Improving the numerical accuracy of programs by automatic transformation

Nasrine Damouche[1] · **Matthieu Martel**[1] · **Alexandre Chapoutot**[2]

Abstract The dangers of programs performing floating-point computations are well known. This is due to the sensitivity of the results to the way formulæ are written. These last years, several techniques have been proposed concerning the transformation of arithmetic expressions in order to improve their numerical accuracy and, in this article, we go one step further by automatically transforming larger pieces of code containing assignments and control structures. We define a set of transformation rules allowing the generation, under certain conditions and in polynomial time, of larger expressions by performing limited formal computations, possibly among several iterations of a loop. These larger expressions are better suited to improve, by re-parsing, the numerical accuracy of the program results. We use abstract interpretation-based static analysis techniques to over-approximate the round-off errors in programs and during the transformation of expressions. A tool has been implemented and experimental results are presented concerning classical numerical algorithms and algorithms for embedded systems.

Keywords Program transformation · Floating-point numbers · Static analysis · IEEE754 standard · Numerical accuracy

✉ Nasrine Damouche
nasrine.damouche@univ-perp.fr

Matthieu Martel
matthieu.martel@univ-perp.fr

Alexandre Chapoutot
alexandre.chapoutot@ensta-paristech.fr

[1] LAMPS Laboratory, University of Perpignan Via Domitia, Perpignan, France

[2] U2IS, ENSTA ParisTech, Université Paris-Saclay, 828 bd des Maréchaux, 91762 Palaiseau cedex, France

1 Introduction

These last years, as the complexity of the floating-point computations carried out in embedded systems and elsewhere increased, numerical accuracy has become a more and more sensible subject in computer science. Due to the important impact of accuracy on the reliability of embedded systems, many industries encourage research to validate [6,16,18,19], verify [15,36] and improve [24,31] their software in order to avoid failures and eventually disasters in aeronautics, automotive, robotics, etc.

In this article, we focus on the transformation [9,14] of intra-procedural pieces of code in order to automatically improve their accuracy. For automatic transformation of single arithmetic expressions, several techniques have already been proposed. We can mention [24] which introduces a new intermediary representation (IR) that manipulates in a single data structure a large set of equivalent arithmetic expressions. This IR, called APEG [24,31] for Abstract Program Expression Graph, succeeds to reduce the complexity of the transformation in polynomial size and time. Starting from this state of the art, we aim at going a step further by automatically transforming larger pieces of code. Our interest is to transform automatically sequences of commands that contain assignments and control structures in order to improve their numerical accuracy. This transformation consists in optimizing a *target variable* with respect to some given ranges for the input variables of the program. Accuracy bounds are computed by abstract interpretation [8] techniques for the floating-point arithmetic [12,30].

We start by motivating our work with a case study concerning an algorithm frequently used in robotics for odometry. We show how to rewrite it into another program which is numerically more accurate but semantically equivalent, in the sense that both programs compute the same function in

exact arithmetic. More generally, our transformation operates by simplifying and developing expressions and inlining them into other expressions. This allows one to generate new formulæ. We also rewrite the codes by unfolding the body of loops, in order to have more computations inside a single iteration. The rewriting rules used to automatically rewrite codes and their proof of correctness are the main contribution of this article. These rules are presented as sequents containing conditions under which transformations may be applied without breaking the semantic equivalence between the source and target programs. In addition, these rules are deterministically applied, yielding a polynomial time transformation. Our work is completed by experimental results involving the transformation of codes coming from multiple domains of sciences.

This article is organized as follows. We start in Sect. 2 by introducing what kind of transformations we do through a case study. Section 3 introduces related work concerning the analysis and transformation of arithmetic expressions. Next, we give in Sect. 4 the set of transformation rules for commands together with the conditions required to preserve the semantic equivalence of programs. In Sect. 5, we present the correctness proof of the transformation rules. Section 6 performs experimental results and shows various experimentations obtained using our tool. Finally, Sect. 7 concludes and presents future work.

2 Case study: odometry

In this section, we are interested in an example taken from robotics and whose code is given in Listing 1. It concerns the computation of the position (x, y) of a two-wheeled robot by odometry. Given the instantaneous rotation speeds s_l and s_r of the left and right wheels, it aims at computing the position of the robot in a Cartesian space. Let C be the circumference of the wheels of the robot and L the length of its axle (see Fig. 1). We assume that s_l and s_r, are coming from sensors, are updated by the system, by side-effect. The computation of the position of the robot is given by

$$x(t+1) = x(t) + \Delta d(t+1) \times \cos\left(\theta(t) + \frac{\Delta\theta(t+1)}{2}\right), \tag{1}$$

$$y(t+1) = y(t) + \Delta d(t+1) \times \sin\left(\theta(t) + \frac{\Delta\theta(t+1)}{2}\right), \tag{2}$$

with

$$\theta(t+1) = \theta(t) + \Delta\theta(t), \tag{3}$$

$$\Delta d(t) = \big(\Delta d_r(t) + \Delta d_l(t)\big) \times 0.5, \tag{4}$$

Listing 1 Listing of the initial odometry program

```
sl = [0.52,0.53]; sr = 0.785398163397;
 theta = 0.0;
t = 0.0; x = 0.0; y = 0.0; inv_l = 0.1;
c = 12.34;
while (t < 100.0) do {
 delta_dl = (c * sl);
 delta_dr = (c * sr);
 delta_d = ((delta_dl + delta_dr) * 0.5);
 delta_theta = ((delta_dr - delta_dl) * inv_l);
 arg = (theta + (delta_theta * 0.5));
 cos = (1.0 - ((arg * arg) * 0.5))
     + ((((arg * arg)* arg)* arg) / 24.0);
 x = (x + (delta_d * cos));
 sin = (arg - (((arg * arg)* arg)/6.0))
     + (((((arg * arg)* arg)* arg)* arg)/120.0);
 y = (y + (delta_d * sin));
 theta = (theta + delta_theta);
 t = (t + 0.1);
}
```

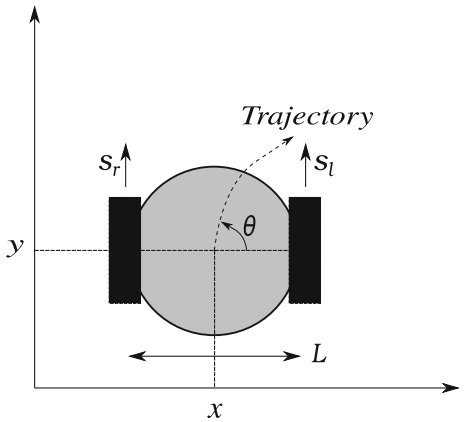

Fig. 1 Parameters of the two-wheeled robot

$$\Delta d_l(t) = s_l(t) \times C, \tag{5}$$

$$\Delta d_r(t) = s_r(t) \times C, \tag{6}$$

$$\Delta\theta(t) = \big(\Delta d_r(t) - \Delta d_l(t)\big) \times \frac{1}{L}. \tag{7}$$

In Eqs. (1)–(7), $\theta(t)$ is the direction of the robot, $d(t)$ is the elementary movement of the robot at time t and $d_l(t)$ and $d_r(t)$ are the elementary movements of the left and right wheels. We assume that cos and sin, not computed by a library, are obtained by a Taylor series expansion as shown in Eq. (8).

$$\cos(x) \approx 1 - \tfrac{x^2}{2!} + \tfrac{x^4}{4!}, \quad \sin(y) \approx x - \tfrac{x^3}{3!} + \tfrac{x^5}{5!}. \tag{8}$$

We aim at rewriting the initial program described in Listing 1 into a better program which improves the numerical accuracy of the computed position. The speed of the left wheel is assumed to belong to an interval of $[0.52, 0.53]$ radians per second ($\frac{\pi}{6} \approx 0.523598$) so that the program is optimized for a range of values of s_l and not only for a single value. Our prototype develops and simplifies the expressions Δ_d, cos and sin and then inlines them within the loop, in

Listing 2 Listing of the transformed odometry program

```
sl = [0.52,0.53]; theta = 0.0; y = 0.0; x
= 0.0; t = 0.0;
while (t < 100.0) do {
 TMP_6 = (0.1 * (0.5 * (9.691813336318980
       - (12.34 * sl))));
 TMP_23 = ((theta + (((9.691813336318980 - (sl
        * 12.34)) * 0.1) * 0.5)) * (theta
        + (((9.691813336318980 - (sl * 12.34))
        * 0.1) * 0.5)));
 TMP_25 = ((theta + TMP_6) * (theta + TMP_6))
        * (theta + (((9.691813336318980
        - (sl * 12.34)) * 0.1) * 0.5));
 TMP_26 = (theta + TMP_6) ;
 x = ((0.5 * (((1.0 - (TMP_23 * 0.5)) +
((TMP_25   * TMP_26) / 24.0)) * ((12.34 * sl)
   + 9.691813336318980))) + x);
 TMP_27 = ((TMP_26 * TMP_26) * (theta
        + (((9.691813336318980 - (sl * 12.34))
        * 0.1) * 0.5)));
 TMP_29 = (((TMP_26 * TMP_26) * TMP_26) *
(theta
        + (((9.691813336318980 - (sl * 12.34))
        * 0.1) * 0.5)));
 y = (((9.691813336318980 + (12.34 * sl))
   * (((TMP_26 - (TMP_27 / 6.0)) + ((TMP_29
   * TMP_26) / 120.0)) * 0.5)) + y);
 theta = (theta + (0.1 * (9.691813336318980
       - (12.34 * sl))));
 t = t + 0.1;
}
```

x and y. In addition, it creates new intermediary variables, called `TMP`, in order to avoid to have too large expressions. This transformation offers several advantages:

- it creates large enough formulæ well-suited to be efficiently rewritten by existing techniques for the transformation of arithmetic expressions, based on the use of abstract program expression graphs [24,31].
- it may create static formulæ made of constant terms which can be evaluated statically in an extended precision arithmetic. This may also reduce the number of operations in the target program and then optimizes its execution time (see [11] and Sect. 6).

Using our tool, we obtain the final program given in Listing 2. If we compare the resulting values x_o and x_t of the original and transformed odometry programs, we observe that the transformation leads to a significant difference in the accuracy of the computations, as shown in Table 1. The results show an important difference on the third or even on the second digit of the decimal values of the result. The difference in the computed trajectory of the robot is shown in Fig. 2.

In previous work [14], we have detailed how to transform a PID controller by hand and the results obtained were significantly important in industrial contexts where end-users sometimes want ten decimal correct digits for this algorithm. In this article, the transformation of the odometry code has been achieved automatically by using our tool. The results show that the accuracy is augmented by several percents [12].

Table 1 Values of x before and after transformation of odometry program at the first iterations

It	x_o (initial code)	x_t (transformed code)
1	8.681698	8.444116
2	17.038230	16.589474
3	24.756744	24.147995
4	31.549016	30.852965
5	37.163761	36.469708
6	41.398951	40.806275
7	44.114126	43.724118
8	45.242707	45.148775

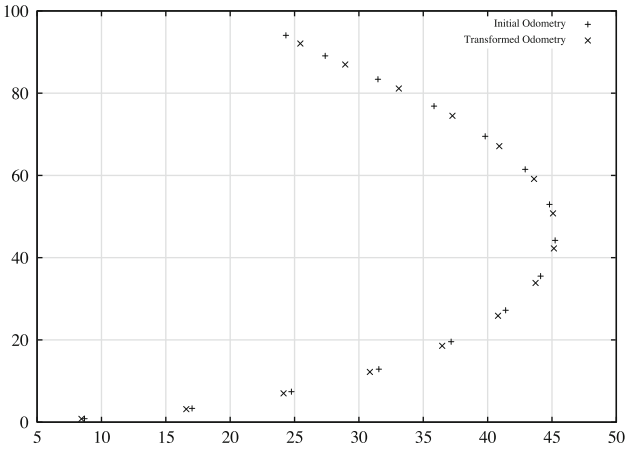

Fig. 2 Computed trajectories by the initial and the transformed odometry programs

We have experimented our tool on various examples coming from different areas such as control-command programs and numerical algorithms [26]. Generally, the relative accuracy is increased by 15 % at least. We have also tested the ability of our tool to generate accurate code on a larger program concerning the simulation of the trajectory of a rocket and whose size is about 100 lines of code [13]. We have shown that we optimize it by up to 25 %. Finally, in recent work, we have demonstrated the usefulness of our tool to accelerate the convergence of numerical iterative methods [11]. Among these methods we find an iterative Gram–Schmidt method as well as Jacobi's and Newton–Raphson's method. We have succeed to reduce significantly the number of iterations needed by each of the methods to converge and, consequently, we have improved the execution time required by each of them.

3 Analysis and transformation of expressions

In this section, we provide a brief summary of the methods that we use to bound and reduce the errors on arithmetic expressions, see [24] for more details. Section 3.1 is dedi-

Table 2 Basic IEEE754 floating-point formats

Format	Name	p	e	e_{min}	e_{max}
Binary 16	Half precision	11	5	−14	+15
Binary 32	Single precision	24	8	−126	+127
Binary 64	Double precision	53	11	−1122	+1223
Binary 128	Quadratic precision	113	15	−16382	+16383

cated to static analysis of numerical accuracy, while Sect. 3.2 concerns the transformation of arithmetic expressions.

3.1 Static analysis of the numerical accuracy

The floating-point arithmetic is defined by the IEEE754 Standard [2,34]. Floating-point numbers are used to encode real numbers. However, because they are a finite representation of their mathematical cousins, round-off errors arise during computations. A floating-point number x is defined by

$$x = s \cdot (d_0.d_1 \dots d_{p-1}) \cdot \beta^e = s \cdot m \cdot \beta^{e-p+1}, \tag{9}$$

where $s \in \{-1, 1\}$ is the sign, $m = d_0 d_1 \cdots d_{p-1}$ is the mantissa with digits $0 \leq d_i < \beta, 0 \leq i \leq p-1$, p is the precision and e is the exponent, $e_{\min} \leq e \leq e_{\max}$. The IEEE754 Standard specifies several formats for floating-point numbers by providing specific values for p, β, $e_{\min}$ and $e_{\max}$ as described in Table 2. It also defines some rounding modes, towards $+\infty$, $-\infty$, 0 and to the nearest. Let $\mathbb{F}$ be the set of floating-point numbers used by the program for the current operation (e.g., binary 32 or binary 64 a.k.a single or double precision) and let us write $\uparrow_{+\infty}$, $\uparrow_{-\infty}$, $\uparrow_0$ and $\uparrow_\sim$ for the rounding functions, the IEEE754 Standard defines the semantics of the elementary operations, with $\uparrow_r : \mathbb{R} \to \mathbb{F}$, by

$$x \circledast_r y = \uparrow_r (x * y), \tag{10}$$

where $\circledast_r$ denotes a floating-point operation $+$, $-$, $\times$ or $\div$ computed using the rounding mode r and $*$ denotes an exact operation. Because of the round-off errors, the results of the computations are not exact. In this article, we also use the function $\downarrow_r : \mathbb{R} \to \mathbb{R}$ which returns the round-off errors. We have

$$\downarrow_r (x) = x - \uparrow_r (x). \tag{11}$$

Our transformation technique, introduced in Sect. 3, is independent of the selected rounding mode and, in this article, for the sake of simplicity, we assume that all the floating-point computations are done by using the rounding mode to the nearest. We write $\uparrow$ and $\downarrow$ instead of $\uparrow_r$ and $\downarrow_r$ whenever the rounding mode r does not matter.

