Numerical Accuracy Improvement by Interprocedural Program Transformation

Nasrine DAMOUCHE
LAMPS Laboratory
Université de Perpignan
nasrine.damouche@univ-perp.fr

Matthieu MARTEL
LAMPS Laboratory
Université de Perpignan
matthieu.martel@univ-perp.fr

Alexandre CHAPOUTOT
U2IS, ENSTA ParisTech,
Université de Paris-Saclay
chapoutot@ensta.fr

ABSTRACT

Floating-point numbers are used to approximate the exact real numbers in a wide range of domains like numerical simulations, embedded software, etc. However, floating-point numbers are a finite approximation of real numbers. In practice, this approximation may introduce round-off errors and this can lead to catastrophic results. To cope with this issue, we have developed a tool which corrects partly these round-off errors and which consequently improves the numerical accuracy of computations by automatically transforming programs in a source to source manner. Our transformation, relies on static analysis by abstract interpretation and operates on pieces of code with assignments, conditionals and loops. In former work, we have focused on the intraprocedural transformation of programs and, in this article, we introduce the interprocedural transformation to improve accuracy.

CCS CONCEPTS

•**General and reference** → **Verification;** *Formal methods;* Compilers; Embedded systems; •**Software and its engineering** → Software verification and validation;

KEYWORDS

Interprocedural program transformation, Numerical accuracy, static analysis, Floating-point arithmetic

ACM Reference format:
Nasrine DAMOUCHE, Matthieu MARTEL, and Alexandre CHAPOUTOT. 2016. Numerical Accuracy Improvement by Interprocedural Program Transformation. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17),* 11 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Floating-point numbers, whose specification is given by the IEEE754 Standard [1, 21], are more and more used in many industrial applications, including critical embedded software. However, floating-point arithmetic is prone to accuracy problems caused by the round-off errors. The approximation becomes dangerous when accumulated errors cause damages whose gravity varies depending on the context of the application. We correct partly these errors by automatically transforming programs in a source to source manner. Our method not only transforms arithmetic expressions [18, 22] but also pieces of code containing assignments, conditionals, loops, etc.Basically, we generate large arithmetic expressions corresponding to the computations of the original program and further, we consider many expressions mathematically equivalent to the original ones in order to, finally, choose a more accurate one in polynomial time [7, 9].

There exist several methods for validating [3, 11, 12, 15, 24] and improving [22] the accuracy of arithmetic expressions in order to avoid numerical failures. In this article, as in our previous work, we rely on static analysis by abstract interpretation [4] to compute variable ranges and round-off error bounds. We use a set of transformation rules for arithmetic expressions and commands [7]. These rules, which are applied in a deterministic order, allow one to obtain a more accurate code among all the codes which are considered. We have shown in previous work [7] that the numerical accuracy of programs is significantly improved by our method, in most cases, the worst error on the result, for all the considered inputs is decreased of about 20%.

Since large codes necessarily contain several functions and procedures, our transformation tool has to use interprocedural techniques [23] to adequately support such programs in order to make them more accurate. The main contribution of this article is to generalize our automatic transformation techniques to handle interprocedural program transformations [23]. First, we define three rules to optimize the accuracy of the computations of programs: *inlining of functions*, *function specialization* and *lazy evaluation of the arguments*. Second, we present a heuristic which helps to select the best transformation rule for each function.

This article is organized as follows. We introduce in Section 2 the floating-point arithmetic, how to compute the error bounds and we explain how to transform arithmetic expressions and commands. In Section 3, we discuss the different interprocedural transformation rules. In addition, we introduce how to choose the interprocedural transformation rules that will be applied on the program to improve the numerical accuracy of computations. Section 4 describes the experimental results. Finally, we give some concluding remarks and perspectives in Section 5.

Conference'17, Washington, DC, USA

DOI: 10.1145/nnnnnnn.nnnnnnn

2 BACKGROUND

In this section, we give some background concerning the techniques used to achieve our program transformation. Section 2.1 briefly describes the floating-point arithmetic. Section 2.2 introduces how to compute the errors introduced by floating-point computations with respect to exact computations. Sections 2.3 and 2.5 respectively present the transformation techniques used to optimize arithmetic expressions [18] and programs [7]. These transformations are implemented in the tool that we use to optimize programs.

