Intra-procedural Optimization of the Numerical Accuracy of Programs

Nasrine Damouche, Matthieu Martel, Alexandre Chapoutot

Intra-Procedural Optimization of the Numerical Accuracy of Programs

Nasrine Damouche[1,2], Matthieu Martel[1,2], Alexandre Chapoutot[3]

[1] University of Perpignan Via Domitia, DALI Team-Project, France
[2] University of Montpellier II & CNRS, LIRMM, UMR 5506, France
[3]ENSTA ParisTech, Palaiseau, France

Abstract. Numerical programs performing floating-point computations are very sensitive to the way formulas are written. These last years, several techniques have been proposed concerning the transformation of arithmetic expressions in order to improve their 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 the numerical accuracy of the target variable. We use abstract interpretation-based static analysis techniques to over-approximate the roundoff errors in programs and during the transformation of expressions. A prototype has been implemented and experimental results are presented concerning classical numerical algorithm analysis and algorithm for embedded systems.

Keywords: Program Transformation, Floating-Point Numbers, Static Analysis, IEEE754 Standard.

1 Introduction

These last years, as the complexity of the floating-point computations [1, 23] carried out in embedded systems and elsewhere increased, numerical accuracy has become a more and more sensitive subject in computer science. Due to the important impact of accuracy on the reliability of embedded systems, many industries and companies encourage research to validate [5, 10, 14, 13] and improve [16, 21] their software in order to avoid failures and eventually disasters in aeronautics, automotives, robotics, etc.

In this article, we focus on the transformation [6, 8] 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 [16] 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 [16, 17] for Abstract Program Expression Graphs, 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 [7] techniques for the floating-point arithmetic [13].

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 more accurate numerically but equivalent semantically (in the sense that both programs compute the same function in exact arithmetic). This transformation operates by simplifying and developing the expressions and inlining them into other expressions. This allows one to generate new formulas and to reduce the number of operations in programs. We also rewrite the codes by unfolding the body of loops, manner to have more computations on a single iteration. The transformation of the odometry program and the rewriting rules used to automatically rewrite codes are the main contribution of this article. These rules are presented as sequents containing conditions under which the transformation may be applied without breaking the semantical equivalence between the source and target programs. In addition, these rules are applied deterministically, yielding a polynomial time transformation. This work is completed by experimental results involving the transformation of codes coming from multiple domains of science.

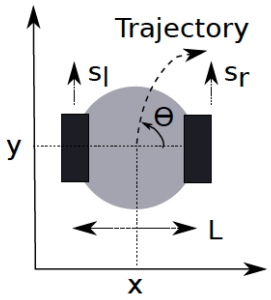
This article is organized as follows. Section 2 is consecrated to our case study about odometry and Section 3 introduces related work concerning the analysis and transformation of arithmetic expressions. In Section 4, we give the set of transformation rules for commands together with the conditions required to conserve the semantical equivalence of programs. Section 5 presents experimental results and shows various experimentations obtained using our prototype. Finally, Section 6 concludes.

2 Case Study: Odometry

In this section, we are interested in an example widely used in embedded systems, taken from robotics and whose code is given in Figure 2. It concerns the computation of the position of a two wheeled robot by odometry. Given the instantaneous rotation speeds s_l and s_r of the left and right wheels, we aim at computing the position of the robot in a cartesian space (x, y). Let C be the circumference of the wheels of the robot and L the length of its axle (see Figure 1). We assume that s_l and s_r are updated by the system, by side-effect. The computation of the position is given by

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

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



It	x (Odometry$_1$)	x (Odometry$_2$)
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. 1. Left: Parameters of the two-wheeled robot. Right: Values of x in `Odometry`$_1$ and `Odometry`$_2$ at the first iterations.

with

$$\theta(t+1) = \theta(t) + \Delta\theta(t), \quad \Delta d(t) = \big(\Delta d_r(t) + \Delta d_l(t)\big) \times 0.5, \tag{3}$$

$$\Delta\theta(t) = \big(\Delta d_r(t) - \Delta d_l(t)\big) \times \frac{1}{L}, \quad \Delta d_l(t) = s_l(t) \times C, \quad \Delta d_r(t) = s_r(t) \times C. \tag{4}$$

In equations (1) to (4), $\theta(t)$ is the direction of the robot, $d(t)$ is the elementary movement of the robot at time t and $d_l(t)$, $d_r(t)$ 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 development as shown in Equation (5).