We present now the computation of errors during the evaluation of arithmetic expressions [30]. Formally, we use values which are pairs $(x, \mu) \in \mathbb{F} \times \mathbb{R} = \mathbb{E}$, where x denotes the floating-point number used by the machine and μ denotes the exact error attached to $\mathbb{F}$, i.e., the exact difference between the real and floating-point numbers as defined in Eq. (11). For example, the real number $\frac{1}{3}$ is represented by the value $v = \left(\uparrow_\sim \left(\frac{1}{3}\right), \downarrow_\sim \left(\frac{1}{3}\right)\right) = \left(0.333333, \left(\frac{1}{3} - 0.333333\right)\right)$. The semantics of the elementary operations on $\mathbb{E}$ is defined in [30].

We do not wish to optimize our programs for a given data set, but for all the possible values of the inputs. Consequently, our tool uses an abstract semantics [8] based on $\mathbb{E}$. The abstract values are represented by a pair of intervals. The first interval contains the range of the floating-point values of the program and the second one contains the range of the errors obtained by subtracting the floating-point values from the exact ones. In the abstract value denoted by $(x^\sharp, \mu^\sharp) \in \mathbb{E}^\sharp$, we have $x^\sharp$ the interval corresponding to the range of the values and $\mu^\sharp$ the interval of errors on $x^\sharp$. This value abstracts a set of concrete values $\{(x, \mu) : \ x \in x^\sharp \text{ and } \mu \in \mu^\sharp\}$ by intervals in a component-wise way. We introduce now the semantics of arithmetic expressions on $\mathbb{E}^\sharp$. We approximate an interval $x^\sharp$ with real bounds by an interval based on floating-point bounds, denoted by $\uparrow^\sharp (x^\sharp)$. Here bounds are rounded to the nearest (see Eq. (12)).

$$\uparrow^\sharp ([\underline{x}, \overline{x}]) = [\uparrow (\underline{x}), \uparrow (\overline{x})]. \tag{12}$$

In the other direction, we have the function $\downarrow^\sharp$ that abstracts the concrete function $\downarrow$. It over-approximates the set of exact values of the error $\downarrow (x) = x - \uparrow (x)$. Every error associated to $x \in [\underline{x}, \overline{x}]$ is included in $\downarrow^\sharp ([\underline{x}, \overline{x}])$. We also have for a rounding mode to the nearest

$$\downarrow^\sharp ([\underline{x}, \overline{x}]) = [-y, y] \quad \text{with} \quad y = \frac{1}{2}\text{ulp}\big(\max(|\underline{x}|, |\overline{x}|)\big). \tag{13}$$

Formally in Eq. (13), the *unit in the last place*, denoted by ulp(x), is the weight of the least significant digit of the floating-point number x. Equations (14) to (16) give the semantics of the addition, subtraction and multiplication over $\mathbb{E}^\sharp$, for other operations see [30]. If we sum two floating-point numbers, we must add the errors on the operands to the error produced by the round-off of the result. If we subtract two floating-point numbers, we must subtract errors on the operands and add the error produced by the round-off of the result. When multiplying two floating-point numbers, the semantics is given by the development of $(x_1^\sharp + \mu_1^\sharp) \times (x_2^\sharp + \mu_2^\sharp)$.

$$(x_1^\sharp, \mu_1^\sharp) + (x_2^\sharp, \mu_2^\sharp) = \left(\uparrow^\sharp (x_1^\sharp + x_2^\sharp), \mu_1^\sharp + \mu_2^\sharp + \downarrow^\sharp (x_1^\sharp + x_2^\sharp) \right), \tag{14}$$

$$(x_1^\sharp, \mu_1^\sharp) - (x_2^\sharp, \mu_2^\sharp) = \left(\uparrow^\sharp (x_1^\sharp - x_2^\sharp), \mu_1^\sharp - \mu_2^\sharp + \downarrow^\sharp (x_1^\sharp - x_2^\sharp) \right), \tag{15}$$

$$(x_1^\sharp, \mu_1^\sharp) \times (x_2^\sharp, \mu_2,^\sharp) = \left(\uparrow^\sharp (x_1^\sharp \times x_2^\sharp), x_2^\sharp \times \mu_1^\sharp + x_1^\sharp \times \mu_2^\sharp + \mu_1^\sharp \times \mu_2^\sharp + \downarrow^\sharp (x_1^\sharp \times x_2^\sharp) \right). \tag{16}$$

Example 1 In order to illustrate the semantics of the elementary operations given in Eqs. (14) to (16), we consider the product of two floating-point numbers x_1 and x_2 with initial errors on them of, respectively, μ_1 and μ_2. In our example, we have chosen that

$$([\underline{x_1}, \overline{x_1}], [\underline{\mu_1}, \overline{\mu_1}]) = ([3.141, 3.142], [0.00059, 0.000592])$$

and

$$([\underline{x_2}, \overline{x_2}], [\underline{\mu_2}, \overline{\mu_2}]) = ([99.98, 99.99], [0.09, 0.1]).$$

The computation is processed as following:

- First, we multiply x_1 by x_2. The product [3.141, 3.142] $\times$ [99.98, 99.99] gives [314.03718, 314.16858]. For the sake of simplicity, we use in this example a simplified system of floating-point numbers whose mantissas are made of four decimal digits. So, we keep only four digits, the result is $\uparrow^\sharp (x_1^\sharp \times x_2^\sharp)$ = [314.0, 314.2],
- Next, the global error generated by the product is equals to [0.3790174, 0.40487328], obtained by the sum of:
  - The first floating-point interval x_1 multiplied by the error μ_2 gives [0.28269, 0.3142],
  - The second floating-point interval x_2 multiplied by the error μ_1 results [0.0589882, 0.05919408],
  - The error μ_1 multiplied by the error μ_2 is equals to [0.0000531, 0.0000592],
  - The error generated by multiplying x_1 and x_2 which is equals to [0.03718, 0.03142] in our case because we have chosen to keep only four digits for the floating-point numbers.

  We assume that $y = \max([|0.3790174|, |0.40487328|]) = 0.40487328$. Let us consider that $u = \text{ulp(y)}$. We have $\text{ulp(y)} = 2^{\text{ufp(y)}-p}$, where ufp() and p, respectively, are the unit in the first place and the precision of the number in question. We have that, in binary, 0.40487328 is equals to $2^{-2} + 2^{-3} + 2^{-6} + 2^{-7} + 2^{-8} + \ldots$. This results that $\text{ufp}(0.40487328) = -2$. Note that, by taking $p = 4$, we find that $\text{ulp(y)} = 2^{-2-4} = 2^{-6}$. In our case, we have chosen to use the rounding mode to the nearest, ($\downarrow (x) = \frac{1}{2}\text{ulp(x)}$), which gives us at the last the following interval

$$\downarrow^\sharp ([0.40487328, 0.40487328]) = [2^{-7}, 2^7].$$

□

Note that more precise abstract domains than intervals exist, e.g., [6,18,19] as well as complementary techniques [4,5,15,36]. Let us also mention that other methods exist to transform, synthesize or repair arithmetic expressions in the integer or fixed arithmetic [21]. We cite also [6, 20,32,33,36] which are interesting to improve the ranges of the floating-point variables which could be complementary of our approach.

3.2 Accuracy improvement of expressions

In this section, we briefly present former work to semantically transform [9] arithmetic expressions using Abstract Program Equivalent Graphs (APEG) [24]. This intermediary representation (IR) of programs is an extension of another IR called equivalent program expression graphs [38,39]. EPEGs have been introduced to address the phase of ordering problems in compilers [1,7,29,40], that is the problem of determining in which order to apply the optimizations of compilers to obtain the best result. For example, this makes it possible to search for the maximal shared sub-expressions [27,37].

Contrary to EPEGs, APEGs make it possible to remain in polynomial size while dealing with an exponential number of equivalent expressions. To prevent any combinatorial problem, APEGs hold in abstraction boxes many equivalent expressions up to associativity and commutativity. A box containing n operands can represent up to $1 \times 3 \times 5 \cdots \times (2n-3)$ possible formulæ. In order to build large APEGs, two algorithms are used *propagation* and *expansion* algorithm. The first one searches recursively in the APEG where a symmetric binary operator is repeated and introduces abstraction boxes. Then, the second algorithm finds a homogeneous part and inserts a polynomial number of boxes. In order to add new shapes of expressions in an APEG, one propagates recursively subtractions and divisions into the concerned operands, propagate products, and factorizing common factors. Finally, an accurate formula is searched among all the equivalent formulæ represented in an APEG using the abstract semantics of Sect. 3.1.

In their article on EPEGs, Tate et al. use rewriting rules to extend the structure up to saturation [38,39]. In our context, such rules would consist of performing some pattern matching in an existing APEG p and then adding new nodes in p, once a pattern has been recognized. For example, the rules corresponding to distributivity and box construction are given in Fig. 3. An alternative technique for APEG construc-

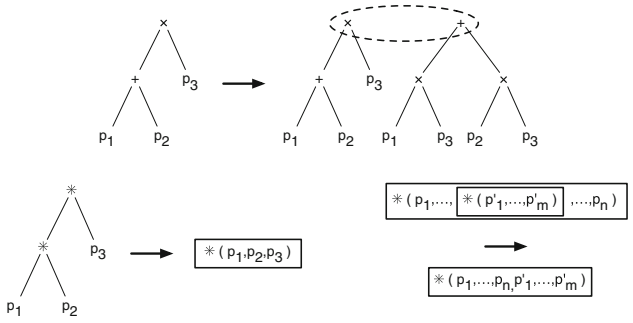

Fig. 3 Some rules for APEG construction by pattern matching

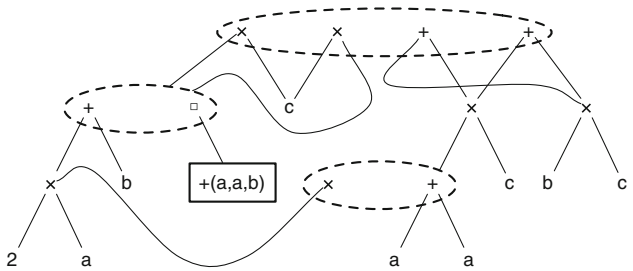

Fig. 4 APEG for the expression $e = \big((a + a) + c\big) \times c$

tion is to use dedicated algorithms. Such algorithms, working in polynomial time, have been proposed in [24].

Let Expr and BExpr be the set of arithmetic and boolean expressions, respectively. We have

$$\begin{aligned} \mathsf{Expr} \ni e &::= id \mid cst \mid e + e \mid e - e \mid e \times e \mid e \div e. \\ \mathsf{BExpr} \ni b &::= \mathsf{true} \mid \mathsf{false} \mid b \wedge b \mid b \vee b \mid \neg b \mid e = e \mid \\ &\quad e < e \mid e > e. \end{aligned} \tag{17}$$

In Eq. (17), id is an identifier in a finite set $\mathscr{V}$ and cst a constant in $\mathbb{R}$.

The APEGs are an extension of the Equivalence Program Expression Graphs (EPEGs) introduced by Tate et al. [38, 39]. An APEG is defined inductively as follows:

1. A constant cst or an identifier id is an APEG,
2. An expression $p_1 * p_2$ is an APEG, where p_1 and p_2 are APEGs and $*$ is a binary operator among $\{+, -, \times, \div\}$,
3. A box $\boxed{*(p_1, \ldots, p_n)}$ is an APEG, where $* \in \{+, \times\}$ is a commutative and associative operator and the $p_{i, 1 \leq i \leq n}$, are APEGs,
4. A non-empty set $\{p_1, \ldots, p_n\}$ of APEGs consists in an APEG, where $p_{i, 1 \leq i \leq n}$, is not a set of APEGs itself. We call the set $\{p_1, \ldots, p_n\}$ the equivalence class.

An example of APEG is given in Fig. 4. When an equivalence class (denoted by a dotted ellipse in Fig. 4) contains many APEGs $p_1, \ldots, p_n$ then one of the p_i, $1 \leq i \leq n$ may be selected in order to build an expression. A box $\boxed{*(p_1, \ldots, p_n)}$ represents any parsing of the expression $p_1 * \ldots * p_n$. From an implementation point of view, when several equivalent expressions share a common sub-expression, the latter is represented only once in the APEG. Then APEGs provide a compact representation of a set of equivalent expressions and make it possible to represent in an unique structure many equivalent expressions of very different shapes. For readability reasons, in Fig. 4, the leaves corresponding to the variables a, b and c are duplicated while, in practice, they are defined only once in the structure.

The set $\mathscr{A}(p)$ of expressions contained inside an APEG p is defined inductively as follows:

1. If p is a constant cst or an identifier id then $\mathscr{A}(p) = \{cst\}$ or $\mathscr{A}(p) = \{x\}$,
2. If p is an expression $p_1 * p_2$ then

$$\mathscr{A}(p) = \bigcup_{e_1 \in \mathscr{A}(p_1),\ e_2 \in \mathscr{A}(p_2)} e_1 * e_2,$$

3. If p is a box $\boxed{*(p_1, \ldots, p_n)}$ then $\mathscr{A}(p)$ contains all the parsings of $e_1 * \ldots * e_n$ for all $e_1 \in \mathscr{A}(p_1), \ldots, e_n \in \mathscr{A}(p_n)$,
4. If p is an equivalence class then $\mathscr{A}(p) = \bigcup_{1 \leq i \leq n} \mathscr{A}(p_i)$.

For instance, the APEG p of Fig. 4 represents all the following expressions:

$$\mathscr{A}(p) = \left\{ \begin{array}{ll} \big((a + a) + b\big) \times c, & \big((a + b) + a\big) \times c, \\ \big((b + a) + a\big) \times c, & \big((2 \times a) + b\big) \times c, \\ c \times \big((a + a) + b\big), & c \times \big((a + b) + a\big), \\ c \times \big((b + a) + a\big), & c \times \big((2 \times a) + b\big), \\ (a + a) \times c + b \times c, & (2 \times a) \times c + b \times c, \\ b \times c + (a + a) \times c, & b \times c + (2 \times a) \times c \end{array} \right\}. \tag{18}$$

4 Transformation of commands

In this section, we describe the formal rules used to transform intra-procedural pieces of code.

