow{\otimes}(\overline{z}_r, \underline{y}_q), \ d_{r_4} = \overleftarrow{\otimes}(\overline{z}_r, \overline{y}_q), \\ \underline{u} = \min(a_{r_1}, \ b_{r_2}, \ c_{r_3}, \ d_{r_4}), \\ \overline{u} = \max(a_{r_1}, \ b_{r_2}, \ c_{r_3}, \ d_{r_4}), \\ \underline{x'} = \max(\underline{u}, \underline{x}), \ \overline{x'} = \min(\overline{u}, \overline{x}), \\ p' = \max\ (r_1, r_2, r_3, r_4). \end{cases} \tag{17}$$

Fig. 5. Forward and backward transfer functions for the addition and product on $\mathcal{D}^\sharp$.

4 Constraint Generation

In this section, we introduce our system of constraints. The transfer functions of Sect. 3 are not directly translated into constraints because the resulting system would be too difficult to solve, containing non-linear constraints among non-integer quantities. Instead, we reduce the problem to a system of constraints made of linear relations between integer elements only. Sections 4.1 and 4.2 introduce the constraints for arithmetic expressions and programs, respectively.

4.1 Constraints for Arithmetic Expressions

In this section, we introduce the constraints generated for arithmetic expressions. As mentioned in Sect. 2, we assume that a range analysis is performed before the accuracy analysis and that a bounding interval is given for each variable and each value at any control point of the input programs.

Let us start with the forward operations. Let $x_p \in \mathbb{F}_p$ and $y_q \in \mathbb{F}_q$ and let us consider the operation $\overrightarrow{\oplus}(x_p, y_q) = z_r$. We know from Eq. (12) that $r_+ = \mathsf{ufp}(x+y) - \mathsf{ufp}(\varepsilon_+)$ with $\varepsilon_+ = \varepsilon(x_p) + \varepsilon(y_q)$. We need to over-approximate ε_+ in order to ensure r_+. Let $a = \mathsf{ufp}(x)$ and $b = \mathsf{ufp}(b)$. We have $\varepsilon(x) < 2^{a-p+1}$ and $\varepsilon(y) < 2^{b-p+1}$ and, consequently, $\varepsilon_+ < 2^{a-p+1} + 2^{b-p+1}$. We introduce the function ι defined by $\iota(u,v) = \begin{cases} 1 \text{ if } u = v, \\ 0 \text{ otherwise} \end{cases}$. We have

$$\begin{aligned} \mathsf{ufp}(\varepsilon_+) &< \max(a-p+1, b-q+1) + \iota(a-p, b-q) \\ &\leq \max(a-p, b-q) + \iota(a-p, b-q) \end{aligned}$$

and we conclude that

$$r_+ = \mathsf{ufp}(x+y) - \max(a-p, b-q) - \iota(a-p, b-q). \tag{20}$$

Note that, since we assume that a range analysis has been performed before the accuracy analysis, $\mathsf{ufp}(x+y)$, a and b are known at constraint generation time. For the forward product, we know from Eq. (13) that $r_\times = \mathsf{ufp}(x \times y) - \mathsf{ufp}(\varepsilon_\times)$ with $\varepsilon_\times = x \cdot \varepsilon(y_q) + y \cdot \varepsilon(x_p) + \varepsilon(x_p) \cdot \varepsilon(y_q)$. Again, let $a = \mathsf{ufp}(x)$ and $b = \mathsf{ufp}(b)$. We have, by definition of ufp, $2^a \leq x < 2^{a+1}$ and $2^b \leq y < 2^{b+1}$. Then $\varepsilon_\times$ may be bound by

$$\begin{aligned} \varepsilon_\times &< 2^{a+1} \cdot 2^{b-q+1} + 2^{b+1} \cdot 2^{a-p+1} + 2^{a-p+1} \cdot 2^{b-q+1} \\ &= 2^{a+b-q+2} + 2^{a+b-p+2} + 2^{a+b-p-q+2}. \end{aligned}$$

Since $a+b-p-q+2 < a+b-p+2$ and $a+b-p-q+2 < a+b-q+2$, we may get rid of the last term of the former equation and we obtain that

$$\begin{aligned} \mathsf{ufp}(\varepsilon_\times) &< \max(a+b-p+2, a+b-q+2) + \iota(p,q) \\ &\leq \max(a+b-p+1, a+b-q+1) + \iota(p,q). \end{aligned}$$

We conclude that

$$r_\times = \mathsf{ufp}(x \times y) - \max(a+b-p+1, a+b-q+1) - \iota(p,q). \tag{21}$$