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

We aim at rewriting the initial program `Odometry`$_1$ into a better program `Odometry`$_2$

```
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. 2. Listing of the initial Odometry program.

which improves the numerical accuracy of the computed position. The speed of

```
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 ; }
```

Fig. 3. Listing of the transformed Odometry program.

the left wheel is assumed to belong to an interval of $[0.52, 0.53]$ radians per second. Our prototype develops and simplifies the expressions δ_d, `cos` and `sin` and then inline them within the loop, in x and y. In addition, it creates new intermediary variables, called `TMP`, in order to avoid to have too large expressions. This process makes it possible to produce constant formulas and, in the same time, reduces the number of operations in the program. Furthermore, the resulting expressions are rewritten using existing techniques for the transformation of arithmetic expressions based on the use of Abstract Program Equivalence Graphs [16, 21]. We obtain the final program given in Figure 3. If we compare the resulting values x_1 and x_2 of $\texttt{Odometry}_1$ and $\texttt{Odometry}_2$, we observe that the transformation leads to a significant difference in the accuracy of the program as shown in Figure 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 (x, y) of the robot is shown in Figure 4.

3 Transformation of Expressions

This section introduces related work concerning the static analysis of the accuracy and the transformation of expressions. The syntax of expressions is

$$\text{Expr} \quad \ni \quad e ::= id \mid cst \mid e + e \mid e - e \mid e \times e \mid e \div e. \tag{6}$$

Expressions in Equation (6) are made of variables $id \in \mathcal{V}$ with $\mathcal{V}$ a finite set, constants $cst \in \mathbb{F}$ with $\mathbb{F}$ the set of floating-point numbers and of the four elementary operations $+$, $-$, $\times$ and $\div$.

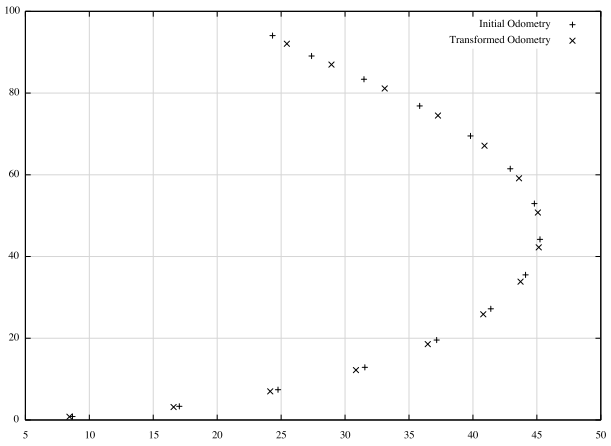


Fig. 4. Computed trajectories by the initial and the transformed odometry programs.

3.1 Static Analysis of the Accuracy

In order to compute safe bounds on the accuracy of arithmetic expressions, an abstract value is defined by a pair of intervals representing the range of the floating-point value seen by the program and the range of the error i.e., the difference between the floating-point and the exact value [22]. An abstract value is denoted by $(x^\sharp,\ \mu^\sharp) \in E^\sharp$ where $x^\sharp$ is the interval of values of the input and $\mu^\sharp$ is the interval of errors on the input. It 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. When working with arithmetic expressions, the propagation of roundoff errors is given by the following semantics. We denote by $\uparrow_\circ^\sharp (x^\sharp)$ the approximation of an interval with real bounds by an interval with floating-point bounds. The bounds are rounded to the nearest to reflect the fact this first interval corresponds to the approximated values seen by the program.

$$\uparrow_\circ^\sharp [(\underline{x}, \overline{x})] = [\uparrow_\circ (\underline{x}), \uparrow_\circ (\overline{x})] \tag{7}$$

where $\uparrow_\circ (x)$ denotes the rounding of x in the IEEE754 Standard [1] rounding mode $\circ \in \{-\infty,\ +\infty,\ 0,\ \sim\}$.

Conversely, the function $\downarrow_\circ^\sharp$ abstracts the concrete function $\downarrow_\circ$ which computes the exact error $\downarrow_\circ (x) = x - \uparrow_\circ (x)$. That means that for all $x \in [\underline{x}, \overline{x}]$ we have $\downarrow_\circ (x) \in \downarrow_\circ^\sharp [(\underline{x}, \overline{x})]$. We have