The syntax of commands is given in Eq. (19). It corresponds to the core of an imperative language.

$$\begin{aligned} \mathsf{Com} \ni c ::= &\ id = e \mid c_1; c_2 \mid \mathsf{if}_\Phi\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \\ &\mid \mathsf{while}_\Phi\ e\ \mathsf{do}\ c \mid \mathsf{nop}. \end{aligned} \tag{19}$$

Our command language admits assignments $id = e$ ($e \in$ Expr), sequences of instructions, a conditional $\mathsf{if}_\Phi\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2$ ($b \in$ BExpr), a loop statement $\mathsf{while}_\Phi\ b\ \mathsf{do}\ c$ ($b \in$ BExpr) and the void operation nop.

$$\frac{\delta' = \delta[id \mapsto e] \quad id \notin \beta}{\langle id = e, \delta, C, \beta\rangle \Rightarrow_\vartheta \langle \mathsf{nop}, \delta', C, \beta\rangle} \quad (A1)$$

$$\frac{e' = \delta(e) \quad \sigma^\sharp = [\![C[c]]\!]^\sharp \iota^\sharp \quad \langle e', \sigma^\sharp\rangle \rightsquigarrow^* e''}{\langle id = e, \delta, C, \beta\rangle \Rightarrow_\vartheta \langle id = e'', \delta, C, \beta\rangle} \quad (A2)$$

$$\frac{}{\langle \mathsf{nop}\ ;\ c\ , \delta, C, \beta\rangle \Rightarrow_\vartheta \langle c, \delta, C, \beta\rangle} \quad (S1)$$

$$\frac{}{\langle c\ ;\ \mathsf{nop}\ , \delta, C, \beta\rangle \Rightarrow_\vartheta \langle c, \delta, C, \beta\rangle} \quad (S2)$$

$$\frac{C' = C[[];c_2] \quad \langle c_1, \delta, C', \beta\rangle \Rightarrow^*_\vartheta \langle c_1', \delta', C', \beta'\rangle \quad C'' = C[c_1';[]] \quad \langle c_2, \delta', C'', \beta'\rangle \Rightarrow_\vartheta \langle c_2', \delta'', C'', \beta''\rangle}{\langle c_1\ ;\ c_2\ , \delta, C, \beta\rangle \Rightarrow_\vartheta \langle c_1'\ ;\ c_2', \delta'', C, \beta''\rangle} \quad (S3)$$

$$\frac{\sigma^\sharp = [\![C[\mathsf{if}_\Phi\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]]\!]^\sharp \iota^\sharp \quad [\![e]\!]^\sharp \sigma^\sharp = \mathsf{true} \quad \langle c_1, \delta, C, \beta\rangle \Rightarrow_\vartheta \langle c_1', \delta', C, \beta\rangle}{\langle \mathsf{if}_\Phi\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \delta, C, \beta\rangle \Rightarrow_\vartheta \langle \Psi(\Phi, c_1'), \Psi(\Phi, \delta'), C, \beta\rangle} \quad (C1)$$

$$\frac{\sigma^\sharp = [\![C[\mathsf{if}_\Phi\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]]\!]^\sharp \iota^\sharp \quad [\![e]\!]^\sharp \sigma^\sharp = \mathsf{false} \quad \langle c_2, \delta, C, \beta\rangle \Rightarrow_\vartheta \langle c_2', \delta', C, \beta\rangle}{\langle \mathsf{if}_\Phi\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \delta, C, \beta\rangle \Rightarrow_\vartheta \langle \Psi(\Phi, c_2'), \Psi(\Phi, \delta'), C, \beta\rangle} \quad (C2)$$

$$\frac{\begin{array}{c} Var(e) \cap Dom(\delta) = \emptyset \quad \beta' = \beta \cup Assigned(c_1) \cup Assigned(c_2) \\ \langle c_1, \delta, C, \beta'\rangle \Rightarrow_\vartheta \langle c_1', \delta_1, C, \beta_1\rangle \quad \langle c_2, \delta, C, \beta'\rangle \Rightarrow_\vartheta \langle c_2', \delta_2, C, \beta_2\rangle \quad \delta' = \delta_1 \cup \delta_2 \end{array}}{\langle \mathsf{if}_\Phi\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \delta, C, \beta\rangle \Rightarrow_\vartheta \langle \mathsf{if}_\Phi\ e\ \mathsf{then}\ c_1'\ \mathsf{else}\ c_2', \delta', C, \beta'\rangle} \quad (C3)$$

$$\frac{\begin{array}{c} V = Var(e) \quad c' = AddDefs(V, \delta) \quad \delta' = \delta_{|Dom(\delta)\setminus V} \\ \langle c'; \mathsf{if}_\Phi\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \delta', C, \beta \cup V\rangle \Rightarrow_\vartheta \langle c'', \delta', C, \beta'\rangle \end{array}}{\langle \mathsf{if}_\Phi\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \delta, C, \beta\rangle \Rightarrow_\vartheta \langle c'', \delta', C, \beta'\rangle} \quad (C4)$$

$$\frac{Var(e) \cap Dom(\delta) = \emptyset \quad C' = C[\mathsf{while}_\Phi\ e\ \mathsf{do}\ []] \quad \langle c, \delta, C', \beta\rangle \Rightarrow_\vartheta \langle c', \delta', C', \beta'\rangle}{\langle \mathsf{while}_\Phi\ e\ \mathsf{do}\ c, \delta, C, \beta\rangle \Rightarrow_\vartheta \langle \mathsf{while}_\Phi\ e\ \mathsf{do}\ c', \delta', C, \beta'\rangle} \quad (W1)$$

$$\frac{V = Var(e) \cup Var(\Phi) \quad c' = AddDefs(V, \delta) \quad \delta' = \delta_{|Dom(\delta)\setminus V} \quad \langle c'; \mathsf{while}_\Phi\ e\ \mathsf{do}\ c, \delta', C, \beta \cup V\rangle \Rightarrow_\vartheta \langle c'', \delta', C, \beta'\rangle}{\langle \mathsf{while}_\Phi\ e\ \mathsf{do}\ c, \delta, C, \beta\rangle \Rightarrow_\vartheta \langle c'', \delta', C, \beta'\rangle} \quad (W2)$$

Fig. 5 Transformation rules used to improve the accuracy of programs

We write $\equiv$ the syntactic equivalence, in other words, $x \equiv c$ means that x is syntactically the command c. In addition, we recall $\mathcal{V}$ denotes the finite set of identifiers. We assume that our programs are written in SSA form [10]. The Φ variables attached to conditional and while statements denote their sets of Φ-nodes. A Φ-node $\Phi(id, id_1, id_2)$ is understood as an assignment of the form $id = \Phi(id_1, id_2)$ where $\Phi(id_1, id_2) = id_1$ or $\Phi(id_1, id_2) = id_2$ depending on the control flow. The construction of Φ-nodes is classical and is left to the reader [3,10]. We only illustrate it by means of the Example 2.

Example 2 Initially, we give the original program in Eq. (20).

$$\begin{array}{l} x = 2; \\ \mathsf{if}\ (x > 1)\ \mathsf{then} \\ \quad x = x \times 2; \\ \mathsf{else} \\ \quad x = x \div 2; \\ z = x \end{array} \quad (20)$$

The SSA form of the program given in Eq. (20) is:

$$\begin{array}{l} x_1 = 2; \\ \mathsf{if}\ (x_1 > 1)\ \mathsf{then} \\ \quad x_2 = x_1 \times 2; \\ \mathsf{else} \\ \quad x_3 = x_1 \div 2; \\ \phi(x_4, x_2, x_3); \\ z = x_4 \end{array} \quad (21)$$

In the new program given in Eq. (21), the variables are assigned only once. Then x as been split into x_1, x_2, x_3 and x_4. The Φ-node $\Phi(x_4, x_2, x_3)$ states that x_4 is assigned to x_2 or x_3 depending on the branch taken by the control flow. □

The rewriting rules $\Rightarrow_\vartheta$ that allow us to transform programs, in order to optimize a user-defined variable ϑ, i.e., the *target variable*, are presented of Fig. 5. We denote by $\Rightarrow^*_\vartheta$ the transitive closure of $\Rightarrow_\vartheta$. In our approach, we use states of the form $\langle c, \delta, C, \beta\rangle$ where:

– c is a command, as defined in Eq. (19),
– δ is an environment $\delta\colon \mathscr{V} \to$ Expr which maps variables to expressions. Intuitively, this environment, fed by Rule $(A1)$, records the expressions assigned to variables in order to inline them later on in larger expressions thanks to Rule $(A2)$. We write $\delta[x \mapsto e]$, the environment which is equal to $\delta(x)$ for all id $\in \mathscr{V}$ such that id $\neq x$, otherwise, it returns the value e.
– $C \in$ Ctx is a single hole context [22] defined in Eq. (22). It records the program enclosing the current expression to be transformed and which is intended to fit in the hole denoted by [ ].

$$\begin{aligned}\text{Ctx} \ni C ::= {} & [\,] \mid id = e \mid C_1 ; C_2 \mid \mathsf{if}_{\Phi}\ e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \\ & \mid \mathsf{while}_{\Phi}\ e\ \mathsf{do}\ C \mid \mathsf{nop}. \end{aligned} \tag{22}$$

– let $\beta \subseteq \mathscr{V}$ be a list of assigned variables that should not be removed from the source program. Initially, $\beta = \{\vartheta\}$, i.e., the target variable ϑ may not be removed. The set β is modified by rules $(C1)$, $(C2)$, $(C4)$ and $(W2)$.

Let us now describe the rules of Fig. 5. Rule $(A1)$ allows one to discard an assignment $id = e$ by memorizing in δ the formal expression e in order to inline it later, in a larger expression. When using Rule $(A1)$, to get a semantically equivalent program, we must respect a restriction which is that the transformation is done only if Identifier id does not belong to the set β of variables which must not be removed. Note that we always have $\vartheta \in \beta$ so that the identifier id of the assignment cannot be the *target variable* when using $(A1)$.

Rule $(A2)$ offers an alternative way of processing assignments, when the conditions of Rule $(A1)$ is not fulfilled. The action of substituting the variables of e by their definitions in δ is denoted by $\delta(e)$. We also use the function $Size(e)$ which computes the size of the expression e (i.e., the number of nodes of its syntactic tree). Rule $(A2)$ transforms the expression $e' = \delta(e)$ into an expression e'' by a call $\langle e', \sigma^\sharp\rangle \rightsquigarrow^* e''$ to the tool based on APEGs and which transforms expressions, as described in Sect. 3.2. The abstract environment $\sigma^\sharp : \mathscr{V} \to \mathbb{E}^\sharp$ used for this transformation results from a static analysis using the domain $\mathbb{E}^\sharp$ also introduced in Sect. 3.1. As mentioned earlier, in Rule $(A2)$, $\iota^\sharp$ denotes the user-defined initial environment which binds the free variables of the program to intervals. For example, in Sect. 2, the variable `sl` is set to `[0.52,0.53]` in $\iota^\sharp$. The program given to the static analyzer is $C[c]$, i.e., the program obtained by inserting the command c into the context C. Accordingly to these notations, if the size of the inlined expression $e' = \delta(e)$ is less than a user-defined maximal size $S_{\max}$, then the expression e' is transformed into an expression e'' by $\langle e', \sigma^\sharp\rangle \rightsquigarrow^* e''$ which transforms the source expression into a more accurate one for the environment $\sigma^\sharp$. In our implementation, this corresponds to a call to the APEG tool [24,31]. The returned expression e'' is inserted in the new assignment $id = e''$.

Remark that by inlining expressions in variables when transforming programs, we create large formulæ. In our implementation, in order to facilitate their manipulation, we slice these formulæ at a defined level of the syntactic tree on several sub-expressions and we assign them to intermediary variables. Finally, we inject these new assignments into the main program, for example, see Listing 2.

Example 3 To explain the use of rules $(A1)$ and $(A2)$, let us consider the example of Eq. (23) in which three variables x, y and z are assigned. In this example, ϑ consists of the variable z that we aim at optimizing and $a = 0.1$, $b = 0.01$, $c = 0.001$ and $d = 0.0001$ are constants.

$$\begin{aligned}
& \langle \mathtt{x = a + b; y = c + d; z = x + y},\ \delta,\ [\,],\ \{\mathtt{z}\}\rangle \\
\underset{(A1)}{\Longrightarrow_{\vartheta}}\ & \langle \mathsf{nop}; \mathtt{y = c + d; z = x + y},\ \delta' = \delta[\mathtt{x} \mapsto \mathtt{a + b}],\ [\,],\ \{\mathtt{z}\}\rangle \\
\underset{(S1)}{\Longrightarrow_{\vartheta}}\ & \langle \mathtt{y = c + d; z = x + y},\ \delta' = \delta[\mathtt{x} \mapsto \mathtt{a + b}],\ [\,],\ \{\mathtt{z}\}\rangle \\
\underset{(A1)}{\Longrightarrow_{\vartheta}}\ & \langle \mathsf{nop}; \mathtt{z = x + y},\ \delta'' = \delta'[\mathtt{y} \mapsto \mathtt{c + d}],\ [\,],\ \{\mathtt{z}\}\rangle \\
\underset{(S1)}{\Longrightarrow_{\vartheta}}\ & \langle \mathtt{z = x + y},\ \delta'' = \delta'[\mathtt{y} \mapsto \mathtt{c + d}],\ [\,],\ \{\mathtt{z}\}\rangle \\
\underset{(A2)}{\Longrightarrow_{\vartheta}}\ & \langle \mathtt{z = ((d + c) + b) + a}, \delta'',\ [\,], \{\mathtt{z}\}\rangle
\end{aligned} \tag{23}$$

In Eq. (23), initially, the environment δ is empty. If we apply the first rule $(A1)$, we may remove the variable x and memorize it in δ. So, the line corresponding to the variable discarded is replaced by nop and the new environment is $\delta = [x \mapsto a + b]$. We then repeat the same process by using $(A1)$ on the variable y and Rule $(S1)$ which discards the nopstatement. For the last step, we may not apply $(A1)$ to z because the condition is not satisfied $(z = \vartheta)$. Then we use $(A2)$, we substitute x and y by their value in δ and we transform the expression. □

Rules $(S1)$ to $(S3)$ deal with sequences. Rules $(S1)$ and $(S2)$ are special cases enabling the system to discard the nop statements while the general rule for sequences is $(S3)$. The first command c_1 is transformed into c_1' in the current environment δ, C, ϑ and β and a new context C' is built which inserts c_1' inside C. Then c_2 is transformed into c_2' using the context $C[c_1'; []]$, the formal environments δ' and the list β' resulting from the transformation of c_1. Finally, we return the state $\langle c_1' ;\ c_2', \delta'', C, \beta''\rangle$.

Example 4 Back to the example of Eq. (23), the rules $(S1)$ and $(S3)$ are needed to achieve the transformation: first, $(S3)$ is used with $c_1 = \{\mathtt{x = a + b}\}$ and $c_2 = \{\mathtt{y = c + d\ ;\ z = x + y}\}$. For c_1, $(A1)$ is called from $(S3)$ and, concerning c_2, $(S3)$ is recursively called, as well as $(A1)$ and $(S1)$. □

Rules $(C1)$ to $(C4)$ concern conditionals. The first two rules correspond to a partial evaluation of the program [25], when the test evaluates to true or false in the environment $\sigma^\sharp$ which is computed by static analysis

$$\sigma^\sharp = [\![C[\mathsf{if}_\Phi\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]]\!]^\sharp \iota^\sharp.$$

The conditional, in rules $(C1)$ and $(C2)$, is replaced by either the Branch c_1 or c_2. Since the conditional is removed, we have to take care of the Φ-nodes. We use the function $\Psi(\Phi, c)$ which replaces in the command c any assignment $x = e$ by $y = e$ if $\Phi(y, u, v) \in \Phi$ with $u = x$ or $v = x$. Similarly, $\Psi(\Phi, \delta)$ replaces, in the formal environment δ, $x \mapsto e$ by $y \mapsto e$ if $\Phi(y, u, v) \in \Phi$ with $u = x$ or $v = x$. Doing so, we propagate the effect of the Φ-nodes when a conditional is removed.