Note that, by reasoning on the exponents of the values, the constraints resulting from a product become linear. We consider now the backward transfer functions. If $\overleftarrow{\oplus}(z_r, y_q) = x_{p_+}$ then we know from Eq. (18) that $p_+ = \mathsf{ufp}(z-y) - \mathsf{ufp}(\varepsilon_+)$ with $\varepsilon_+ = \varepsilon(z_r) - \varepsilon(y-q)$. Let $c = \mathsf{ufp}(z)$, we over-approximate ε_+ using the relations $\varepsilon(z_r) < 2^{c-r+1}$ and $\varepsilon(y_q) > 0$. So, $\mathsf{ufp}(\varepsilon_+) < c - r + 1$ and

$$p_+ = \mathsf{ufp}(z-y) - c + r \tag{22}$$

Finally, for the backward product, using Eq. (19) we know that if $\overleftarrow{\otimes}(z_r, y_q) = x_{p_\times}$ then $p_\times = \mathsf{ufp}(x) - \mathsf{ufp}(\varepsilon_\times)$ with $\varepsilon_\times = \frac{y\cdot\varepsilon(z) - z\cdot\varepsilon(y)}{y\cdot(y+\varepsilon(y))}$. Using the relations $2^b \le y < 2^{b+1}$, $2^c \le z < 2^{c+1}$, $\varepsilon(y) < 2^{b-q+1}$ and $\varepsilon(z) < 2^{c-r+1}$, we deduce that $y\cdot\varepsilon(z) - z\cdot\varepsilon(y) < 2^{b+c-r+2} - 2^{b+c-q+1}$ and that $\frac{1}{y\cdot(y+\varepsilon(y))} < 2^{-2b}$. Consequently, $\varepsilon_\times < 2^{-2b}\cdot(2^{b+c-r+2} - 2^{b+c-q+1}) \le 2^{c-b-r+1} - 2^{c-b-q}$ and it results that

$$p_\times = \mathsf{ufp}(x) - \max(c-b-r+1, c-b-q). \tag{23}$$

4.2 Systematic Constraint Generation

To explain the constraint generation, we use the simple imperative language of Eq. (24) in which a unique label $\ell \in \mathrm{Lab}$ is attached to each expression and command to identify without ambiguity each node of the syntactic tree.

$$\begin{aligned} e &::= c\texttt{\#p}^\ell \mid x^\ell \mid e_1^{\ell_1} +^\ell e_2^{\ell_2} \mid e_1^{\ell_1} -^\ell e_2^{\ell_2} \mid e_1^{\ell_1} \times^\ell e_2^{\ell_2} \\ c &::= x :=^\ell e^{\ell_1} \mid c_1^{\ell_1}\ ;\ c_2^{\ell_2} \mid \texttt{if}^\ell\ e^{\ell_0}\ \texttt{then}\ c_1^{\ell_1}\ \texttt{else}\ c_2^{\ell_2} \\ &\quad \mid \texttt{while}^\ell\ e^{\ell_0}\ \texttt{do}\ c_1^{\ell_1} \mid \texttt{require\_accuracy}(x,n)^\ell \end{aligned} \tag{24}$$

As in Sect. 2, $c\texttt{\#p}$ denotes a constant c with accuracy p and the statement $\texttt{require\_accuracy}(x,n)^\ell$ indicates that x must have at least accuracy n at control point ℓ. The set of identifiers occurring in the source program is denoted Id. Concerning the arithmetic expressions, we assign to each label ℓ of the expression three variables in our system of constraints, $\mathsf{acc}_F(\ell)$, $\mathsf{acc}_B(\ell)$ and $\mathsf{acc}(\ell)$ respectively corresponding to the forward, backward and final accuracies and we systematically generate the constraints $0 \le \mathsf{acc}_B(\ell) \le \mathsf{acc}(\ell) \le \mathsf{acc}_F(\ell)$.

For each control point in an arithmetic expression, we assume given a range $[\underline{\ell}, \overline{\ell}] \subseteq \mathbb{F}$, computed by static analysis and which bounds the values possibly occurring at Point ℓ at run-time. Our constraints use the unit in the first place $\mathsf{ufp}(\underline{\ell})$ and $\mathsf{ufp}(\overline{\ell})$ of these ranges. Let $\Lambda : \mathrm{Id} \to \mathrm{Id}\times\mathrm{Lab}$ be an environment which relates each identifier x to its last assignment x^ℓ: Assuming that $x :=^\ell e^{\ell_1}$ is the last assignment of x, the environment Λ maps x to x^ℓ (we will use join operators when control flow branches will be considered). Then $\mathcal{E}[\![e]\!]\ \Lambda$ generates the set of constraints for the expression e in the environment Λ. These constraints, defined in Fig. 6, are derived from equations of Sect. 4.1. For commands, labels are used