$$\downarrow_\circ^\sharp [(\underline{x}, \overline{x})] = [-y, y] \quad \texttt{with} \quad y = \begin{cases} \frac{1}{2}\texttt{ulp}\big(max(|\underline{x}|, |\overline{x}|)\big) & \text{if} \quad \circ = \sim \\ \texttt{ulp}\big(max(|\underline{x}|, |\overline{x}|)\big) & \text{otherwise.} \end{cases} \tag{8}$$

Note that the *unit in the last place* $\mathrm{ulp}(x)$ is the weight of the least significant digit of the floating-point number x. A sample of the elementary operations over $E^\sharp$ are defined in equations (9) to (10), for other operations see [22].

$$(x_1^\sharp, \mu_1^\sharp) + (x_2^\sharp, \mu_2^\sharp) = (\uparrow_\circ^\sharp (x_1^\sharp + x_2^\sharp), \mu_1^\sharp + \mu_2^\sharp + \downarrow_\circ^\sharp (x_1^\sharp + x_2^\sharp)), \tag{9}$$

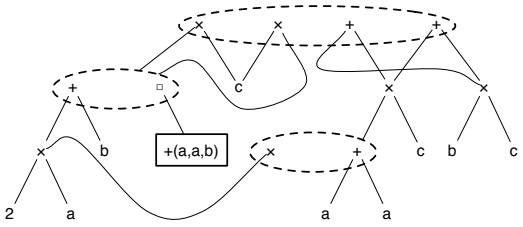


Fig. 5. APEG for the expression $e = \big((a + a) + b\big) \times c$.

$$(x_1^\sharp, \mu_1^\sharp) \times (x_2^\sharp, \mu_2, ^\sharp) = (\uparrow_\circ^\sharp (x_1^\sharp \times x_2^\sharp), x_2^\sharp \times \mu_1^\sharp + x_1^\sharp \times \mu_2^\sharp + \mu_1^\sharp \times \mu_2^\sharp + \downarrow_\circ^\sharp (x_1^\sharp \times x_2^\sharp)). \tag{10}$$

For example, if we add two numbers, the errors on the operands are added to the error due to the roundoff of the result. For the product, the semantic consists of the development of $(x_1^\sharp + \mu_1^\sharp) \times (x_2^\sharp + \mu_2^\sharp)$.

Note that more efficient abstract domains exist, e.g., [5, 14, 13] as well as complementary techniques [3, 4]. Let us also mention that other methods exist to transform, synthesize or repair arithmetic expressions in the integer or fixed arithmetic [12, 20].

3.2 Accuracy Improvement of Expressions

Here, we briefly present former work [16, 21, 24] to semantically transform arithmetic expressions using Abstract Program Expression Graph (APEG). This data structure remains 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 $1 \times 3 \times 5 ... \times (2n-3)$ possible formulas. In order to build large APEGs, two algorithms are used (propagation and expansion algorithms). 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 formulas represented in an APEG using the abstract semantics of Section 3.1.

Example 1. An example of APEG is given in Figure 5. When an equivalence class (denoted by a dotted ellipse) 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 $p_1 * \ldots * p_n$. For instance, the APEG p of Figure 5 represents all the following expressions:

$$\mathcal{A}(p) = \left\{ \begin{array}{c} \big((a + a) + b\big) \times c,\ \big((a + b) + a\big) \times c,\ \big((b + a) + a\big) \times c, \\ \big((2 \times a) + b\big) \times c,\ c \times \big((a + a) + b\big),\ c \times \big((a + b) + a\big), \\ c \times \big((b + a) + a\big),\ c \times \big((2 \times a) + b\big),\ (a + a) \times c + b \times c, \\ (2 \times a) \times c + b \times c,\ b \times c + (a + a) \times c,\ b \times c + (2 \times a) \times c \end{array} \right\}. \tag{11}$$

For this example, the last step of transformation would consist of evaluating all the expressions in $\mathcal{A}(p)$ with the abstract semantics of Section 3.1 in order to select the most accurate one. □

4 Transformation of Commands

In this section, we introduce the formal rules used to transform intra-procedural pieces of code. The syntax of commands is given in Equation (12). It corresponds to the core of an imperative language.

$$\text{Com} \quad \ni \quad c ::= id = e \mid c_1 \ ; \ c_2 \mid \mathsf{if}_{\Phi}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \mid \mathsf{while}_{\Phi}\ e\ \mathsf{do}\ c \mid \mathsf{nop}. \qquad (12)$$