Example 5 Let us consider the program p, in SSA form.

$$p \quad \equiv \quad \begin{array}{l} x_1 = 0; \\ \mathsf{if}_{\Phi(x_4,x_2,x_3)}\ cond\ \mathsf{then} \\ \quad x_2 = a + b \\ \mathsf{else} \\ \quad x_3 = c + d; \\ \vartheta = x_4 \end{array} \tag{24}$$

In the rest of this example, we assume that $c_1 \equiv a + b$ and $c_2 \equiv c + d$. Depending on the value of condition, we transform this program into

$$\begin{cases} x_1 = 0; \\ \vartheta = a + b & \text{if } cond, \\ \vartheta = c + d & \text{if } \neg cond. \end{cases} \tag{25}$$

For example, if the condition is true, the steps followed by the transformation are

$$\begin{array}{rcl} \langle p, \delta, C, \beta \rangle & \underset{(C1)}{\Longrightarrow_\vartheta} & \langle \Psi(\Phi, c_1), \Psi(\Phi, \delta), C, \beta \rangle \\ & \equiv & \langle \Psi(\Phi, x_2 = a + b; \vartheta = x_2), \Psi(\Phi, \delta), C, \beta \rangle \\ & \equiv & \langle x_4 = a + b; \vartheta = x_4), \delta, C, \beta \rangle \\ & \underset{(A1)}{\Longrightarrow_\vartheta} & \langle \mathsf{nop}; \vartheta = x_4, \delta'[x_4 \mapsto a + b], C, \beta \rangle \\ & \underset{(S1)}{\Longrightarrow_\vartheta} & \langle \vartheta = x_4, \delta'[x_4 \mapsto a + b], C, \beta \rangle \\ & \underset{(A2)}{\Longrightarrow_\vartheta} & \langle \vartheta = a + b, \delta', C, \beta \rangle \end{array}$$

□

Rule $(C3)$ is the general rule for conditionals. The then and else branches are transformed, assuming that the variables of the condition do not appear in the domain of δ. Here, $Assigned(c)$ denotes the set of identifiers assigned in the command c, Dom(δ) denotes the list of variables memorized in δ and Var(e) denotes the set of variables of the expression e.

The variables assigned in the branches have to be added to β and the environment δ' resulting from the transformation joins the environments of both branches (thanks to the SSA form, the variables assigned in both branches are distinct). The function $Var(e)$ returns the set of variables occurring in the expression e while Dom(δ) denotes the domain of definition of δ. Finally, Rule $(C4)$ is used when the conditions for Rule $(C3)$ do not hold. In this case, Var(e)$\cap$Dom(δ)$/$ $= \emptyset$ and we need to reinsert the common variables into the source code. Let $Var(e)$ be the list of variables occurring in the expression e. Firstly, a new command c' corresponding to sequences of assignments of the form $id = \delta(id)$ is built from $AddDefs(Var(e), \delta)$ such that for any set of variables V

$$AddDefs(V, \delta) \equiv \mathrm{id}_1 = \delta(\mathrm{id}_1); \ldots; \mathrm{id}_n = \delta(\mathrm{id}_n) \quad \text{with} \quad V = \{\mathrm{id}_1, \ldots, \mathrm{id}_n\}. \tag{26}$$

Secondly, the variables of $Var(e)$ are removed from the domain of δ, yielding δ'. The resulting command is the command c'' obtained by transforming c'; $\mathsf{if}_\Phi\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2$ with δ' and $\beta \cup Var(e)$.

Example 6 Let us take another example to explain the rules $(C3)$ and $(C4)$. Initially, we have $\langle q, \delta, C, \beta = \{\vartheta\} \rangle$ and

$$q \quad \equiv \quad \begin{array}{l} x_1 = a; \\ \mathsf{if}_{\Phi(y_3,y_1,y_2)}\ (x_1 > 1)\ \mathsf{then} \\ \quad y_1 = x_1 + 2; \\ \mathsf{else} \\ \quad y_2 = x_1 - 1; \\ \vartheta = y_3 \end{array} \tag{27}$$

By rule $(A1)$, x_1 is stored in δ. Then, we transform recursively the new program

$$q' \quad \equiv \quad \begin{array}{l} \mathsf{if}_{\Phi(y_3,y_1,y_2)}\ (x_1 > 1)\ \mathsf{then} \\ \quad y_1 = x_1 + 2; \\ \mathsf{else} \\ \quad y_2 = x_1 - 1; \\ \vartheta = y_3 \end{array} \tag{28}$$

We write $\langle q, \delta, C, \beta = \{\vartheta\} \rangle \underset{(A1)}{\Longrightarrow_\vartheta} \langle q', \delta'[x_1 \mapsto 0], C, \beta = \{\vartheta\} \rangle$.

This program is semantically incorrect since the test is undefined. However, $Var(e) \cap Dom(\delta) \neq \emptyset$ and we cannot apply Rule $(C3)$. Instead, Rule $(C4)$ is used to re-inject the statements $x_1 = 0$ in the program and to add x_1 to the blacklist β in order to avoid an infinite loop in the transformation. The new blacklist is $\beta' = \{\vartheta, x_1\}$. □

The last two rules $(W1)$ and $(W2)$ are for the while statements. They follow the same spirit than the rules $(C3)$ and $(C4)$ for the conditional statement. Rule $(W1)$ makes it possible to transform the body c of the loop assuming that the variables of the condition e have not been stored in δ. In this case, c is optimized in the context $C[\mathsf{while}_{\Phi}\ e\ \mathsf{do}\ []]$ where C is the context of the loop itself. Rule $(W2)$ first builds the list $V = Var(e) \cup Var(\Phi)$ where $Var(\Phi)$ is the list of variables read and written in the Φ nodes of the loop. The set V is used to achieve two tasks: firstly, it is used to build a new command c' corresponding to the sequence of assignments $id = \delta(id)$, for all $id \in V$ (as for Rule $(C4)$). Secondly, the variables of V are removed from the domain of δ and added to β. The resulting command is the command c'' obtained by transforming c'; $\mathsf{while}_{\Phi}\ e\ \mathsf{do}\ c$ with δ' and $\beta \cup V$.

We end this section with complexity considerations. At each step of the transformation of a program p, only one rule of Fig. 5 may be selected. Consequently, the transformation would be linear in the size n, i.e., the number of statements, of p if we would not re-inject assignments. However, a given assignment cannot be removed twice, so the transformation is quadratic. Finally, the entire transformation of a program p is repeated until nothing changes, that is at most n times. Hence, the global complexity for a program transformation of size n is $\mathcal{O}(n^3)$.

Note that, in this article, we optimize only one variable at once. The simplest way to optimize a set of variables X at the same time is by code duplication, by optimizing the original program independently for each $id \in X$ and by merging the resulting programs. Obviously, this makes the code grow too much in practice. In future work, we aim at defining a better method to improve the accuracy of several variables simultaneously without code duplication. A first difficulty is to define a (partial) order relation to compare the accuracy of programs on several variables at once. This relation would be used instead of the order $\sqsubseteq$ introduced in Sect. 5 (Definition 1) and which states that a program is better than another one if the accuracy of the reference variable has been improved.

5 Soundness of transformations

In order to ensure the soundness of our transformation, we introduce a theorem which compares two programs and states that the more accurate one may be used in place of the less accurate one.

5.1 Operational semantics

In this section, we introduce an usual SOS operational semantics for our `while` language defined in Sect. 4. This semantics is standard, but we need to make it explicit in order to achieve the proof of correctness of our transformation in Sect. 5.2.

Let $\mathscr{V}$ be a finite set of variables and let $\sigma \in \mathsf{Mem}$ be an environment that map variables to values

$$\sigma\ : \mathscr{V} \to \mathbb{E} \tag{29}$$

with $\mathbb{E}$ the concrete domain of values introduced in Sect. 3.1. In Fig. 6, we define

$$\to_e :\ \mathrm{Expr} \to \mathbb{E} \tag{30}$$

and

$$\to_b :\ \mathrm{BExpr} \to \{\mathsf{true}, \mathsf{false}\} \tag{31}$$

the transition functions for the evaluation of arithmetic and boolean expressions. Note that in Fig. 6 only the most relevant, for the proof of correctness, definitions of the operational semantics are given as they follow a standard definition as found in [41].

The operational semantics uses states which are defined by a pair of command and memory. We assume that

$$\mathrm{State} = \{\langle c, \sigma\rangle : c \in \mathrm{Cmd}, \sigma \in \mathrm{Mem}\}. \tag{32}$$

The operational semantics maps States to States:

$$\begin{array}{l} \to :\ \mathrm{State} \to \mathrm{State} \\ \qquad \langle c, \sigma\rangle \mapsto \langle c', \sigma'\rangle. \end{array} \tag{33}$$

The standard semantics of programs is given in Fig. 6 (rules $(R8)$ to $(R15)$). The rules for assignments rely on the semantics $\to_e$ of expressions also given in Fig. 6 (rules $(R1)$ to $(R4)$). Recall that, as explained in Sect. 3.1, our technique is independent of the rounding mode used in the floating-point arithmetics and, in Fig. 6, we omit to mention it (Figs. 7, 8, 9).

For a sequence of commands (rules $(R10)$ and $(R11)$), we execute the first command in the initial environment, and then we execute the second in the resulting environment. The next kind of rules is for conditionals and uses the semantics $\to_b$ also given in Fig. 6 (rules $(R5)$ to $(R7)$). If the condition is evaluated to true (rule $(R12)$), we take the semantics of the then branch, otherwise, we take the semantics of the else branch (rule $(R13)$). The rules for while loops execute the body c if the condition is true and then run again the loop. They return the environment unchanged if the condition is false (Rule $(R15)$).

5.2 Proof of correctness

In an effort to assert the correctness of our transformation for numerical accuracy, we now prove that our approach

Fig. 6 Small-step operational semantics of programs

$$\frac{\sigma(x) = cst}{\langle x, \sigma\rangle \rightarrow_e \langle cst, \sigma\rangle} \quad (R1)$$

$$\frac{\langle e_1, \sigma\rangle \rightarrow_e \langle e_1', \sigma\rangle}{\langle e_1 \odot e_2, \sigma\rangle \rightarrow_e \langle e_1' \odot e_2, \sigma\rangle} \quad with \quad \odot \in \{+, -, \times, \div\} \quad (R2)$$

$$\frac{\langle e_2, \sigma\rangle \rightarrow_e \langle e_2', \sigma\rangle}{\langle cst_1 \odot e_2, \sigma\rangle \rightarrow_e \langle cst_1 \odot e_2', \sigma\rangle} \quad with \quad \odot \in \{+, -, \times, \div\} \quad (R3)$$

$$\frac{cst = cst_1 \odot cst_2}{\langle cst_1 \odot cst_2, \sigma\rangle \rightarrow_e \langle cst, \sigma\rangle} \quad with \quad \odot \in \{+, -, \times, \div\} \quad (R4)$$

$$\frac{\langle e_1, \sigma\rangle \rightarrow_e \langle e_1', \sigma\rangle}{\langle e_1 \mathscr{R} e_2, \sigma\rangle \rightarrow_b \langle e_1' \mathscr{R} e_2, \sigma\rangle} \quad with \quad \mathscr{R} \in \{=, <, >, \leq, \geq\} \quad (R5)$$

$$\frac{\langle e_2, \sigma\rangle \rightarrow_b \langle e_2', \sigma\rangle}{\langle cst_1 \mathscr{R} e_2, \sigma\rangle \rightarrow_b \langle cst_1 \mathscr{R} e_2', \sigma\rangle} \quad with \quad \mathscr{R} \in \{=, <, >, \leq, \geq\} \quad (R6)$$

$$\frac{cst = cst_1 \mathscr{R} cst_2}{\langle cst_1 \mathscr{R} cst_2, \sigma\rangle \rightarrow_b \langle cst, \sigma\rangle} \quad with \quad \mathscr{R} \in \{=, <, >, \leq, \geq\} \quad (R7)$$

$$\frac{\langle e, \sigma\rangle \rightarrow \langle e', \sigma\rangle}{\langle id = e, \sigma\rangle \rightarrow \langle id = e', \sigma\rangle} \quad (R8) \qquad \frac{\sigma' = \sigma[id \mapsto cst]}{\langle id = cst, \sigma\rangle \rightarrow \langle nop, \sigma'\rangle} \quad (R9)$$

$$\frac{\langle c_1, \sigma\rangle \rightarrow \langle c_1', \sigma'\rangle}{\langle c_1; c_2, \sigma\rangle \rightarrow \langle c_1'; c_2, \sigma'\rangle} \quad (R10) \qquad \frac{}{\langle \mathsf{nop}; c, \sigma\rangle \rightarrow \langle c, \sigma\rangle} \quad (R11)$$

$$\frac{}{\langle \mathsf{if}_\Phi \text{ true then } c_1 \text{ else } c_2, \sigma\rangle \rightarrow \langle c_1, \sigma\rangle} \quad (R12) \qquad \frac{}{\langle \mathsf{if}_\Phi \text{ false then } c_1 \text{ else } c_2, \sigma\rangle \rightarrow \langle c_2, \sigma\rangle} \quad (R13)$$

$$\frac{\langle b, \sigma\rangle \rightarrow_b \langle b', \sigma\rangle}{\langle \mathsf{if}_\Phi \ b \text{ then } c_1 \text{ else } c_2, \sigma\rangle \rightarrow \langle \mathsf{if}_\Phi \ b' \text{ then } c_1 \text{ else } c_2, \sigma\rangle} \quad (R14)$$

$$\frac{}{\langle \mathsf{while}_\Phi \ b \text{ do } c, \sigma\rangle \rightarrow \langle \mathsf{if}_\Phi \ b \text{ then } c; \text{ while}_\Phi \ b \text{ do } c \text{ else nop}, \sigma\rangle} \quad (R15)$$

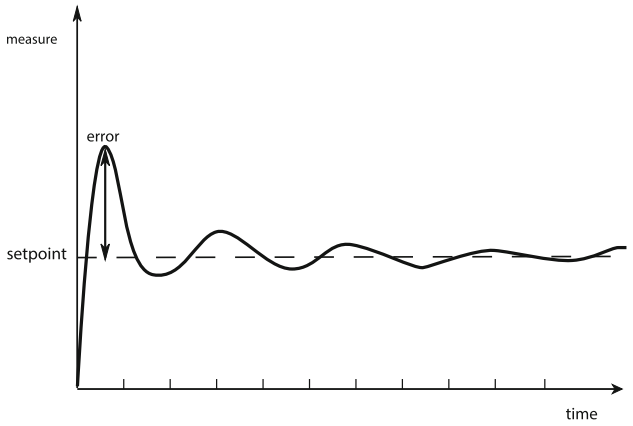

Fig. 7 Output of a PID controller

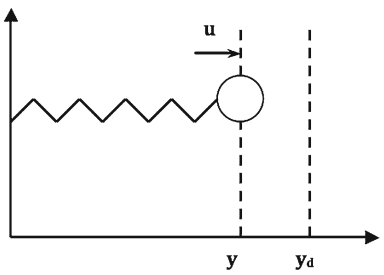

Fig. 8 The lead–lag system

really generates a more accurate and correct program among the many equivalent programs. This proof relies on the operational semantics of both arithmetic expressions and commands. It is crucial because the transformed programs are really different from the original ones. Indeed, if we operate on critical embedded systems, one necessarily needs to be sure that the transformed programs really behave like the original ones.