$$\mathcal{E}[\ c\#\mathtt{p}^{\ell}\ ]\Lambda = \{\mathsf{acc}_F(\ell) = p\}$$

$$\mathcal{E}[\ x^{\ell}\ ]\Lambda = \{\mathsf{acc}_F(\ell) = \mathsf{acc}_F(\Lambda(x)),\ \mathsf{acc}_B(\ell) = \mathsf{acc}_B(\Lambda(x))\}$$

$$\mathcal{E}[e_1^{\ell_1} +^{\ell} e_2^{\ell_2}]\Lambda = C[e_1^{\ell_1}]\Lambda \ \cup\ C[e_2^{\ell_2}]\Lambda \ \cup\ F_+(\ell_1,\ell_2,\ell) \ \cup\ O_+(\ell_1,\ell_2,\ell)$$

$$\mathcal{E}[e_1^{\ell_1} \times^{\ell} e_2^{\ell_2}]\Lambda = C[e_1^{\ell_1}]\Lambda \ \cup\ C[e_2^{\ell_2}]\Lambda \ \cup\ F_\times(\ell_1,\ell_2,\ell) \ \cup\ O_\times(\ell_1,\ell_2,\ell)$$

$$\mathcal{O}_+(\ell_1,\ell_2,\ell) = \left|\begin{array}{l}\mathcal{B}_+(\ell_1,\ell_2,\ell) \ \cup\ B_+(\ell_2,\ell_1,\ell)\\ \cup\ \big\{\big(\mathsf{acc}(\ell_1) \le \mathsf{acc}_F(\ell_1)\ \wedge\ \mathsf{acc}(\ell_2) \ge \mathsf{acc}_B(\ell_2)\big)\\ \qquad \vee\ \big(\mathsf{acc}(\ell_2) \le \mathsf{acc}_F(\ell_2)\ \wedge\ \mathsf{acc}(\ell_1) \ge \mathsf{acc}_B(\ell_1)\big)\big\}\end{array}\right.$$

$$\mathcal{O}_\times(\ell_1,\ell_2,\ell) = \left|\begin{array}{l}\mathcal{B}_\times(\ell_1,\ell_2,\ell) \ \cup\ B_\times(\ell_2,\ell_1,\ell)\\ \cup\ \big\{\big(\mathsf{acc}(\ell_1) \le \mathsf{acc}_F(\ell_1)\ \wedge\ \mathsf{acc}(\ell_2) \ge \mathsf{acc}_B(\ell_2)\big)\\ \qquad \vee\ \big(\mathsf{acc}(\ell_2) \le \mathsf{acc}_F(\ell_2)\ \wedge\ \mathsf{acc}(\ell_1) \ge \mathsf{acc}_B(\ell_1)\big)\big\}\end{array}\right.$$

$$\mathcal{F}_+(\ell_1,\ell_2,\ell) = \left\{\begin{array}{l}\overline{r}^{\ell} = \mathsf{ufp}(\overline{\ell}) - \max(\overline{\ell_1} - \mathsf{acc}_F(\ell_1), \overline{\ell_2} - \mathsf{acc}_F(\ell_2)),\\ \underline{r}^{\ell} = \mathsf{ufp}(\underline{\ell}) - \max(\underline{\ell_1} - \mathsf{acc}_F(\ell_1), \underline{\ell_2} - \mathsf{acc}_F(\ell_2)),\\ \overline{i}^{\ell} = \big(\mathsf{ufp}(\overline{\ell_1}) - \mathsf{acc}_F(\ell_1) = \mathsf{ufp}(\overline{\ell_2}) - \mathsf{acc}_F(\ell_2)\big)?\ 1\ :\ 0,\\ \underline{i}^{\ell} = \big(\mathsf{ufp}(\underline{\ell_1}) - \mathsf{acc}_F(\ell_1) = \mathsf{ufp}(\underline{\ell_2}) - \mathsf{acc}_F(\ell_2)\big)?\ 1\ :\ 0,\\ \mathsf{acc}_F(\ell) = \min(\underline{r^{\ell}} - \underline{i^{\ell}}, \overline{r^{\ell}} - \overline{i^{\ell}})\end{array}\right\}$$