The command language is made of assignments $id = e$, sequences of instructions, the void operation nop, a conditional statement $\mathsf{if}_{\Phi}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2$ and a loop statement $\mathsf{while}_{\Phi}\ b\ \mathsf{do}\ c$. Programs are assumed to be written in SSA form [9] and the Φ variables attached to conditional and while statements denote their sets of Φ nodes. The Φ node $\Phi(id, id_1, id_2)$ is understood as an assignment of 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 [2, 9].

The transformation defined by the rules of Figure 6 uses states of the form $\langle c, \delta, C, \nu, \beta \rangle$ where:

- c is a command, as defined in Equation (12),
- δ is an environment $\delta : \mathcal{V} \to \text{Expr}$ 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 $(A2)$,
- $C \in \text{Ctx}$ is a single hole context [15] defined in Equation (13). It records the program englobing the current expression to be transformed and which is intended to fit in the hole denoted by [].

$$\text{Ctx} \ni C ::= [] \mid id = e \mid C_1 \ ; C_2 \mid \mathsf{if}_{\Phi}\ e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \mid \mathsf{while}_{\Phi}\ e\ \mathsf{do}\ C \mid \mathsf{nop}. \quad (13)$$

- let $\nu \in \mathcal{V}$ denote the reference variable that we aim at optimizing.
- let $\beta \subseteq \mathcal{V}$ be a list of assigned variables that should not be removed from the source program. Initially, $\beta = \{\nu\}$, i.e., the target variable ν must not be removed. The set β is modified by rules $(C1), (C2), (C4)$ and $(W2)$.

Let us now describe the rules of Figure 6. 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. The function $Var(e)$ returns the set of variables occurring in the expression e while $Dom(\delta)$ denotes the domain of definition of δ. When using Rule $(A1)$, to get a semantically equivalent program, we must respect some restrictions. The first one requires that the variables occurring in e do not meet the domain of δ (otherwise we would break some data dependencies). Finally, Rule $(A1)$ requires that the transformation is done if the identifier id does not belong to the set β of variables which may not be removed.

$$\frac{\delta' = \delta[id \mapsto e] \quad Var(e) \cap Dom(\delta) = \emptyset \quad id \notin \beta}{\langle id = e, \delta, C, \nu, \beta\rangle \to \langle \mathsf{nop}, \delta', \beta\rangle} \quad (A1)$$

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

$$\frac{\langle c\ , \delta, C, \nu, \beta\rangle \to \langle c', \delta', \beta'\rangle}{\langle \mathsf{nop}\ ;\ c\ , \delta, C, \nu, \beta\rangle \to \langle c', \delta', \beta'\rangle} \quad (S1)$$

$$\frac{\langle c\ , \delta, C, \nu, \beta\rangle \to \langle c', \delta', \beta'\rangle}{\langle c\ ;\ \mathsf{nop}\ , \delta, C, \nu, \beta\rangle \to \langle c', \delta', \beta'\rangle} \quad (S2)$$

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

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

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

$$\frac{Var(e) \cap Dom(\delta) = \emptyset \quad \beta' = \beta \cup Assigned(c_1) \cup Assigned(c_2) \quad \langle c_1, \delta, C, \nu, \beta'\rangle \to^* \langle c_1', \delta_1, \beta_1\rangle \quad \langle c_2, \delta, C, \nu, \beta'\rangle \to^* \langle c_2', \delta_2, \beta_2\rangle \delta' = \delta_1 \cup \delta_2}{\langle \mathsf{if}_\Phi\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \delta, C, \nu, \beta\rangle \to \langle \mathsf{if}_\Phi\ e\ \mathsf{then}\ c_1'\ \mathsf{else}\ c_2', \delta', \beta'\rangle} \quad (C3)$$

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

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

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

Fig. 6. Transformation rules used to improve the accuracy of programs.