Key to our approach is the use of Theorem 1 introduced later on in this section and which compares two programs and states that the most accurate one may be used in place of the less accurate one. This comparison needs to specify a variable of reference ϑ defined by the user. More precisely, a transformed program p_t is more accurate than an original program p_o if and only if the two conditions given hereafter are verified:

- The first condition consists in ensuring that the *target variable* ϑ of both programs corresponds to the same mathematical expression.
- The second condition requires that the *target variable* ϑ is more accurate in the transformed program p_t than in the original program p_o.

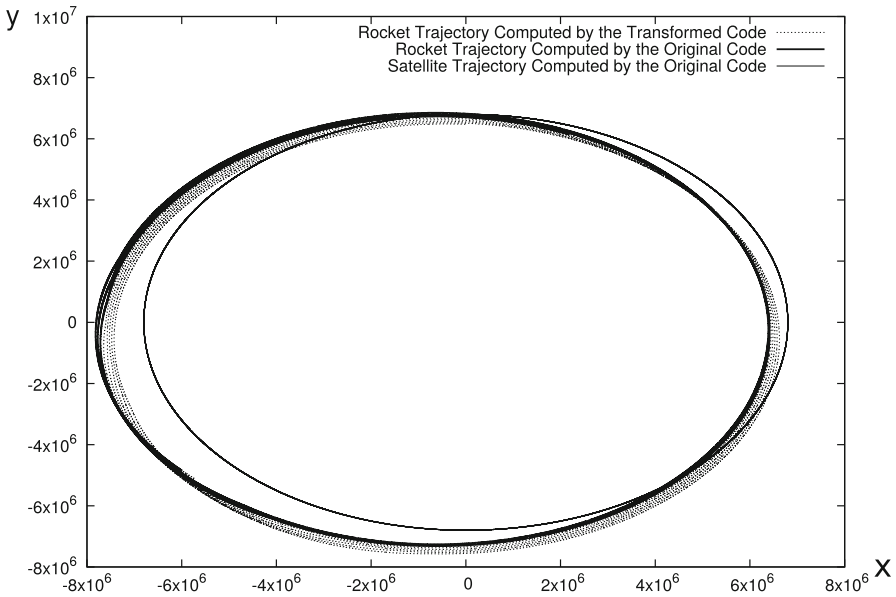

Fig. 9 Difference between the initial and the transformed trajectories of the rocket

The main difficulty of the proof comes from the fact that the transformation discards some assignments which have to be re-injected later on to build larger expressions. This makes our transformation unusual and forms the originality of the proof presented below.

In the rest of this section, we are going to use the rules introduced in Sect. 3, $\langle c, \delta, C, \beta\rangle \Rightarrow^*_\vartheta \langle c', \delta', C, \beta'\rangle$.

We denote by $\Rightarrow^*_\vartheta$ the reflexive transitive closure of $\Rightarrow_\vartheta$. In order to define the correctness of the transformation, we introduce two order relations which make it possible to compare commands based on their accuracy.

First, we use an order relation $\sqsubseteq\subseteq \mathbb{E}\times\mathbb{E}$ which states that a value (x, μ) is more accurate than a value (x', μ') if they correspond to the same real value and if the error μ is less than μ'.

Definition 1 *(comparison of expressions)* Let us consider $v_1 = (x_1, \mu_1) \in \mathbb{E}$ and $v_2 = (x_2, \mu_2) \in \mathbb{E}$. We say that v_1 is more accurate than v_2, denoted by $v_1 \sqsubseteq v_2$ iff $x_1 + \mu_1 = x_2 + \mu_2$ and $|\mu_1| \leq |\mu_2|$. □

Second, Definition 2 says that both commands compute the same value of ϑ in any environment in the exact arithmetic, that the transformed command is more accurate and that if a variable is not defined in σ_t then the corresponding formal expression has been stored in δ_t.

Definition 2 *(comparison of commands)* Let c_o and c_t be two commands, let δ_o and δ_t be two formal environments. Finally, let ϑ be the reference variable. We say that

$$\langle c_t, \delta_t\rangle \prec_\vartheta \langle c_o, \delta_o\rangle$$

if and only if for all $\sigma \in$ Mem,

$$\exists \sigma_o \in \text{Mem}, \langle c_o, \sigma\rangle \to^* \langle \text{nop}, \sigma_o\rangle,$$
$$\exists \sigma_t \in \text{Mem}, \langle c_t, \sigma\rangle \to^* \langle \text{nop}, \sigma_t\rangle,$$

such that $\sigma_t(\vartheta) \sqsubseteq \sigma_o(\vartheta)$ and such that for all $\text{id} \in \text{Dom}(\sigma_o) \setminus \text{Dom}(\sigma_t)$, $\delta_t(\text{id}) = e$ and $\langle e, \sigma\rangle \to^*_e \sigma_o(\text{id})$. □

Let $\delta\colon \mathscr{V} \to$ Expr be a formal environment. In the following, $\delta(e)$ is the expression obtained by substituting in the expression e any identifier id of δ by its value $\delta(\text{id})$.

First of all, we introduce a lemma which states that we may substitute an expression e to a variable id inside a larger expression as long as e evaluates to $\sigma(id)$.

Lemma 1 (sub-expression substitution) *Let e be an expression containing the identifier id. Assume a formal environment δ such that $\delta(id) = e'$ with e' another expression. Moreover, assume that $\forall\sigma \in Mem$, $\exists cst \in \mathbb{E}$,*

$$\langle e', \sigma\rangle \to^* cst.$$

If, $\forall\sigma \in Mem$,

$$\exists r \in \mathbb{E}, \quad \langle e, \sigma[id \mapsto cst]\rangle \to^* r,$$
$$\exists r' \in \mathbb{E}, \quad \langle \delta(e), \sigma\rangle \to^* r',$$

then $r = r'$. □

Proof Given Lemma 1, the proof is done by induction on the structure of the expressions introduced in Eq. (17). We have to consider the following cases:

– If the expression is a constant, in other words, $e \equiv cst$, then we know that $\delta(cst) = cst$. Obviously, there is no change. In addition, we have $\langle cst, \sigma\rangle \to cst$ and

$\langle \delta(e), \sigma \rangle \to cst$. Consequently, when we deal with constants the lemma is always correct.

- If the expression is a variable then $e \equiv id$ and we know by hypothesis that $\delta(id) = e'$. In this case, we have that $\langle id, \sigma \rangle \to v$ and, by hypothesis, we know that if $\langle \delta(id), \sigma \rangle \to^* v'$ then $v = v'$. So the lemma is also correct in this case.
- Finally, if we have an expression of the form $e \equiv e_1 \odot e_2$ where $\odot \in \{+, -, \times, \div\}$ then we apply the induction hypothesis separately to both expressions e_1 and e_2. So, we have $\delta(e_1) = e_1'$, $\langle e_1, \sigma[id \mapsto v_1] \rangle \to v_1'$ and $\langle \delta(e_1), \sigma \rangle \to v_1''$. The same process is applied to e_2 and we know by induction hypothesis that $v_1' = v_1''$ and $v_2' = v_2''$. We conclude that $v_1' \odot v_1'' = v_2' \odot v_2''$. The lemma is correct for arithmetic operations.

□

We introduce now a second lemma concerning the soundness of the transformation of arithmetic expressions. As in Sect. 4, we assume that a function $\rightsquigarrow$ transforming arithmetic expressions is given; see [24]. This function transforms an expression e_o into a more accurate expression e_t in the environment σ. Again, $\rightsquigarrow^*$ denotes the reflexive transitive closure of $\rightsquigarrow$.

Lemma 2 (soundness of expressions transformation) *Let e_o be an original arithmetic expression, e_t be the transformed expression. For all $\sigma \in Mem$, assume that $\langle e_o, \sigma \rangle \rightsquigarrow^* e_t$, i.e., e_o is transformed into a mathematically equivalent expression e_t, such that*

$$\exists v_o \in \mathbb{E}, \quad \langle e_o, \sigma \rangle \to_e v_o,$$
$$\exists v_t \in \mathbb{E}, \quad \langle e_t, \sigma \rangle \to_e v_t,$$

then $v_t \sqsubseteq v_o$. □

Proof Such a transformation is proved correct in [24].

The following theorem relates the original commands and the transformed commands by using the relation $\prec_\vartheta$ introduced in Definition 2.

Theorem 1 (soundness of command transformation) *Let c_o be the original code, c_t be the transformed code, δ_o be the initial environment of the transformation, δ_t be the final environment of the transformation,*

then we have

$$\big(\langle c_o, \delta_o, C, \beta \rangle \Rightarrow_\vartheta \langle c_t, \delta_t, C, \beta \rangle\big) \Longrightarrow \big(\langle c_t, \delta_t, C, \beta \rangle \prec_\vartheta \langle c_o, \delta_o, C, \beta \rangle\big).$$

□

Proof The proof is by structural induction on commands. Hence, we consider each kind of expression and each rule of transformation of programs presented in Fig. 6 which applies to the current kind of expression. □

Assignments The first case is when the command c is an assignment, i.e., $c \equiv \mathrm{id} = e$. In this case, we have two transformation rules, $(A1)$ and $(A2)$. Rule $(A1)$ consists in discarding an assignment when some conditions are satisfied while Rule $(A2)$ is used to substitute expressions within the assignment using the information already stored in the environment δ.

- Let us start with $(A1)$ which produces

  $$\langle id = e, \delta, C, \beta \rangle \Rightarrow_\vartheta \langle \mathsf{nop}, \delta_t, C, \beta \rangle.$$

  We consider two cases:

  1. $\mathrm{id} \equiv \vartheta$ is impossible because the variable id is in the black list β. In Rule $(A1)$, all the variables of the black list may not be discarded from the source code.
  2. $\mathrm{id} \not\equiv \vartheta$ then the new program is nop and the assignment is memorized in δ_t. That is to say, if (i) $\mathrm{id} \not\equiv \vartheta$ and (ii) $\mathrm{id} \notin \beta$ then $c_t \equiv \mathsf{nop}$ and $\delta_t = \delta[id \mapsto e]$. In one hand, we have by composition of Rule (R9) and (R10) of the operational semantics, for all $\sigma \in \mathrm{Mem}$

     $$\exists \sigma_o \in \mathrm{Mem}, \exists v \in \mathbb{E} \quad \langle \mathrm{id} = e, \sigma \rangle \to^* \langle \mathsf{nop}, \sigma_o \rangle \quad \text{with} \quad \sigma_o = \sigma[\mathrm{id} \mapsto v]$$

     On the other hand, by discarding the previous assignment $\mathrm{id} = e$, we have $c_t \equiv \mathsf{nop}$. The execution ends with $\langle \mathsf{nop}, \sigma_t \rangle$ where $\sigma = \sigma_t$. Then, since $\vartheta \neq \mathrm{id}$, we have that $\sigma_o(\vartheta) = \sigma_t(\vartheta)$. In addition, we have $\langle \mathrm{id}, \sigma \rangle \to v = \sigma_o(\mathrm{id})$. This proves that the theorem holds in this case, i.e., $\langle c_o, \delta_o \rangle \prec_\vartheta \langle c_t, \delta_t \rangle$.

- The second part of the proof in the case of assignments is dedicated to demonstrate $(A2)$. First, if

  $$\langle \mathrm{id} = e, \delta, C, \beta \rangle \Rightarrow_\vartheta \langle \mathrm{id} = e'', \delta, C, \beta \rangle,$$

  then the initial program c_o is the assignment $\mathrm{id} = e$ and the transformed program c_t is $\mathrm{id} = e''$, such that

  $$e' = \delta(e), \quad \langle e', \sigma \rangle \rightsquigarrow^* e''.$$

  In other words, the new expression e'' is obtained by inlining in the expression e the variables stored in the environment δ, yielding an expression e'. Next, e'' is obtained from e' using the transformation of the expressions $\rightsquigarrow^*$.

Let us assume that $\forall\sigma \in \text{Mem}$, we have

$$\exists v \in \mathbb{E}, \quad \langle \text{id} = e, \sigma\rangle \rightarrow^* \langle \mathsf{nop}, \sigma[\text{id} \mapsto v]\rangle \quad \text{with} \quad \langle e, \sigma\rangle \rightarrow_e^* v$$

For the transformed program, assuming that $\forall\sigma \in \text{Mem}$, we have

$$\exists v' \in \mathbb{E}, \quad \langle \text{id} = e'', \sigma\rangle \rightarrow^* \langle \mathsf{nop}, \sigma[id \mapsto v']\rangle \quad \text{with} \quad \langle e'', \sigma\rangle \rightarrow_e^* v'$$

According to Lemma 1, we kown that $\forall\sigma \in \text{Mem}$

$$\exists v \in \mathbb{E}, \quad \langle e, \sigma'\rangle \rightarrow_e^* v,$$
$$\exists v' \in \mathbb{E}, \quad \langle e', \sigma\rangle \rightarrow_e^* v',$$

such that $v = v'$, with $\sigma' = \sigma[\text{id} \mapsto v]$ for all $\text{id} \in \text{Dom}(\delta)$ and $\langle \delta(id), \sigma\rangle \rightarrow_e^* v$.
According to Lemma 2, we know that if $\forall\sigma \in \text{Mem}$

$$\langle e', \sigma\rangle \rightsquigarrow^* e''$$

we have

$$\exists v' \in \mathbb{E}, \quad \langle e', \sigma\rangle \rightarrow_e^* v',$$
$$\exists v'' \in \mathbb{E}, \quad \langle e'', \sigma\rangle \rightarrow_e^* v'',$$

and

$$v'' \sqsubseteq v' = v.$$

In addition, we know that $\delta_o = \delta_t$. This demonstrates that $\langle c_t, \delta\rangle \prec_\vartheta \langle c_o, \delta\rangle$.

Sequences For a sequence of commands, if one member of sequence of commands is nop, we have the following situations.

– According to Rule $(S1)$, if $c \equiv \mathsf{nop}; c_2$, then we have

$$\big(\langle \mathsf{nop}; c_2, \delta, C, \beta\rangle \Rightarrow_\vartheta \langle c_2, \delta', C, \beta\rangle\big)$$

and $\delta = \delta'$. So, for all $\sigma \in Mem$, we have

$$\langle \mathsf{nop}; c_2, \sigma\rangle \rightarrow^* \langle \mathsf{nop}, \sigma'\rangle,$$
$$\langle c_2, \sigma\rangle \rightarrow^* \langle \mathsf{nop}, \sigma''\rangle,$$

and $\sigma' = \sigma''$. Consequently,

$$\langle c_2, \delta, C, \beta\rangle \prec_\vartheta \langle \mathsf{nop}; c_2, \delta', C, \beta\rangle.$$

– The case for Rule $(S2)$ is similar to the former one.