$$\mathcal{B}_+(\ell_1,\ell_2,\ell) = \left\{\begin{array}{l}\overline{s}^{\ell_1} = \mathsf{ufp}(\overline{\ell_1}) - (\mathsf{ufp}(\overline{\ell}) - \mathsf{acc}_B(\ell)),\\ \underline{s}^{\ell_1} = \mathsf{ufp}(\underline{\ell_1}) - (\mathsf{ufp}(\underline{\ell}) - \mathsf{acc}_B(\ell)),\ \mathsf{acc}_B(\ell_1) = \max(\underline{s}^{\ell_1}, \overline{s}^{\ell_1})\end{array}\right\}$$

$$\mathcal{F}_\times(\ell_1,\ell_2,\ell) = \left\{\begin{array}{l}r_1^{\ell} = \mathsf{ufp}(\overline{\ell_1} \times \overline{\ell_2}) - \max\Big(\mathsf{ufp}(\overline{\ell_1}) + \mathsf{ufp}(\overline{\ell_2}) - \mathsf{acc}_F(\ell_1), \mathsf{ufp}(\overline{\ell_1}) + \mathsf{ufp}(\overline{\ell_2}) - \mathsf{acc}_F(\ell_2)\Big),\\ r_2^{\ell} = \mathsf{ufp}(\underline{\ell_1} \times \underline{\ell_2}) - \max\Big(\mathsf{ufp}(\underline{\ell_1}) + \mathsf{ufp}(\underline{\ell_2}) - \mathsf{acc}_F(\ell_1), \mathsf{ufp}(\underline{\ell_1}) + \mathsf{ufp}(\underline{\ell_2}) - \mathsf{acc}_F(\ell_2)\Big),\\ r_3^{\ell} = \mathsf{ufp}(\underline{\ell_1} \times \overline{\ell_2}) - \max\Big(\mathsf{ufp}(\underline{\ell_1}) + \mathsf{ufp}(\overline{\ell_2}) - \mathsf{acc}_F(\ell_1), \mathsf{ufp}(\underline{\ell_1}) + \mathsf{ufp}(\overline{\ell_2}) - \mathsf{acc}_F(\ell_2)\Big),\\ r_4^{\ell} = \mathsf{ufp}(\overline{\ell_1} \times \underline{\ell_2}) - \max\Big(\mathsf{ufp}(\overline{\ell_1}) + \mathsf{ufp}(\underline{\ell_2}) - \mathsf{acc}_F(\ell_1), \mathsf{ufp}(\overline{\ell_1}) + \mathsf{ufp}(\underline{\ell_2}) - \mathsf{acc}_F(\ell_2)\Big),\\ i^{\ell} = (\mathsf{acc}_F(\ell_1) = \mathsf{acc}_F(\ell_2))?\ 1\ :\ 0,\ \mathsf{acc}_F(\ell) = \min\Big(r_1^{\ell} - i^{\ell}, r_2^{\ell} - i^{\ell}, r_3^{\ell} - i^{\ell}, r_4^{\ell} - i^{\ell}\Big)\end{array}\right\}$$

$$\mathcal{B}_\times(\ell_1,\ell_2,\ell) = \left\{\begin{array}{l}s_1^{\ell_1} = \mathsf{ufp}(\overline{\ell_1}) - \max\Big(\mathsf{ufp}(\overline{\ell}) - \mathsf{ufp}(\overline{\ell_2}) + 1 - \mathsf{acc}_B(\ell), \mathsf{ufp}(\overline{\ell}) - \mathsf{ufp}(\overline{\ell_2}) - \mathsf{acc}_F(\ell_2)\Big),\\ s_2^{\ell_1} = \mathsf{ufp}(\underline{\ell_1}) - \max\Big(\mathsf{ufp}(\underline{\ell}) - \mathsf{ufp}(\underline{\ell_2}) + 1 - \mathsf{acc}_B(\ell), \mathsf{ufp}(\underline{\ell}) - \mathsf{ufp}(\underline{\ell_2}) - \mathsf{acc}_F(\ell_2)\Big),\\ s_3^{\ell_1} = \mathsf{ufp}(\underline{\ell_1}) - \max\Big(\mathsf{ufp}(\underline{\ell}) - \mathsf{ufp}(\overline{\ell_2}) + 1 - \mathsf{acc}_B(\ell), \mathsf{ufp}(\underline{\ell}) - \mathsf{ufp}(\overline{\ell_2}) - \mathsf{acc}_F(\ell_2)\Big),\\ s_4^{\ell_1} = \mathsf{ufp}(\overline{\ell_1}) - \max\Big(\mathsf{ufp}(\overline{\ell}) - \mathsf{ufp}(\underline{\ell_2}) + 1 - \mathsf{acc}_B(\ell), \mathsf{ufp}(\overline{\ell}) - \mathsf{ufp}(\underline{\ell_2}) - \mathsf{acc}_F(\ell_2)\Big),\\ \mathsf{acc}_B(\ell_1) = \max(s_1^{\ell_1}, s_2^{\ell_1}, s_3^{\ell_1}, s_4^{\ell_1})\end{array}\right\}$$