Rule ($A2$) offers an alternative way of processing assignments, when the conditions of Rule ($A1$) are not fulfilled. The action of substituting the variables of e by their definitions in δ is denoted by $\delta(e)$. 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 Section 3. The abstract environment $\sigma^\sharp : \mathcal{V} \to E^\sharp$ used for this transformation results from a static analysis using the domain $E^\sharp$ also introduced in Section 3. 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 Section 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, the expression e' is transformed into an ex-

pression e'' by $\langle e', \sigma^\sharp \rangle \rightsquigarrow e''$ which transforms the source expression into a more accurate one for the environment σ. In our implementation this corresponds to a call to the APEG tool [16, 17]. 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 formulas. In our implementation, in order to facilitate their manipulation, we slice these formulas 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.

Example 2. To explain the use of rules $(A1)$ and $(A2)$, let us consider the example of Equation (14) in which three variables x, y and z are assigned. In this example, ν consists of the variable z that we aim to optimize and $a = 0.1$, $b = 0.01$, $c = 0.001$ and $d = 0.0001$ are constants.

$$\begin{array}{ll} & \langle \mathtt{x} = \mathtt{a} + \mathtt{b}; \mathtt{y} = \mathtt{c} + \mathtt{d}; \mathtt{z} = \mathtt{x} + \mathtt{y},\ \delta, [], \nu, \emptyset \rangle \\ \underset{(A1)}{\longrightarrow} & \langle \mathsf{nop}; \mathtt{y} = \mathtt{c} + \mathtt{d}; \mathtt{z} = \mathtt{x} + \mathtt{y}, \delta' = \delta[\mathtt{x} \mapsto \mathtt{a} + \mathtt{b}], [], \nu, \emptyset \rangle \\ \underset{(A1)}{\longrightarrow} & \langle \mathsf{nop}; \mathsf{nop}; \mathtt{z} = \mathtt{x} + \mathtt{y}, \delta'' = \delta'[\mathtt{y} \mapsto \mathtt{c} + \mathtt{d}], [], \nu, \emptyset \rangle \\ \underset{(A2)}{\longrightarrow} & \langle \mathsf{nop}; \mathsf{nop}; \mathtt{z} = ((\mathtt{d} + \mathtt{c}) + \mathtt{b}) + \mathtt{a}, \delta'', [], \nu, \emptyset \rangle \end{array} \tag{14}$$

In Equation (14), 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. For the last step, we may not apply $(A1)$ to z because the condition is not satisfied $(z = \nu)$. 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, ν 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, the state $\langle c_1' \ ; \ c_2', \delta'', \beta'' \rangle$ is returned.

Rules $(C1)$ to $(C4)$ concern conditionals. The first two rules correspond to a partial evaluation of the program [18], when the test evaluates to true or false in the environment $\sigma^\sharp$ which is computed by static analysis, $\sigma^\sharp = [\![C[\mathsf{if}_\Phi\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]]\!]^\sharp \iota^\sharp$. In rules $(C1)$ and $(C2)$, the conditional is replaced by the branch c_1 or c_2. In this case, the reference variable ν does not appear necessarily in c_1 or c_2 but the variables assigned in these branches are used in the Φ nodes. Consequently, they may not be removed from c_1 or c_2 and we have to transform the command with $\beta' = \beta \cup \mathit{Assigned}(c_i)$, for $i = 1$ or 2. Here, $\mathit{Assigned}(c)$ denotes the set of identifiers assigned in the command c.

Example 3. Let us consider the program, in SSA form.

$$x_1 = 0;\ \mathsf{if}_{\Phi(x_3,x_1,x_2)}\ \mathit{cond}\ \mathsf{then}\ x_2 = a + b\ \mathsf{else}\ y_1 = c + d; \nu = x_3. \tag{15}$$

Depending on the value of the test, we transform this program into

$$\begin{cases} \nu = a + b & \text{if } cond, \\ \nu = 0 & \text{if } \neg cond. \end{cases} \tag{16}$$

However, when *cond* is *true*, without the blacklist, Rule $(A1)$ would store x_2 in δ during the transformation of the branch. The Φ-node $\Phi(x_3, x_1, x_2)$ would be wrong. □