– For the general case $c \equiv c_1; c_2$, we know by induction hypothesis that:

$$\big(\langle c_1, \delta, C\big[[\,]; c_2\big], \beta\rangle \Rightarrow_\vartheta \langle c_1', \delta', C\big[[\,]; c_2\big], \beta\rangle\big) \Longrightarrow \big(\langle c_1', \delta', C\big[[\,]; c_2\big], \beta\rangle \prec_\vartheta \langle c_1, \delta, C\big[[\,]; c_2\big], \beta\rangle\big),$$
$$\big(\langle c_2, \delta, C\big[c_1'; [\,]\big], \beta\rangle \Rightarrow_\vartheta \langle c_2', \delta', C\big[c_1'; [\,]\big], \beta\rangle\big) \Longrightarrow \big(\langle c_2', \delta', C\big[c_1'; [\,]\big], \beta\rangle \prec_\vartheta \langle c_2, \delta, C\big[c_1'; [\,]\big], \beta\rangle\big).$$

We know that $\forall\sigma \in Mem$, we have $\langle c_1, \sigma\rangle \rightarrow \langle c_1', \sigma\rangle$ and $\langle c_2, \sigma\rangle \rightarrow \langle c_2', \sigma\rangle$. According to Rule $(S3)$, we have

$$\langle c_1; c_2, \delta, C, \beta\rangle \Rightarrow_\vartheta \langle c_1'; c_2', \delta'', C, \beta\rangle \; with \; \delta = \delta''.$$

Then for all $\sigma \in Mem$, we have that

$$\langle c_1; c_2, \sigma\rangle \rightarrow^* \sigma_2, \;\; \langle c_1'; c_2', \sigma\rangle \rightarrow^* \sigma_2' \text{ and } \sigma_2'(\vartheta) \sqsubseteq \sigma_2(\vartheta).$$

Consequently, $\langle c_1'; c_2', \delta_2', C, \beta\rangle \prec_\vartheta \langle c_1; c_2, \delta_1, C, \beta\rangle$.

Conditionals The next case concerns the conditional statement $c \equiv \mathsf{if}_\Phi\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2$. We have to demonstrate the correctness of the transformation for the four cases corresponding to rules $(C1)$ to $(C4)$ introduced in Fig. 5. If the condition is statically known then:

– If the condition e is always evaluated to true, then by induction hypothesis we write

$$\langle c_1, \delta, C, \beta\rangle \Rightarrow_\vartheta \langle c_1', \delta', C, \beta\rangle,$$

and, for all $\sigma \in Mem$ we have

$$\langle c_1, \sigma\rangle \rightarrow^* \langle \mathsf{nop}, \sigma_1\rangle,$$
$$\langle c_1', \sigma\rangle \rightarrow^* \langle \mathsf{nop}, \sigma_1'\rangle,$$

and $\sigma_1'(\vartheta) \sqsubseteq \sigma_1(\vartheta)$.
By using the Rule $(C1)$, we have that

$$\langle \mathsf{if}_\Phi\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \delta, C, \beta\rangle \Rightarrow_\vartheta \langle c_1', \delta', C, \beta\rangle$$

and $\delta = \delta'$. Consequently, we deduce that for all σ, that

$$\langle \mathsf{if}_\Phi\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \sigma\rangle \rightarrow^* \langle \mathsf{nop}, \sigma_1\rangle,$$
$$\langle \mathsf{if}_\Phi\ e\ \mathsf{then}\ c_1'\ \mathsf{else}\ c_2', \sigma\rangle \rightarrow^* \langle \mathsf{nop}, \sigma_1'\rangle,$$

and $\sigma_1'(\vartheta) \sqsubseteq \sigma_1(\vartheta)$. Thus it results that:

$$\langle c_1', \delta', C, \beta\rangle \prec_\vartheta \langle \mathsf{if}_\Phi\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \delta, C, \beta\rangle.$$

– If the condition e is always false, then we execute only the else branch. This case is similar to the previous one.

Otherwise:

– if Var(e) ∩ Dom(δ) ≠ ∅ then we use (C3) and by induction hypothesis, we have:

$$\langle c_1, \delta, C, \beta\rangle \Rightarrow_\vartheta \langle c_1', \delta', C, \beta\rangle,$$
$$\langle c_2, \delta, C, \beta\rangle \Rightarrow_\vartheta \langle c_2', \delta', C, \beta\rangle,$$

and, for all $\sigma \in Mem$ we have

$$\langle c_1, \sigma\rangle \to^* \langle \mathsf{nop}, \sigma_1\rangle, \langle c_1', \sigma\rangle \to^* \langle \mathsf{nop}, \sigma_1'\rangle,$$
$$\langle c_2, \sigma\rangle \to^* \langle \mathsf{nop}, \sigma_2\rangle, \langle c_2', \sigma\rangle \to^* \langle \mathsf{nop}, \sigma_2'\rangle,$$

and $\sigma_1'(\vartheta) \sqsubseteq \sigma_1(\vartheta)$ and $\sigma_2'(\vartheta) \sqsubseteq \sigma_2(\vartheta)$. In the program transformation, we transform c_1 and c_2 as indicated above and, at the end, $\delta' = \delta_1' \cup \delta_2'$. In the operational semantics, we execute either $\langle c_1', \sigma\rangle \to^* \langle \mathsf{nop}, \sigma_1'\rangle$ instead of $\langle c_1, \sigma\rangle \to^* \langle \mathsf{nop}, \sigma_1\rangle$ or $\langle c_2', \sigma\rangle \to^* \langle \mathsf{nop}, \sigma_2'\rangle$ instead of $\langle c_2, \sigma\rangle \to^* \langle \mathsf{nop}, \sigma_2\rangle$. In any case,

$$\sigma_1'(\vartheta) \sqsubseteq \sigma_1(\vartheta) \text{ or } \sigma_2'(\vartheta) \sqsubseteq \sigma_2(\vartheta).$$

Consequently, since $\delta' = \delta_1' \cup \delta_2'$,

$$\langle \mathsf{if}_\Phi\ e\ \mathsf{then}\ c_1'\ \mathsf{else}\ c_2', \delta', C, \beta\rangle \prec_\vartheta \langle \mathsf{if}_\Phi\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \delta, C, \beta\rangle.$$

– In the last case, let $c \equiv \mathsf{if}_\Phi\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2$. If

$$\langle c', \delta', C, \beta\rangle \prec_\vartheta \langle c, \delta, C, \beta\rangle,$$

and if we add the same assignments within a command c_a at the beginning of c and c' then necessarily, $\langle c_a; c', \delta'\rangle \prec_\vartheta \langle c_a; c, \delta\rangle$.

While Loop The last kind of transformation rules concerns the while loop $c \equiv \mathsf{while}_\Phi\ e\ \mathsf{do}\ c$. The rules (W1) and (W2) are similar to the transformation rules (C3) and (C4) for conditionals. The most important rule is (W1).

– if Var(e) ∩ Dom(δ) ≠ ∅ then by induction hypothesis, we have:

$$\langle c, \delta, C, \beta\rangle \Rightarrow_\vartheta \langle c', \delta', C, \beta\rangle,$$

and, for all $\sigma \in Mem$ we have

$$\langle c, \sigma\rangle \to^* \langle \mathsf{nop}, \sigma'\rangle,$$
$$\langle c', \sigma\rangle \to^* \langle \mathsf{nop}, \sigma''\rangle,$$

and $\sigma'(\vartheta) \sqsubseteq \sigma(\vartheta)$. We transform c as in the case of conditionals. In the operational semantics, we execute repeatedly $\langle c', \sigma\rangle \to^* \langle \mathsf{nop}, \sigma''\rangle$ instead of $\langle c, \sigma\rangle \to^* \langle \mathsf{nop}, \sigma'\rangle$ as long as the condition e is true. Then at each iteration σ, and $\sigma''(\vartheta) \sqsubseteq \sigma'(\vartheta)$. Consequently, by transitivity of $\sqsubseteq$ we obtain that

$$\langle \mathsf{while}_\Phi\ e\ \mathsf{do}\ c', \delta', C, \beta\rangle \prec_\vartheta \langle \mathsf{while}_\Phi\ e\ \mathsf{do}\ c, \delta, C, \beta\rangle.$$

□

We give hereafter a general case of Theorem 1.

Theorem 2 (multi-steps transformation) *We assume that c_o is the original program, c_t is the transformed program and ϑ is the reference variable to be improved. Assume also that $\langle c_o, \delta_o, C, \beta\rangle \Rightarrow_\vartheta^* \langle c_t, \delta_t, C, \beta\rangle$, then*

$$\langle c_t, \delta_t, C, \beta\rangle \prec_\vartheta \langle c_o, \delta_o, C, \beta\rangle.$$

□

Proof By induction on the length of the transformation (number of application of $\Rightarrow_\vartheta$). □

In the following, we extend our definitions and theorems to the abstract semantics introduced in Sect. 3.1.

Definition 3 *(abstract comparison of expressions)* Let $v_1^\sharp = (x_1^\sharp, \mu_1^\sharp) \in \mathbb{E}^\sharp$ and $v_2^\sharp = (x_2^\sharp, \mu_2^\sharp) \in \mathbb{E}^\sharp$. We say that $v_1^\sharp$ is more accurate than $v_2^\sharp$, denoted by $v_1^\sharp \sqsubseteq^\sharp v_2^\sharp$ iff $x_1^\sharp +^\sharp \mu_1^\sharp \subseteq x_2^\sharp +^\sharp \mu_2^\sharp$ and

$$\max(|\underline{\mu_1}|, |\overline{\mu_1}|) \leq \max(|\underline{\mu_2}|, |\overline{\mu_2}|),$$

where $\mu_1^\sharp = [\underline{\mu_1}, \overline{\mu_1}]$ and $\mu_2^\sharp = [\underline{\mu_2}, \overline{\mu_2}]$. □

Abstract environments of values are defined, which map identifiers to abstract values. Let $\mathscr{V}$ be a set of identifiers and let $\sigma^\sharp \in \mathsf{Mem}^\sharp$ be an environment that map variables of $\mathscr{V}$ to abstract values

$$\sigma^\sharp : \mathscr{V} \to \mathbb{E}^\sharp. \tag{34}$$

We assume given an abstract semantics of commands which uses states defined by a pair of command and abstract memory:

$$\text{State}^\sharp = \{\langle c, \sigma^\sharp\rangle : c \in \text{Cmd}, \sigma^\sharp \in \text{Mem}^\sharp\}. \tag{35}$$

The operational semantics maps abstract states to abstract states

$$\begin{aligned} \to^\sharp :\ & \text{State}^\sharp \to \text{State}^\sharp \\ & \langle c, \sigma^\sharp\rangle \mapsto \langle c', \sigma'^\sharp\rangle. \end{aligned} \tag{36}$$

Such operational semantics correspond to standard abstract interpretations used in static analyses [8,18,19,30]. Definition 4 extends the comparison of commands to abstract environments.

Definition 4 *(abstract comparison of commands)* Let c_o and c_t be two commands, let δ_o and δ_t be two formal environments. Finally, let ϑ be the reference variable. We say that

$$\langle c_t, \delta_t \rangle \prec_{\vartheta}^{\sharp} \langle c_o, \delta_o \rangle$$

if and only if for all $\sigma^{\sharp} \in \text{Mem}$,

$$\exists \sigma_o^{\sharp} \in \text{Mem}, \langle c_o, \sigma^{\sharp} \rangle \rightarrow^{\sharp *} \langle \text{nop}, \sigma_o^{\sharp} \rangle,$$
$$\exists \sigma_t^{\sharp} \in \text{Mem}, \langle c_t, \sigma^{\sharp} \rangle \rightarrow^{\sharp *} \langle \text{nop}, \sigma_t^{\sharp} \rangle,$$

such that:

- $\sigma_t^{\sharp}(\vartheta) \sqsubseteq^{\sharp} \sigma_o^{\sharp}(\vartheta)$,
- $\forall \text{id} \in \text{Dom}(\sigma_o^{\sharp}) \backslash \text{Dom}(\sigma_t^{\sharp})$, $\delta_t(\text{id}) = e$ and

  $\langle e, \sigma^{\sharp} \rangle \rightarrow_e^{\sharp *} \sigma_o^{\sharp}(\text{id})$.

□

Theorem 3 (static soundness of commands transformation) *Let c_o be the original code, c_t be the transformed code, δ_o be the initial environment of the transformation, δ_t be the final environment of the transformation, then we have*

$$\big(\langle c_o, \delta_o, C, \beta \rangle \Rightarrow_{\vartheta} \langle c_t, \delta_t, C, \beta \rangle\big) \Longrightarrow \big(\langle c_t, \delta_t, C, \beta \rangle \prec_{\vartheta}^{\sharp} \langle c_o, \delta_o, C, \beta \rangle\big).$$

□

Obviously, Theorem 2 can be easily extended to the abstract case in the same way than Theorem 3. Beside this formal proof of correctness, to go further with this question, complementary work is carried out in this direction in order to certify and validate that the programs transformed by our tool are equivalent to the original programs by using the Coq proof assistant [23]. More precisely, we aim at generating certificates, written as formal proofs, stating that the generated programs are mathematically equivalent and more accurate than the source programs.

6 Examples

In Sect. 4, we have presented the transformation rules implemented in our tool in order to improve the numerical accuracy of programs and we have shown their ability to improve the numerical accuracy of pieces of code, especially for standard control-command algorithms used in embedded systems.

As a step toward ensuring the efficiency of our tool and in order to perform experiments with it, we have considered a number of examples in the domains of robotics, avionics, numerical analysis, etc. These examples have been chosen for their representativity and have been written as the average programmer would do since our tool is intended to optimize programs written by non-specialists. In another work, we are participating in the definition of `FPBench` a benchmark suite for numerical accuracy optimization tools [12,35] and numerical validation [15,18,36].

Listing 3 Listing of the initial PID program

```
m = [4.5,9.0]; kp = 9.4514; ki = 0.69006;
kd = 2.8454; t = 0.0; i = 0.0; eold = 0.0;
c = 5.0; dt = 0.2; invdt = 5.0;
while true do {
 e = c - m;
 p = kp * e;
 i =  i + ((ki * dt) * e);
 d = ((kd * invdt) * (e - eold));
 r = ((p + i) + d);
 m = m + (0.01 * r);
 eold = e;
 t = t + dt;
}
```

First, we briefly describe each of these programs and of their parameters. As for the odometry example of Sect. 2, in our experiments, some of inputs of the programs are intervals and our tool optimizes the accuracy of the reference variable in the worst case for all the possible values given by the ranges.

Obviously, our tool is sensitive to the given intervals and since we compute the accuracy in the worst case, the larger the intervals are, the least the optimization is. The impact of the width of the intervals is discussed in [24,28]. In Sects. 6.1 to 6.5, we give the listings before and after being optimized. Their accuracy is then discussed in Sect. 6.6. Note that all our experiments are done using the rounding mode to the nearest.