Fig. 6. Constraint generation for arithmetic expressions.

$$\mathcal{C}[x :=^{\ell} e^{\ell_1}]\ \Lambda = (C, \Lambda[x \mapsto x^{\ell}])$$
$$\text{where } C = \Big(\mathcal{E}[e^{\ell_1}]\ \Lambda\Big) \cup \Big\{\mathsf{acc}_F(x^{\ell}) = \mathsf{acc}_F(\ell_1),\ \mathsf{acc}_B(x^{\ell}) = \mathsf{acc}_B(\ell_1)\Big\}$$

$$\mathcal{C}[c_1^{\ell_1}\ ;\ c_2^{\ell_2}]\ \Lambda = (C_1 \cup C_2, \Lambda_2) \text{ where } (C_1, \Lambda_1) = \mathcal{C}[c_1]\ \Lambda,\ (C_2, \Lambda_2) = \mathcal{C}[c_2]\ \Lambda_1$$

$$\mathcal{C}[\mathtt{while}^{\ell}\ e^{\ell_0}\ \mathtt{do}\ c^{\ell_1}]\ \Lambda = (C_1 \cup C_2, \Lambda') \text{ where } \left|\begin{array}{l}(C_1, \Lambda_1) = \mathcal{C}[c_1^{\ell_1}]\ \Lambda',\ \forall x \in \mathrm{Id},\ \Lambda'(x) = x^{\ell},\\ C_2 = \bigcup\limits_{x \in \mathrm{Id}} \left\{\begin{array}{l}\mathsf{acc}_F(x^{\ell}) \le \mathsf{acc}_F(\Lambda(x)),\\ \mathsf{acc}_F(x^{\ell}) \le \mathsf{acc}_F(\Lambda_1(x)),\\ \mathsf{acc}_B(x^{\ell}) \ge \mathsf{acc}_B(\Lambda(x)),\\ \mathsf{acc}_B(x^{\ell}) \ge \mathsf{acc}_B(\Lambda_1(x))\end{array}\right\}\end{array}\right.$$

$$\mathcal{C}[\mathtt{if}^{\ell}\ e^{\ell_0}\ \mathtt{then}\ c^{\ell_1}\ \mathtt{else}\ c^{\ell_2}]\ \Lambda = (C_1 \cup C_2 \cup C_3, \Lambda')$$
$$\text{where } \left|\begin{array}{l}(C_1, \Lambda_1) = \mathcal{C}[c_1^{\ell_1}]\ \Lambda, (C_2, \Lambda_2) = \mathcal{C}[c_1^{\ell_1}]\ \Lambda,\ \forall x \in \mathrm{Id},\ \Lambda'(x) = x^{\ell},\\ C_3 = \bigcup\limits_{x \in \mathrm{Id}} \left\{\begin{array}{l}\mathsf{acc}_F(x^{\ell}) = \min\big(\mathsf{acc}_F(\Lambda_1(x)), \mathsf{acc}_F(\Lambda_2(x))\big),\\ \mathsf{acc}_B(x^{\ell}) = \max\big(\mathsf{acc}_B(\Lambda_1(x)), \mathsf{acc}_B(\Lambda_2(x))\big)\end{array}\right\}\end{array}\right.$$

$$\mathcal{C}[\mathtt{require\_accuracy(}x\mathtt{,}n\mathtt{)}^{\ell}]\ \Lambda = \big\{\mathsf{acc}_B(\Lambda(x)) = n\big\}$$

Fig. 7. Constraint generation for commands.

to distinguish many assignments of the same variable or to implement joins in conditions and loops. Given a command c and an environment Λ, $\mathcal{C}[c]\ \Lambda$ returns a pair (C, Λ') made of a set C of constraints and of a new environment Λ'. $\mathcal{C}$ is

defined by induction on the structure of commands in Fig. 7. These constraint join values at control flow junctions and propagate the accuracies as described in Sect. 2. In forward mode, accuracy decreases while in backward mode accuracy increases (we weaken pre-conditions and strengthen post-conditions).