Rule $(C3)$ is the general rule for conditionals. The then and else branches are transformed, assuming that the variables of the condition do not meet the variables of δ. As for rules $(C1)$ and $(C2)$, 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 (note that thanks to the SSA form, the variables assigned in both branches are distinct). Finally, Rule $(C4)$ is used when the conditions for Rule $(C3)$ do not hold. In this case, $Var(e) \cap Dom(\delta) \neq \emptyset$ and we need to reinsert the common variables into the source code. Let $Var(e)$ be the list of variables occuring in the expression e. Firstly, a new command c' corresponding to sequences of assignments of the form $id = \delta(id)$ is built for all the variables $id \in Var(e)$ by $AddDefs(V, \delta)$ and, 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 4. Let us take another example to explain the Rules $(C3)$ and $(C4)$.

$$x_1 = 0;\ \mathsf{if}_{\Phi(y_3,y_1,y_2)}\ x_1 > 1\ \mathsf{then}\ y_1 = x_1 + 2;\ \mathsf{else}\ y_2 = x_1 - 1; \nu = y_3. \tag{17}$$

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

$$\mathsf{if}_{\Phi(y_3,y_1,y_2)}\ x_1 > 1\ \mathsf{then}\ y_1 = x_1 + 2;\ \mathsf{else}\ y_2 = x_1 - 1; \nu = y_3. \tag{18}$$

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 reinject 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 last two rules $(W1)$ and $(W2)$ are for the while statements. 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. 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 Figure 6 can be selected. Consequently, the transformation would be linear in the size n of the program if we would not reinject 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 the transformation of a program of size n is $\mathcal{O}(n^3)$.

5 Experimental Results

In this section, we evaluate the efficiency of the transformation presented in Section 4 through a series of experiments using our prototype. We have chosen several algorithms coming from various application fields (avionics, chemistry, mathematics, etc.) In each case, we compare the numerical accuracy of the sample program with the accuracy of the generated code. The upper bounds on the rounding errors are computed as in Section 3.1. We optimize the value of the reference variable, named ν in Section 4. The original and the transformed codes are shown in Figure 9 and their accuracy is given in Figure 8. This transformation is achieved almost instantaneously (less than one second) on a standard laptop (Intel Core i5 with 4 Go memory).

5.1 Control Algorithms

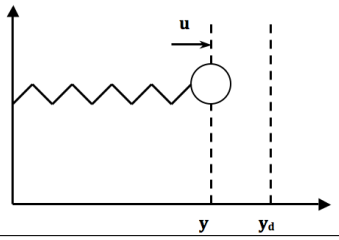
In this section, we consider three classical algorithms from control theory, namely a PID Controller, Lead-Lag Compensator and the running example of Odometry.

PID. The PID Controller [6] 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. We have $e = c - m$,

$$p = k_p \times e, \quad i = i + k_i \times e \times dt \quad \text{and} \quad d = k_d \times (e - e_{old}) \times \frac{1}{dt}.$$

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

Lead-Lag System. A second test has been performed on a dynamical system illustrated in Figure 7. This system includes a single mass and a single spring and is governed by an automatically synthesized controller [11] which tries to move the mass from the initial position y to the desired one y_d. 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].



$yc = max(min(y - yd, 1), -1);$
$u = Cc * xc + Dc * yc;$
$xc = Ac * xc + Bc * yc;$
$receive(y, 2);$
$receive(yd, 3);$

Fig. 7. Left: The lead-lag system of section 5.1. Right: Parameters of the system.

5.2 Numerical Algorithms

Runge-Kutta Methods. This example concerns Runge-Kutta methods [19]. 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

$$k_1 = (\frac{dy}{dx}) = h \times f(x_i, y_i) \quad \text{and} \quad k_2 = (\frac{dy}{dx}) = h \times f(x_i + \frac{h}{2}, y_i + \frac{h}{2}). \tag{19}$$