6.1 PID controller

The PID [14] is an algorithm widely used in embedded and critical systems, like aeronautic and avionic systems. It keeps a physical parameter at a specific value known as the *setpoint*. In other words, it tries to correct a measure by maintaining it at a defined value. To compute this correction, the controller incorporates three terms: the integral term i and the derivative term d of the error, as well as a proportional error term p. The error e is the difference between the setpoint c and the measure m. After discretization, we have

$$\begin{aligned} e &= c - m, \\ p &= k_p \times e, \\ i &= i + k_i \times e \times dt, \\ d &= k_d \times (e - e_{old}) \times \tfrac{1}{dt}. \end{aligned} \tag{37}$$

The weighted sum of these terms contributes to improve the reactivity, the robustness and the speed of the program. We assume that $m \in [4.5, 9.0]$. The program correspond-

Listing 4 Listing of the optimized PID program

```
m = [4.5,9.0];  t = 0.0; i = 0.0;
while true do {
 i = (i + (0.138012 * (5.0 - m)));
 eold = (5.0 - m);
 m = (m + (0.01 * ((((5.0 - m) * 9.4514)
   + i) + (((5.0 - m) - eold) * 14.227))));
 t = t + 0.2;
}
```

Listing 5 Listing of the initial Lead–Lag program

```
y = [2.1,17.9]; xc0 = 0.0; xc1 = 0.0; yd = 5.0;
Ac11 = 1.0; Bc0 = 1.0; Bc1 = 0.0; Cc0 = 564.48;
Ac00 = 0.499;  Ac01 = -0.05; Ac10 = 0.01;
Cc1 = 0.0; Dc = -1280.0; t = 0.0;
while(t < 5.0) do {
 yc = (y - yd);
 if(yc < -1.0) { yc = -1.0; }
 if(1.0 < yc) { yc = 1.0; }
 xc0 = (Ac00 * xc0)+(Ac01 * xc1)+(Bc0 * yc);
 xc1 = (Ac10 * xc0)+(Ac11 * xc1)+(Bc1 * yc);
 u = (Cc0 * xc0)+(Cc1 * xc1)+(Dc * yc);
 t = (t + 0.1);
}
```

ing to the implementation of the PID controller is given in Listing 7.

We optimize the initial PID program shown in Listing 7, in order to improve its numerical accuracy, by applying the transformation rules presented previously on Fig. 5 to it. In this case, our tool has simplified and developed expressions and then inlined them into other expressions. Listing 4 shows the optimized program.

6.2 Lead–lag system

Our second example is a dynamical system. This system includes a single mass and a single spring and is governed by an automatically synthesized controller [17] which tries to move the mass from the initial position y to the desired one y_d as illustrated in Listing 8. The main variables in this algorithm are: x_c consists of the discrete-time controller state, y_c is the bounded output tracking error and u presents the mechanical system output. We assume that the position y of the mass $m \in [2.1,17.9]$. The parameters of the system are given in Eq. (38).

$$\begin{aligned} yc &= \mathsf{max}(\mathsf{min}(y - yd, 1), -1);\\ u &= Cc * xc + Dc * yc;\\ xc &= Ac * xc + Bc * yc. \end{aligned} \tag{38}$$

Listing 5 gives the code of the original Lead–Lag program. By optimizing the numerical accuracy of the program presented in Listing 5, we obtain the transformed program given in Listing 6.

Listing 6 Listing of the transformed Lead–Lag program

```
y = [2.1,17.9]; t = 0.0; xc1 = 0.0; xc0 = 0.0;
while true do {
 yc = (-5.0+y);
 if(yc < -1.0) { yc = -1.0; }
 if(1.0 < yc)  { yc = 1.0; }
 u = (((564.48 * xc0)+(0.0 * xc1))+
(-1280.0 * yc));
 xc0 = (((-0.05 * xc1)+(1.0 * yc))+
(0.499 * xc0));
 xc1 = (((0.01 * xc0)+(0.0 * yc))+(1.0 * xc1));
 t = (t + 0.1);
}
```

Listing 7 Listing of the initial Runge–Kutta program

```
yn = [-10.1,10.1]; t = 0.0; k = 1.2; c = 100.1;
h = 0.1;
while (t < 1.0) do {
 k1 = (k * (c - yn)) * (c - yn);
 k2 = (k * (c - (yn + ((0.5 * h) * k1))))
    * (c - (yn + ((0.5 * h) * k1)));
 k3 = (k * (c - (yn + ((0.5 * h) * k2))))
    * (c - (yn + ((0.5 * h) * k2)));
 k4 = (k * (c - (yn + (h * k3))))
    * (c - (yn + (h * k3)));
 yn+1 = yn + ((1/6 * h) * (((k1 + (2.0 * k2))
      + (2.0 * k3)) + k4));
 t = (t + h);
}
```

6.3 Runge–Kutta methods

This example concerns Runge–Kutta methods [26]. We consider an order 2 and an order 4 method. They are employed to solve the equation describing the dynamics of a chemical reaction $A + B \rightarrow C$. The order 2 method integrates a differential equation whose solution is $y(t)$. The second-order method uses the derivative on the starting point x_i in order to find the intermediary point. Then, it uses this intermediary point to have the next value of the function. The derivative of $y(x)$ at the points x_i and $x_i + \frac{h}{2}$ are

$$\begin{aligned} k_1 &= \left(\frac{\mathrm{d}y}{\mathrm{d}x}\right) = h \times f(x_i, y_i),\\ k_2 &= \left(\frac{\mathrm{d}y}{\mathrm{d}x}\right) = h \times f\left(x_i + \frac{h}{2}, y_i + \frac{h}{2}\right). \end{aligned} \tag{39}$$

Finally, we have $y_{i+1} = y_i + k_2 + O(h^3)$. We assume that initially, $y_0 \in [-10.1, 10.1]$. For the order 4 method, we obtain as final formula:

$$y_{i+1} = y_1 + \tfrac{1}{6}\left[k_1 + 2 \times k_2 + 2 \times k_3 + k_4\right] \times h, \tag{40}$$

with

$$k_1 = h \times f(x_i, y_i), \qquad k_2 = h \times f\left(x_i + \frac{h}{2}, y_i + \frac{h}{2}\right),$$
$$k_3 = h \times f\left(x_i + \frac{h}{2}, y_i + \frac{h}{2}\right), \qquad k_4 = h \times f\left(x_i + \frac{h}{2}, y_i + \frac{h}{2}\right).$$

Listing 8 Listing of the transformed Runge–Kutta program

```
yn = [-10.1,10.1]; t = 0.0;
while(t < 1.0) do {
 TMP_7 = (1.2 * (100.099 - yn));
 TMP_8 = (100.099 - yn);
 TMP_13 = (1.2 * (100.099 - (yn + (0.05 * ((1.2
        * (100.099 - (yn + (0.05 * (TMP_7
        * TMP_8))))) * (100.099 - (yn + (0.05
        * ((1.2 * TMP_8) *
(100.099 - yn)))))))))));
 TMP_14 = (100.099 - (yn + (0.05 * ((1.2 *
(100.099
        - (yn + (0.05 *(TMP_7 * TMP_8))))) *
 (100.099
        - (yn + (0.05 * ((1.2 * TMP_8)
        * (100.099 - yn));
 TMP_18 = (yn + (0.05 * ((1.2 * (100.099 - (yn
        + (0.05 *(TMP_7 * TMP_8))))) * (100.099
        - (yn + (0.05 * ((1.2 * TMP_8)
        * (100.099 -yn)))))))));
 TMP_28 = ((1.2 * (100.099 - (yn + (0.05 *
 (TMP_7
        * TMP_8))))) *(100.099 - (yn + (0.05
        * ((1.2 * TMP_8) * (100.099 - yn))))));
 TMP_38 = ((TMP_14 * TMP_13) * 0.1) + yn;
 TMP_40 = 0.1 * ((1.2 * TMP_14)*(100.099 -
TMP_18));
 yn_plus_1 = (yn + (0.016666667 *
((((TMP_7 * TMP_8)
           + (2.0 * TMP_28)) + (2.0 * (TMP_13
           * TMP_14))) + ((1.2 * (100.099
- TMP_38))
           * (100.099 - (yn + TMP_40))))));
 [...] ;
 t = (t + 0.1);
}
```

Listing 9 Listing of the initial trapeze program

```
u = [1.11, 2.22]; a = 0.25; b = 5000.0;

n = 25.0;
r = 0.0; xa = 0.25; h = ((b - a) / n);
while (xa <  5000.0) do {
 xb = (xa + h) ;
 if(xb > 5000.0) {
   xb = 5000.0 ;
   gxa = (u / ((((((0.7 * xa) * xa) * xa)
       - ((0.6 * xa) * xa)) + (0.9 * xa))
- 0.2));
   gxb = (u / ((((((0.7 * xb) * xb) * xb)
       - ((0.6 * xb) * xb)) + (0.9 * xb))
- 0.2));
   r = (r + (((gxb + gxa) * 0.5) * h));
   xa = (xa + h);
 }
}
```

Listing 10 Listing of the transformed trapeze program

```
u = [1.11, 2.22]; xa = 0.25; r = 0.0;
while (xa <  5000.0) do {
 TMP_1 = (0.7 * (xa + 199.99));
 TMP_2 = (xa + 199.99);
 TMP_9 = ((((0.7 * xa) * xa) * xa) - ((0.6 * xa)
       * xa)) + (0.9 * xa);
 TMP_11= (((199.99 + xa) * (TMP_2 * TMP_1))
       - ((199.99 + xa) * (TMP_2 * 0.6)))
       + (0.9 * TMP_2);
 r = (r + ((((u / (TMP_11 - 0.2)) + (u / (TMP_9
   - 0.2))) * 0.5) * 199.99));
 xa = (xa + 199.99);
}
```

The function f (see Eq. (39)) needs to be evaluated four times to improve its accuracy. We assume that $k_i = \frac{dy}{dx}$. Equation (40) will be described by:

$$y_{i+1} = y_1 + \frac{1}{6}\left[k_1 + 2 \times k_2 + 2 \times k_3 + k_4\right]. \tag{41}$$

The listing of the initial program performing the Eq. (41) is presented in Listing 7.

Instead, by applying a sequence of transformation rules to the initial Runge–Kutta code, we obtain a program considerably more accurate than the original one. This optimized program is given in Listing 8.

6.4 The Trapezoidal rule

This example concerns an algorithm for the trapezoidal rule [26], well known in numerical analysis to approximate the definite integral $\int_a^b f(x)\,\mathrm{d}x$. This trapezoidal rule works by approximating the region between x and $x + h$ under the graph of the function f(x) as a trapezoid and calculates its area. Here, we compute the integral $\int_{0.25}^{5000} g(x)dx$ of some function:

$$g(x) = \frac{u}{0.7x^3 - 0.6x^2 + 0.9x - 0.2}. \tag{42}$$

We assume that u is a user-defined parameter in the range [1.11, 2.22]. In addition, we have unfolded the body of the loop twice to obtain better results with our tool. The program performing the function $g(x)$ corresponding to Eq. (42) is described in Listing 9.

Once the trapezoidal rule program is given to our tool, many transformation rules would be applied on it to improve its accuracy. Listing 10 corresponds to the program generated after optimization.

6.5 Rocket trajectory simulation

Our tool demonstrated its efficiency on small programs by improving their numerical accuracy. Among other examples, we have taken a larger example that contains more than one hundred lines of code [13]. The example computes the positions of a rocket and a satellite in space. It consists of simulating their trajectories around the Earth using the Cartesian and polar systems, in order to project the gravitational forces in the system composed of the Earth, the rocket and the satellite. Note that the coordinates of the satellite u_i and of the rocket w_i, $1 \leq i \leq 4$ are computed by Euler's method.

The program corresponding to this example is given in Listing 11. The `else` branch is similar to the `then` branch at the difference that w_2' and w_4' are computed without the

Listing 11 Listing of the initial rocket trajectory simulation program

```
u = [1.11, 2.22]; xa = 0.25; r = 0.0; M_f
= 150000;
R = 6.4 * 10e6; G = 6.67428 * 10e-11; A = 140;
M_t = 5.9736 * 10e24;
D = R + 4.0 * 10e5;
v_l = 0.7 * sqrt((G * M_t)(D));
while (i < nbsteps) do {
 if(m_f > 0.0){
   u'1 = u2 * dt + u1;
   u'3 = u4 * dt + u3;
   w'1 = w2 * dt + w1;
   w'3 = w4 * dt + w3;
   u'2 = -G * Mt / (u1 * u1) * dt + u1 *
 u4 * u4
        * dt + u2;
   u'4 = -2.0 * u2 * u4 / u1 * dt + u4;
   w'2 = -G * Mt / (w1 * w1) * dt + w1 *
 w4 * w4
        * dt + (A * w2) / (Mf - A * t) * dt
+ w2;
   w'4 = -2.0 * w2 * w4 / w1 * dt + A * w4
        / (Mf - A * t) * dt + w4;
   m'f = mf - A * t ;
   t = t + dt;
 }
 else
 {  [...] }
 x = cos(w'3)*w'1);
 y = sin(w'3)*w'1);
 i = i + 1.0 ;
 [...]
}
```

Listing 12 Listing of the transformed rocket trajectory simulation program

```
u = [1.11, 2.22]; xa = 0.25; r = 0.0;
while (i < nbsteps) do {
 if (m_f > 0.0) {
   TMP_2 = (u1 * u1);
   TMP_4 = (59735.99e20 / (w1 * w1));
   TMP_10 = (140.0 * t);
   m'f = (mf + (t * (-140.0)));
   u'1 = (u1 + (u2 * 0.1));
   u'3 = (u3 + (u4 * 0.1));
   w'1 = (w1 + (w2 * 0.1));
   w'3 = (w3 + (w4 * 0.1));
   u'2 = ((-((0.66743e(-10) * (59735.99e(20)
        / TMP_2)) * 0.1) + (((u1 * u4) * u4)
        * 0.1)) + u2);
   u'4 = (((-2.0 * (u2 * (u4 / u1))) * 0.1)
 + u4);
   w'2 = (((-((0.66743e(-10) * TMP_4) * 0.1)
+ (((w1
        * w4) * w4) *  0.1)) + (((140.0 *  w2)
        / (150000.0 - (140.0 * t))) * 0.1))
 + w2);
   w'4 = (((-2.0 *  (w2 * (w4 / w1))) * 0.1)
        + ((140.0 * ((w4 / (150000.0 - TMP_10))
        * 0.1)) + w4));
   t = t + 0.1;
 }
 else { [...] }
 x = (cos(((0.1 * w'4)+w'3)) * (w'1+w'2
* 0.1)));
 y = (sin(((0.1 * w'4)+w'3)) * (w'1+(w'2
* 0.1)));
 i = i + 1.0;
 [...]
}
```

expression $A \cdot w_i/(M_f - A \cdot t) \cdot \mathrm{d}t$. At the end of the loop, variables are updated, for example $u_1 = u'_1$, etc.
Constants are: the radius of the Earth $R = 6.4 \cdot 10^6\ m$, the gravity $G = 6.67428 \cdot 10^{-11}\ m^3\ kg^{-1}\ s^{-2}$, the mass of the Earth $M_t = 5.9736 \times 10^{24}$ kg, the mass of the rocket $M_f = 150,000$ kg and the gas mass ejected by second $A = 140\ kg \cdot s^{-1}$. The release rate of the rocket v_l is $0.7 \cdot \sqrt{((G \cdot M_t)(D)}$ with $D = R + 4.0 \cdot 10^5\ m$ the distance between the rocket and the Earth. Other variables are set to 0.