5 Experimental Results

In this section we present some experimental results obtained with our prototype. Our tool generates the constraints defined in Sect. 4 and calls the Z3 SMT solver [21] in order to obtain a solution. Since, when they exist, solutions are not unique in general, we add an additional constraint related to a cost function φ to the constraints of Figs. 6 and 7. The cost function $\varphi(c)$ of a program c computes the sum of all the accuracies of the variables and intermediary values stored in the control points of the arithmetic expressions, $\varphi(c) = \sum_{x \in \mathrm{Id},\ \ell \in \mathrm{Lab}} \mathsf{acc}(x^\ell) + \sum_{\ell \in \mathrm{Lab}} \mathsf{acc}(\ell)$. Then, by binary search, our tool searches the smallest integer P such that the system of constraints $(\mathcal{C}[c]\ \Lambda_\perp) \cup \{\varphi(c) \leq P\}$ admits a solution (we aim at using an optimizing solver in future work [6,25,28]). In our implementation we assume that, in the worst case, all the values are in double precision, consequently we start the binary search with $P \in [0, 52 \times n]$ where n is the number of variables and intermediary values stored in the control points. When a solution is found for some P, a new iteration of the binary search is run with a smaller P. Otherwise, a new iteration is run with a larger P.

We consider three sample codes displayed in Fig. 8. The first program computes the determinant $\det(M)$ of a 3×3 matrix $M = \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}$. We have $\det(M) = (a \cdot e \cdot i + d \cdot h \cdot c + g \cdot b \cdot f) - (g \cdot e \cdot c + a \cdot h \cdot f + d \cdot b \cdot i)$. The matrix coefficients belong to the ranges $\begin{pmatrix} [-10.1, 10.1] & [-10.1, 10.1] & [-10.1, 10.1] \\ [-20.1, 20.1] & [-20.1, 20.1] & [-20.1, 20.1] \\ [-5.1, 5.1] & [-5.1, 5.1] & [-5.1, 5.1] \end{pmatrix}$ and we require that the variable `det` containing the result has accuracy 10 which corresponds to a fairly rounded half precision number. By default, we assume that in the original program all the variables are in double precision. Our tool infers that all the computations may be carried out in half precision.

The second example of Fig. 8 concerns the evaluation of a degree 9 polynomial using Horner's scheme: $p(x) = a_0 + \big(x \times \big(a_1 + x \times (a_2 + \ldots)\big)\big)$. The coefficients $a_i, 0 \leq i \leq 9$ belong to $[-0.2, 0.2]$ and $x \in [-0.5, 0.5]$. Initially all the variables are in double precision and we require that the result is fairly rounded in single precision. Our tool then computes that all the variables may be in single precision but `p` which must remain in double precision. Our last example is a proportional differential controller. Initially the measure `m` is given by a sensor which sends values in $[-1.0, 1.0]$ and which ensures an accuracy of 32. All the other variables are assumed to be in double precision. As shown in Fig. 8, many variables may fit inside single precision formats.

For each program, we give in Fig. 9 the number of variables of the constraint system as well as the number of constraints generated. Next, we give the total

```
a:=b:=c:=[-10.1,10.1];
d:=e:=f:=[-20.1,20.1];
g:=h:=i:=[-5.1,5.1];
det:=(a * e * i +
 d * h * c + g * b * f)
   - (g * e * c +
 a * h * f + d * b * i);
require_accuracy
          (det,10);
```

```
a|5| :=b|5| :=c|6| :=[-10.1,10.1]|6|;
d|5| :=e|5| :=f|6| :=[-20.1,20.1]|6|;
g|5| :=h|5| :=i|6| :=[-5.1,5.1]|6|;
det|10| :=(a|5| *|6| e|5| *|8| i|6| +|9|
 d|5| *|6| h|5| *|8| c|6| +|9| g|5| *|6| b|5| *|8| f|6|)
   -|10| (g|5| *|6| e|5| *|8| c|6| +|9|
 a|5| *|6| h|5| *|8| f|6| +|9| d|5| *|6| b|5| *|8| i|6|);
require_accuracy(det,10);
```

```
half a,b,c,d,e;
half f,g,h,i,det;
//init a,b,c,d,e,
//    f,g,h and i
det:=(a * e * i +
      d * h * c +
      g * b * f)
   - (g * e * c +
      a * h * f +
      d * b * i);
```

```
a:=array
   (10,[-0.2,0.2]#53);
x:=[0.0,0.5]#53;
p:=0.0; i:=0;
while(i<10) {
  p:=p * x + a[i];
};
require_accuracy(p,23);
```

```
a|23| :=array(10,[-0.2,0.2]|23|);
x|23| :=[0.0,0.5]|23|;
p|23| :=0.0|23|; i := 0;
while(i<10) {
  p|24| :=p|23| *|23| x|23| +|24| a[i]|23|;
};
require_accuracy(p,23);
```

```
float a[10];
float x,tmp;
double p;
// init a and x
p:=0.0; i:=0;
while(i<10) {
  tmp:=p * x;
  p:=tmp + a[i];};
```

```
m:=[-1.0,1.0]#32;
kp:=0.194; kd:=0.028;
invdt:=10.0; c:=0.5;
e0:=0.0;
while (true) {
  e:=c - m;
  p:=kp * e;
  d:=kd*invdt*(e-e0);
  r:=p + d;
  e0:=e;
};
require_accuracy(r,23);
```

```
m|21| :=[-1.0,1.0]|21|;
kp|21| :=0.194|21|;kd|20| :=0.028|20|;
invdt|20| :=10.0|20|;
c|21| :=0.5|21|;e0|21| :=0.0|21|;
while (true) {
  e|21| :=c|21| -|21| m|22|;
  p|22| :=kp|21| *|22| e|21|;
  d|23| :=kd|20| *|22| invdt|20|
        *|23| (e|21| -|22| e0|21|);
  r|23| :=p|22| +|23| d|23|;e0|21| :=e|21|;};
require_accuracy(r,23);
```

```
volatile float m;
float kp,kd,p,d,r;
float invdt,c,e0;
double e,tmp;
kp:=0.194;kd:=0.028;
invdt:=10.0; c:=0.5;
e0:=0.0 ;
while (true) {
  e:=c-m;p:=kp*e;
  tmp:=e - e0;
  d:=kd * invdt;
  d:=d * tmp;
  r:=p + d; e0:=e;};
```