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:

$$y_{i+1} = y_1 + \tfrac{1}{6}\left[k_1 + 2 \times k_2 + 2 \times k_3 + k_4\right] \times h. \tag{20}$$

The Trapezoidal Rule. This example concerns with an algorithm for the trapezoidal rule [19], well known in numerical analysis to approximate the definite integral $\int_a^b f(x)\,dx$. 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:

$$g(x) = \frac{u}{0.7x^3 - 0.6x^2 + 0.9x - 0.2}. \tag{21}$$

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

Fig. 8. Initial and new errors on the examples programs of Section 5.

We assume that u is a user defined parameter in the range $[1.11, 2.22]$. In addition, we have unfold the body of the loop twice to obtain better results with our prototype.

Code	Source Code	Optimized Code

PID
$\nu = m$
	m = [4.5,9.0]; ki = 0.69006; kp = 9.4514;
	kd = 2.8454; t = 0.0; i = 0.0; c = 5.0;
	dt = 0.2; invdt = 5.0; eold = 0.0;
	while (t < 20.0) 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 }
		m = [4.5,9.0]; t = 0.0; eold = 0.0;
		i = 0.0;
		while (t < 20.0) 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 }

Lead-Lag
$\nu = \mathrm{xc}_1$
	y = [2.1,17.9] ; xc0 = 0.0 ; xc1 = 0.0
	; t = 0.0 ; yd = 5.0; Ac00 = 0.499;
	Ac01 = -0.05; Ac10 = 0.01; Ac11 = 1.0;
	Bc0 = 1.0; Bc1 = 0.0; Cc0 = 564.48;
	Cc1 = 0.0; Dc = -1280.0;
	while (t < 5.0) do {
	yc = (y - yd) ;
	if (yc < -1.0) then {yc = -1.0} ;
	if (1.0 < yc) then {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) }
		y = [2.1,17.9]; t = 0.0; xc1 = 0.0;
		xc0 = 0.0;
		while (t < 5.0) do {
		yc = (-5.0+y) ;
		if (yc < -1.0) then {yc = -1.0} ;
		if (1.0< yc) then {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) }

Fig. 9. Original and optimized codes for the examples of Section 5.1.

5.3 Results

Our prototype consists of an implementation of the rules described in Section 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. We compare then the initial error and the new error of each program before and after transformation. Figures 9 and 10 show the source and target program 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 Figure 8, where

Code	Source Code	Optimized Code

RK4
$\nu = y_{n+1}$
	yn = [-10.1,10.1]; t = 0.0; k = 1.2;
	c = 100.1; h = 0.1;
	while (t < 1.) 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) }
		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) }

Trapeze
$\nu = r$
	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.) then { 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) }
		u = [1.11, 2.22] xa = 0.25; r = 0.0;
		while (xa < 5000.) 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)
		}

Fig. 10. Original and optimized codes for the examples of Section 5.2.

s is the slice size, i.e., the parameter defining at which height of the syntactic tree we cut the expressions.

6 Conclusion

In our search for automatic transformation of programs, we have developed a tool which rewrites codes 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 Figure 6, correctness conditions have been defined to guarantee that the dependencies are respected and to ensure the correctness of the rewritings in conditions and loops. In order to validate our tool, we have chosen a set of representative programs taken from various fields of

science and engineering. We have automatically tuned them and analyzed their accuracy before and after transformation.

The further research directions consists of generalizing our techniques to other kinds of programming patterns like for loops, arrays and, specially functions in order to obtain an intra-procedural program transformation with function refactoring and specialization with respect to the values of arguments. Another extension looks 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 done. Finally, our transformation relies on a static analysis of the source codes. Indeed, we select the optimized program by using the abstract semantics in Section 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.