Listing 11 illustrates the initial rocket trajectory simulation program. Table 3 shows the program that corresponds to the optimized rocket trajectory simulation. Lastly, when observing the behavior of both trajectories before and after being optimized, a significant difference between the two curves can be seen. This difference shows on how much we improve the numerical accuracy of this large program. Note that Listing 9 is obtained after 2.25 days of simulated time.

6.6 Experiments

Our prototype consists of an implementation of the rules described in Sect. 4 coupled to the APEG tool for the transformation of expressions. For the demonstration of its efficiency, we evaluate through it the examples described previously in this section. Our tool takes as input an initial program and intervals for some parameters and returns another program mathematically equivalent but numerically more accurate as long as the parameters remain in the given ranges.

Table 3 compares the initial and the new error of each of programs presented in this section and shows as well as how much our tool improves the numerical accuracy of these programs. For example, if we take the case of `odometry`, we observe that we optimize it by 21.43 %. If we compare the implementation of Runge–Kutta method, we remark that the order four methods is improved of 15.87 %. The lead–lag system is optimized by 19.96 %. The improvement of the error is given in Table 3, where s is the slice size, i.e., the parameter defining at which height of the syntactic tree we cut the expressions.

Note that all the example programs are optimized. This is natural since the original codes are direct implementations of the mathematical formulæ and have not been written for the floating-point arithmetic. Nevertheless, our tool never returns a code whose accuracy in the worst case is worst than the original program. In the worst case, we return the original program without optimization.

In general, execution times matter when dealing with programs performing floating-point computations. Indeed, numerical simulations are time consuming and embedded software is real time. So, in addition to improving the accuracy of programs described above by using automatic transformation, we evaluated our tool on their execution time.

Table 3 Initial and new errors on the examples programs of Sect. 6

Code	Initial error	New error	s	%
PID	$0.453945103062736 \times 10^{-14}$	$0.440585745442590 \times 10^{-14}$	5	2.94
Odometry	$0.106578865995068 \times 10^{-10}$	$0.837389354639250 \times 10^{-11}$	5	21.43
RK2	$0.750448486755706 \times 10^{-7}$	$0.658915054553695 \times 10^{-7}$	5	12.19
RK4	$0.201827996912328 \times 10^{-1}$	$0.169791306481639 \times 10^{-1}$	5	15.87
Lead-Lag	$0.294150262243136 \times 10^{-11}$	$0.235435212105148 \times 10^{-11}$	10	19.96
Trapezoid	$0.536291684923368 \times 10^{-9}$	$0.488971110442931 \times 10^{-9}$	20	8.82

Table 4 Execution time measurements of programs of Sect. 6

	Original code execution time in s	Optimized code execution time in s	Percentage improvement (%)	Mean on n runs
Odometry	3.2×10^{-2}	2.0×10^{-2}	37.5	10^4
PID controller	2.0×10^{-2}	0.8×10^{-2}	60.0	10^2
Lead-lag system	8.0×10^{-2}	6.0×10^{-2}	25.0	10^3
Runge-Kutta 4	1.6×10^{-2}	1.2×10^{-2}	25.0	10^3
Trapezoidal rule	2.0×10^{-2}	0.8×10^{-2}	60.0	10^4
Rocket	0.82×10^2	0.61×10^2	25.6	10^2

Our evaluation shows the effect of improving the numerical accuracy of programs through the examples described previously, on their execution time. In Table 4, we compare the execution time needed by each program before and after optimization. For example, if we take the case of odometry, we observe that we optimize its execution time by 37.5 %. More generally, the percentage of execution time improvement of all programs presented before, is between 25 and 60 % which is significantly important. Note that, the original and optimized programs have been implemented in an imperative programming language, compiled with our tool, and made them run on an Intel Core i5 with 4 Go memory in IEEE754 single precision in order to emphasize the effect of the finite precision.

Our tool relies on a static analysis which computes a range for each variable at each control point of program. Obviously, the more the static analysis is precise, the more the transformation is efficient. In our current implementation, we use a simple non-relational abstract domain described in Sect. 3. For this reason, in our examples, we have given a finite number of iterations to the loops, in order to help the static analysis. However, this is independent of the transformation itself and we would obtain similar results with more general exit conditions (e.g., with a convergence criterion) if a better static analysis were used. Indeed, in another side of our research, we have explored many numerical iterative methods to detect the impact of improving their numerical accuracy on their convergence speed [11]. We have carried out our experiments using four well-known methods, namely the Jacobi's method, the Newton–Raphson's method, an iterative Gram–Schmidt method and finally a method that computes the eigenvalues of a matrix. We have demonstrated that we can reduce the total number of iterations required by these methods to converge by about 20% when their accuracy is optimized.

7 Conclusion

In this article, we have presented the principles of a tool that rewrites automatically programs in order to improve their numerical accuracy. More precisely, we have shown how to perform intra-procedural rewritings of commands and how to transform assignments. In the rules of Fig. 5, correctness conditions have been defined to guarantee that the dependencies are respected and to ensure the correctness of the rewritings in conditions and loops.

The soundness of the transformation relies on Theorems 1, 2 and 3 given in Sect. 5. In order to validate our tool, we have chosen a set of representative programs taken from various fields of sciences and engineering. We have automatically tuned them and analyzed their accuracy before and after transformation. The experimental results show the usefulness of our approach on the accuracy. A step further, we have extended this study with complementary results concerning the execution time of each of programs.

In future research, we will generalize our techniques to cover other kinds of programming language patterns like pointers, arrays, and especially functions in order to obtain an intra-procedural program transformation with function refactoring and specialization with respect to the values of arguments.

Another extension aims at extending our approach to optimize several reference variables simultaneously. A difficulty is that the optimization of one variable may decrease the accuracy of other variables. Compromises have to be made. Finally, our transformation relies on a static analysis of the source code. Indeed, we select the optimized program by using the abstract semantics in Sect. 3.1, we compute certified error bounds which can be over-approximated. We would like to improve it by using more accurate relational domains in order to obtain finer error bounds completed by statistical results on the actual accuracy gains on concrete executions.

The transformation introduced in this article is based on a static analysis which computes an over-approximation of the round-off errors due to the floating-point arithmetics. However, the transformation itself is independent of the computation of the errors, it only depends on the order $\sqsubseteq$ introduced in Definition 1 and using another order would change the objective of the transformation. We could use it with another static analysis which would estimate differently which expression of an APEG is the best. For example, we could use the compromise between accuracy and execution time or different arithmetics such as the fixed-point arithmetics which is widely used in embedded systems. A way of coupling our transformation with fixed, integer or interval arithmetics is discussed in [31].

A key issue is to study the impact of accuracy optimization on the convergence time of distributed numerical algorithms like the ones used usually for high-performance computing. In addition, still on distributed systems, an important issue concerns the reproducibility of the results: different runs of the same application yield different results due to the variations in the order of evaluation of the mathematical expression. We would like to study how our technique could improve reproducibility.

References

1. Almagor, L., Cooper, K.D., Grosul, A., Harvey, T.J., Reeves, S.W., Subramanian, D., Torczon, L., Waterman, T.: Finding effective compilation sequences. In: Whalley D.B., Cytron, R. (eds.) ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'04), pp. 231–239. ACM (2004)
2. ANSI/IEEE. IEEE Standard for Binary Floating-point Arithmetic, std 754-2008 edition (2008)
3. Appel, A.-W.: Modern compiler implementation in ML. Cambridge University Press, Cambridge (1998)
4. Barr, E.T., Vo, T., Le, V., Su, Z.: Automatic detection of floating-point exceptions. In: Giacobazzi, R., Cousot, R. (eds.) ACM SIGPLAN-SIGACT POPL'13, pp. 549–560, ACM (2013)
5. Benz, F., Hildebrandt, A., Hack, S.: A dynamic program analysis to find floating-point accuracy problems. In: Vitek, J., Lin, H., Tip, F. (eds.) ACM SIGPLAN PLDI'12, pp. 453–462. ACM (2012)
6. 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)
7. Cooper, K.D., Subramanian, D., Torczon, L.: Adaptive optimizing compilers for the 21st century. J. Supercomput. **23**(1), 7–22 (2002)
8. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Graham, R.M., Harrison, M.A., Sethi, R. (eds.) POPL'77, pp. 238–252 ACM (1977)
9. Cousot, P., Cousot, R.: Systematic design of program transformation frameworks by abstract interpretation. In: Launchbury, J., Mitchell, J.C. (eds.) POPL'02, pp. 178–190. ACM (2002)
10. Cytron, R., Gershbein, R.: Efficient accomodation of may-alias information in SSA form. In PLDI'93, pp. 36–45, ACM (1993)
11. Damouche, N., Martel, M., Chapoutot, A.: Impact of accuracy optimization on the convergence of numerical iterative methods. In: Falaschi, M. (ed.) LOPSTR 2015, volume 9527 of Lecture Notes in Computer Science, pp. 143–160. Springer (2015)
12. Damouche, N., Martel, M., Chapoutot, A.: Intra-procedural optimization of the numerical accuracy of programs. In M. Núñez and M. Güdemann, editors, FMICS'15, volume 9128 of Lecture Notes in Computer Science, pp. 31–46 Springer (2015)
13. Damouche, N., Martel, M., Chapoutot, A.: Optimizing the accuracy of a rocket trajectory simulation by program transformation. In: CF'15, pp. 40:1–40:2. ACM (2015)
14. Damouche, N., Martel, M., Chapoutot, A.: Transformation of a PID controller for numerical accuracy. Electr. Notes Theor. Comput. Sci. **317**, 47–54 (2015)
15. Darulova, E., Kuncak, V.: Sound compilation of reals. In: Jagannathan, S., Sewell, P. (eds.) POPL'14, pp. 235–248. ACM (2014)
16. Delmas, D., Goubault, E., Putot, S., Souyris, J., Tekkal, K. Védrine, F.: Towards an industrial use of FLUCTUAT on safety-critical avionics software. In: FMICS'09, pp. 53–69 (2009)
17. Feron, E.: From control systems to control software, ieee control systems magazine. IEEE **30**, 50–71 (2010)
18. Goubault E.: Static analysis by abstract interpretation of numerical programs and systems, and FLUCTUAT. In: Logozzo, F., Fähndrich, M. (eds.) SAS'13, volume 7935 of Lecture Notes in Computer Science, pp. 1–3. Springer (2013)
19. Goubault E., Putot S.: Static analysis of finite precision computations. In: Jhala, R., Schmidt, D.A. (eds.) VMCAI'11, volume 6538 of Lecture Notes in Computer Science, pp. 232–247. Springer (2011)
20. Feret, J.: Static analysis of digital filters. In: Schmidt, D.A. (ed.) Programming Languages and Systems, 13th European Symposium on Programming, ESOP 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS, volume 2986 of Lecture Notes in Computer Science, pp. 33–48. Springer (2004)
21. Gao, X., Bayliss, S., G. A. Constantinides. SOAP: structural optimization of arithmetic expressions for high-level synthesis. In: FPT'13, pp. 112–119. IEEE (2013)
22. Hankin, E.: Lambda Calculi A guide for computer scientists. Clarendon Press, Oxford (1994)
23. Huet, G., Kahn, G., Paulin-Mohring, Ch.: The Coq ProofAssistant—A tutorial-Version 8.0, (2004)
24. Ioualalen, A., Martel, M.: A new abstract domain for the representation of mathematically equivalent expressions. In: SAS'12, volume 7460 of LNCS, pp. 75–93. Springer (2012)

25. Jones, N.D.: An introduction to partial evaluation. ACM Comput. Surv. **28**(3), 480–503 (1996)
26. Kendall, A.: An introduction to numerical analysis. Wiley, Hoboken (1989)
27. Knoop, J., R"uthing, O., Steffen, B.: Optimal code motion: Theory and practice. ACM Transactions on Programming Languages and Syst. 16(4):1117–1155 (1994)
28. Langlois, P., Martel, M., Thévenoux, L.: Accuracy versus time: a case study with summation algorithms. In: Maza, M.M., Roch, J.-L. (eds) 4th International Workshop on Parallel Symbolic Computation, PASCO, pp. 121–130. ACM (2010)
29. Lerner, S., Grove, D., Chambers, C.: Composing dataflow analyses and transformations. In: Launchbury, J., Mitchell, J.C. (eds) 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 270–282. ACM (2002)
30. Martel, M.: Semantics of roundoff error propagation in finite precision calculations. High. Order Symb. Comput. **19**(1), 7–30 (2006)
31. Martel, M.: Accurate evaluation of arithmetic expressions (invited talk). Electr. Notes Theor. Comput. Sci. **287**, 3–16 (2012)
32. Miné, A.: Relational abstract domains for the detection of floating-point run-time errors. In D. A. Schmidt, editor, Programming Languages and Systems, 13th European Symposium on Programming, ESOP 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Proceedings, volume 2986 of Lecture Notes in Computer Science, p. 3–17. Springer (2004)
33. Monniaux, D.: The pitfalls of verifying floating-point computations. ACM Trans. Program. Lang. Syst. 30(3) 2008
34. 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. B. Boston (2010)
35. Wilcox, J. R., Panchekha, P., Sanchez-Stern, A., Tatlock, Z.: Automatically improving accuracy for floating point expressions. In: Grove, D., Blackburn, S. (eds.) ACM SIGPLAN PLDI'2015, pp. 1–11. ACM (2015)
36. Solovyev, A., Jacobsen, C., Rakamaric, Z., Gopalakrishnan, G.: Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. In FM'15, volume 9109 of LNCS, pp. 532–550. Springer (2015)
37. Steffen, B., Knoop, J., Rüthing, O.: The value flow graph: a program representation for optimal program transformations. In: Jones, N.D. (ed.) ESOP'90, 3rd European Symposium on Programming, volume 432 of Lecture Notes in Computer Science, pp. 389–405. Springer (1990)
38. Tate, R., Stepp, M., Tatlock, Z., Lerner, S.: Equality saturation: a new approach to optimization. In Z. Shao and B. C. Pierce, editors, ACM SIGPLAN-SIGACT POPL'09, pages 264–276. ACM, (2009)
39. Tate, R., Stepp, M., Tatlock, Z., Lerner, S.: Equality saturation: a new approach to optimization. Logical Methods in Computer Science 7(1) (2011)
40. Whitfield, D., Lou Soffa, M.: An approach for exploring code-improving transformations. ACM Trans. Program. Lang. Syst. **19**(6), 1053–1084 (1997)
41. Winskel, G.: The formal semantics of programming languages - an introduction. Foundation of computing series. MIT Press, Cambridge (1993)