Fig. 8. Examples of mixed-precision inference. Source programs, inferred accuracies and formats. Top: 3×3 determinant. Middle: Horner's scheme. Bottom: a PD controller.

Program	#Var.	#Constr.	Time(s)	#Bits-Init.	#Bits-Optim.	Z3-Calls
Linear filter	239	330	0.31	1534	252	12
Determinant	604	775	0.45	2912	475	14
Horner	129	179	0.18	884	346	11
PD Controller	388	530	0.49	2262	954	12

Fig. 9. Measures of efficiency of the analysis on the codes of Figs. 2 and 8.

execution time of the analysis (including the generation of the system of constraints and the calls to the SMT solver done by the binary search). Then we give the number of bits needed to store all the values of the programs, assuming that all the values are stored in double precision (column #Bits-Init.) and as computed by our analysis (column #Bits-Optim.) Finally, the number of calls to the SMT solver done during the binary search is displayed. Globally, we can observe that the numbers of variables and constraints are rather small and very tractable for the solver. This is confirmed by the execution times which are very short. The improvement, in the number of bits needed to fulfill the requirements, compared to the number of bits needed if all the computations are done in double precision, ranges from 57% to 83% which is very important.

6 Conclusion

We have defined a static analysis which determines the floating-point formats needed to ensure a given accuracy. This analysis is done by generating a set of linear constraints between integer variables only, even if the programs contain non-linear computations. These constraints are easy to solve by a SMT solver.

Our technique can be easily extended to other language structures. For example, since all the elements of an array must have the same type, we just need to join all the elements in a same abstract value to obtain a relevant result. Similarly, functions are also easy to manage since only one type per argument and returned value need. Our analysis is built upon a range analysis performed before. Obviously, the precision of this analysis impacts the precision of the floating-point format determination and the inference of sharp ranges given by relational domains, improves the quality of the results. In future work, we aim at exploring the use a solver based on optimization modulo theories [6,25,28] instead of the non-optimizing solver coupled to a binary search used presently.

References

1. Patriot missile defense: Software problem led to system failure at Dhahran, Saudi Arabia. Technical Report GAO/IMTEC-92-26, General Accounting office (1992)
2. ANSI/IEEE: IEEE Standard for Binary Floating-Point Arithmetic (2008)
3. Barr, E.T., Vo, T., Le, V., Su, Z.: Automatic detection of floating-point exceptions. In: POPL 2013, pp. 549–560. ACM (2013)
4. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 825–885. IOS Press (2009)
5. Bertrane, J., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Rival, X.: Static analysis by abstract interpretation of embedded critical software. ACM SIGSOFT Softw. Eng. Notes **36**(1), 1–8 (2011)
6. Bjørner, N., Phan, A.-D., Fleckenstein, L.: νZ - an optimizing SMT solver. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 194–199. Springer, Heidelberg (2015). doi:10.1007/978-3-662-46681-0_14
7. Chiang, W., Baranowski, M., Briggs, I., Solovyev, A., Gopalakrishnan, G., Rakamaric, Z.: Rigorous floating-point mixed-precision tuning. In: POPL, pp. 300–315. ACM (2017)
8. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Principles of Programming Languages, pp. 238–252. ACM Press (1977)
9. Cousot, P., Cousot, R.: A gentle introduction to formal verification of computer systems by abstract interpretation. NATO Science Series III: Computer and Systems Sciences, pp. 1–29. IOS Press (2010)
10. Damouche, N., Martel, M., Chapoutot, A.: Intra-procedural optimization of the numerical accuracy of programs. In: Núñez, M., Güdemann, M. (eds.) FMICS 2015. LNCS, vol. 9128, pp. 31–46. Springer, Cham (2015). doi:10.1007/978-3-319-19458-5_3
11. Darulova, E., Kuncak, V.: Sound compilation of reals. In: Symposium on Principles of Programming Languages, POPL 2014, pp. 235–248. ACM (2014)