References

1. ANSI/IEEE. *IEEE Standard for Binary Floating-Point Arithmetic*. SIAM, 2008.
2. A-W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.
3. E-T. Barr, T. Vo, V. Le, and Z. Su. Automatic detection of floating-point exceptions. In *Symposium on Principles of Programming Languages, POPL '13, 2013*, pages 549–560. ACM, 2013.
4. F. Benz, A. Hildebrandt, and S. Hack. A dynamic program analysis to find floating-point accuracy problems. In *Programming Language Design and Implementation, PLDI '12, 2012*, pages 453–462. ACM, 2012.
5. J. Bertrane, P. Cousot, R. Cousot, F. Feret, L. Mauborgne, A. Miné, and X. Rival. Static analysis by abstract interpretation of embedded critical software. *ACM SIGSOFT Software Engineering Notes*, 36(1):1–8, 2011.
6. A. Chapoutot, N. Damouche, and M. Martel. Automatic transformation of a PID controller. In *International Workshop on Numerical Software Verification*, 2014.
7. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages*, pages 238–252, 1977.
8. P. Cousout and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *Principles of Programming Languages*, pages 178–190. ACM, 2002.
9. R. Cytron and R. Gershbein. Efficient accomodation of may-alias information in SSA form. In *Programming Language Design and Implementation (PLDI)*, pages 36–45. ACM, 1993.
10. D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, and V. Védrine. Towards an industrial use of FLUCTUAT on safety-critical avionics software. In *Formal Methods for Industrial Critical Systems*, volume 5825 of *Lecture Notes in Computer Science*, pages 53–69. Springer, 2009.
11. E. Feron. From control systems to control software. *IEEE Control Systems Magazine*, 30(6):50–71, 2010.

12. X. Gao, S. Bayliss, and G-A. Constantinides. SOAP: structural optimization of arithmetic expressions for high-level synthesis. In *Field-Programmable Technology, FPT*, pages 112–119. IEEE, 2013.
13. E. Goubault. Static analysis by abstract interpretation of numerical programs and systems, and FLUCTUAT. In *Static Analysis Symposium, SAS*, volume 7935 of *Lecture Notes in Computer Science*, pages 1–3. Springer, 2013.
14. E. Goubault and S. Putot. Static analysis of finite precision computations. In *Verification, Model Checking, and Abstract Interpretation*, volume 6538 of *LNCS*. Springer, 2011.
15. E. Hankin. *Lambda Calculi A Guide For Computer Scientists*. Clarendon Press, Oxford, 1994.
16. A. Ioualalen and M. Martel. A new abstract domain for the representation of mathematically equivalent expressions. In *Static Analysis Symposium*, pages 75–93, 2012.
17. A. Ioualalen and M. Martel. Synthesizing accurate floating-point formulas. In *Application-Specific Systems, Architectures and Processors, ASAP*, pages 113–116, 2013.
18. N-D. Jones. An introduction to partial evaluation. *ACM Computing Surveys*, 28(3):480–503, 1996.
19. A. Kendall. *An Introduction to Numerical Analysis*. John Wiley & Sons, 1989.
20. F. Logozzo and T. Ball. Modular and verified automatic program repair. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*, pages 133–146. ACM, 2012.
21. M. Martel. Accurate evaluation of arithmetic expressions (invited talk). *Electr. Notes Theor. Comput. Sci.*, 287:3–16, 2012.
22. Matthieu Martel. Semantics of roundoff error propagation in finite precision calculations. *Higher-Order and Symbolic Computation*, 19(1):7–30, 2006.
23. J-M. Muller, N. Brisebarre, F. De Dinechin, C-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser, 2010.
24. R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: A new approach to optimization. *Logical Methods in Computer Science*, 7(1), 2011.