2.1 Floating-Point Arithmetic

Floating-point numbers are used to represent real numbers. Because of their finite representation, round-off errors arise during the computations which may cause damages in critical contexts. The IEEE754 Standard formalizes a binary floating-point number as a triplet of sign, mantissa and exponent. We consider that a number x is written:

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

where, s is the sign $\in \{-1, 1\}$, b is the basis, $b = 2$, m is the mantissa, $m = d_0.d_1 \dots d_{p-1}$ with digits $0 \leq d_i < b$, $0 \leq i \leq p-1$, p is the precision and e is the exponent $e \in [e_{min}, e_{max}]$.

A floating-point number x is *normalized* whenever $d_0 \neq 0$. Normalization avoids multiple representations of the same number. IEEE754 Standard specifies some particular values for p, e_{min} and e_{max}, as well as *denormalized numbers* which are floating-point numbers with $d_0 = d_1 = \ldots = d_k = 0$, $k < p-1$ and $e = e_{min}$. Denormalized numbers make underflow gradual [14].

The IEEE754 Standard defines four rounding modes for elementary operations over floating-point numbers. These modes are towards $-\infty$, towards $+\infty$, towards zero and to the nearest respectively denoted by $\uparrow_{+\infty}$, $\uparrow_{-\infty}$, $\uparrow_0$ and $\uparrow_\sim$. The semantics of the elementary operations specified by the IEEE754 Standard is given by Equation (2).

$$x \circledast_r y = \uparrow_r (x * y) \ , \text{ with } \uparrow_r : \mathbb{R} \to \mathbb{F} \qquad (2)$$

where a floating-point operation, denoted by $\circledast_r$, is computed using the rounding mode r and $* \in \{+, -, \times, \div\}$ an exact operation. Obviously, the results of the computations are not exact because of the round-off errors. This is why, we use also the function $\downarrow_r : \mathbb{R} \to \mathbb{R}$ that returns the round-off error. We have $\downarrow_r (x) = x - \uparrow_r (x)$.

2.2 Error Bound Computation

In order to compute the errors during the evaluation of arithmetic expressions [20], we use values which are pairs $(x, \mu) \in \mathbb{F} \times \mathbb{R} \equiv \mathbb{E}$ where x is the floating-point number used by the machine and μ is the exact error attached to $\mathbb{F}$, *i.e.*, the exact difference between the real and floating-point numbers as defined in Section 2.1. For example, the real number $\frac{1}{3}$ is represented by the value $v = (\uparrow_\sim \left(\frac{1}{3}\right), \downarrow_\sim \left(\frac{1}{3}\right)) = (0.333333, (\frac{1}{3} - 0.333333))$. The semantics of the elementary operations on $\mathbb{E}$ is defined in [20].

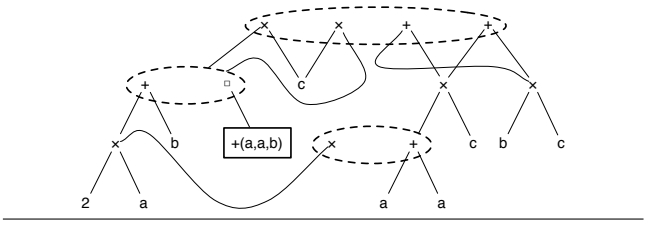

Figure 1: APEG for the expression $e = \big((a+a)+b\big) \times c$.

Our tool uses an abstract semantics [4] 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 now introduce 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 Equation (3).

$$\uparrow^\sharp_\sim ([\underline{x}, \overline{x}]) = [\uparrow_\sim (\underline{x}), \uparrow_\sim (\overline{x})] \ . \qquad (3)$$

We denote by $\downarrow^\sharp$ the function that abstracts the concrete function $\downarrow_\sim$. It over-approximates the set of exact values of the error $\downarrow_\sim (x) = x - \uparrow_\sim (x)$. 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

$$\downarrow^\sharp ([\underline{x}, \overline{x}]) = [-y, y] \quad \text{with} \quad y = \frac{1}{2}\text{ulp}\big(\max(|\underline{x}|, |\overline{x}|)\big) \ . \qquad (4)$$

Formally, the *unit in the last place*, denoted by ulp(x), consists of the weight of the least significant digit of the floating-point number x. Equations (5) and (6) give the semantics of the addition and multiplication over $\mathbb{E}^\sharp$, for other operations see [20]. If we sum two numbers, we must add errors on the operands to the error produced by the round-off of the result. When multiplying two numbers, the semantics is given by the development of $(x_1^\sharp + \mu_1^\sharp) \times (x_2^\sharp + \mu_2^\sharp)$.