12. Gao, X., Bayliss, S., Constantinides, G.A.: SOAP: structural optimization of arithmetic expressions for high-level synthesis. In: International Conference on Field-Programmable Technology, pp. 112–119. IEEE (2013)
13. Goubault, E.: Static analysis by abstract interpretation of numerical programs and systems, and FLUCTUAT. In: Logozzo, F., Fähndrich, M. (eds.) SAS 2013. LNCS, vol. 7935, pp. 1–3. Springer, Heidelberg (2013). doi:10.1007/978-3-642-38856-9_1
14. Goubault, E., Putot, S.: Static analysis of finite precision computations. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 232–247. Springer, Heidelberg (2011). doi:10.1007/978-3-642-18275-4_17
15. Halfhill, T.R.: The truth behind the Pentium bug. Byte, March 1995
16. Lam, M.O., Hollingsworth, J.K., de Supinski, B.R., LeGendre, M.P.: Automatically adapting programs for mixed-precision floating-point computation. In: Supercomputing, ICS 2013, pp. 369–378. ACM (2013)
17. Lamotte, J.L., Chesneaux, J.M., Jézéquel, F.: CADNA_C: a version of CADNA for use with C or C++ programs. Comput. Phys. Commu. **181**(11), 1925–1926 (2010)
18. Martel, M.: Semantics of roundoff error propagation in finite precision calculations. High.-Order Symb. Comput. **19**(1), 7–30 (2006)
19. Martel, M., Najahi, A., Revy, G.: Code size and accuracy-aware synthesis of fixed-point programs for matrix multiplication. In: Pervasive and Embedded Computing and Communication Systems, pp. 204–214. SciTePress (2014)
20. Miné, A.: Inferring sufficient conditions with backward polyhedral under-approximations. Electr. Notes Theor. Comput. Sci. **287**, 89–100 (2012)
21. Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). doi:10.1007/978-3-540-78800-3_24
22. Muller, J.M.: On the definition of ulp(x). Technical report 2005–09, Laboratoire d'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon (2005)
23. Muller, J.M., Brisebarre, N., de Dinechin, F., Jeannerod, C.P., Lefèvre, V., Melquiond, G., Revol, N., Stehlé, D., Torres, S.: Handbook of Floating-Point Arithmetic. Birkhäuser Boston, Boston (2010)
24. Nguyen, C., Rubio-Gonzalez, C., Mehne, B., Sen, K., Demmel, J., Kahan, W., Iancu, C., Lavrijsen, W., Bailey, D.H., Hough, D.: Floating-point precision tuning using blame analysis. In: International Conference on Software Engineering (ICSE). ACM (2016)
25. Nieuwenhuis, R., Oliveras, A.: On SAT modulo theories and optimization problems. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 156–169. Springer, Heidelberg (2006). doi:10.1007/11814948_18
26. Panchekha, P., Sanchez-Stern, A., Wilcox, J.R., Tatlock, Z.: Automatically improving accuracy for floating point expressions. In: PLDI, pp. 1–11. ACM (2015)
27. Rubio-Gonzalez, C., Nguyen, C., Nguyen, H.D., Demmel, J., Kahan, W., Sen, K., Bailey, D.H., Iancu, C., Hough, D.: Precimonious: tuning assistant for floating-point precision. In: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 27:1–27:12. ACM (2013)
28. Sebastiani, R., Tomasi, S.: Optimization modulo theories with linear rational costs. ACM Trans. Comput. Log. **16**(2), 12:1–12:43 (2015)
29. Solovyev, A., Jacobsen, C., Rakamarić, Z., Gopalakrishnan, G.: Rigorous estimation of floating-point round-off errors with symbolic Taylor expansions. In: Bjørner, N., de Boer, F. (eds.) FM 2015. LNCS, vol. 9109, pp. 532–550. Springer, Cham (2015). doi:10.1007/978-3-319-19249-9_33