$$(x_1^\sharp, \mu_1^\sharp) + (x_2^\sharp, \mu_2^\sharp) = \big(\uparrow^\sharp (x_1^\sharp + x_2^\sharp), \mu_1^\sharp + \mu_2^\sharp + \downarrow^\sharp (x_1^\sharp + x_2^\sharp)\big) \ , \qquad (5)$$

$$\begin{aligned}(x_1^\sharp, \mu_1^\sharp) \times (x_2^\sharp, \mu_2^\sharp) = \big(\uparrow^\sharp (x_1^\sharp \times x_2^\sharp), x_2^\sharp \times \mu_1^\sharp \\ + x_1^\sharp \times \mu_2^\sharp + \mu_1^\sharp \times \mu_2^\sharp + \downarrow^\sharp (x_1^\sharp \times x_2^\sharp)\big) \ .\end{aligned} \qquad (6)$$

2.3 Accuracy Improvement of Expressions

We consider 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 the operators $+$, $-$, $\times$ and $\div$. The syntax is

$$\text{Expr} \quad \ni \quad e \quad ::= \quad id \mid cst \mid e + e \mid e - e \mid e \times e \mid e \div e. \qquad (7)$$

Here, we briefly present former work [18, 25] to semantically transform [5] arithmetic expressions using Abstract Program Expression Graph (APEG). This data structure remains in polynomial size while dealing with an exponential number of equivalent expressions.

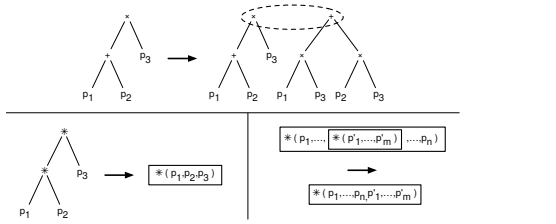

Figure 2: APEG construction by pattern matching.

An APEG is defined inductively as follows: (1) A value v or a variable x 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, (3) A box $\boxed{*(p_1, \ldots, p_n)}$ is an APEG, where $*$ is a commutative and associative operator and the $p_{i,1\leq i\leq n}$, are APEGs and (4) A non-empty set $\{p_1, \ldots, p_n\}$, called equivalence class, of APEGs is an APEG where $p_{i,1\leq i\leq n}$, is not a set of APEGs itself.

An example of APEG is given in Figure 1. When an equivalence class (denoted by a dotted ellipse in Figure 1) 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 Figure 1, the leafs corresponding to the variables a, b and c are duplicated while, in practice, they are defined only once in the structure. The set $\mathcal{A}(p)$ of expressions contained inside an APEG p is defined inductively:

- If p is a value v or a variable x then $\mathcal{A}(p) = \{v\}$ or $\mathcal{A}(p) = \{x\}$.
- If p is an expression $p_1 * p_2$ then
$$\mathcal{A}(p) = \bigcup_{e_1 \in \mathcal{A}(p_1),\ e_2 \in \mathcal{A}(p_2)} e_1 * e_2$$
- If p is a box $\boxed{*(p_1, \ldots, p_n)}$ then $\mathcal{A}(p)$ contains all the parsings of $e_1 * \ldots * e_n$, $\forall e_1 \in \mathcal{A}(p_1), \ldots, e_n \in \mathcal{A}(p_n)$.
- If p is an equivalence class $\{p_1, \ldots, p_n\}$ then $\mathcal{A}(p) = \bigcup_{1\leq i\leq n} \mathcal{A}(p_i)$.

EXAMPLE 2.1. *For instance, the APEG p of Figure 1 represents all the following expressions:*

$$\mathcal{A}(p) = \left\{ \begin{array}{ll} ((a+a)+b)\times c\ , & ((a+b)+a)\times c\ , \\ ((b+a)+a)\times c\ , & ((2\times a)+b)\times c\ , \\ c\times((a+a)+b)\ , & c\times((a+b)+a)\ , \\ c\times((b+a)+a)\ , & c\times((2\times a)+b)\ , \\ (a+a)\times c+b\times c\ , & (2\times a)\times c+b\times c\ , \\ b\times c+(a+a)\times c\ , & b\times c+(2\times a)\times c \end{array} \right\} \quad (8)$$

■

In their article on EPEGs, R. Tate *et al.* use rewriting rules to extend the structure up to saturation [25]. 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 Figure 2. An alternative technique for APEG construction is to use dedicated algorithms. Such algorithms, working in polynomial time, have been proposed in [18].

Once some APEG is built, we use an heuristic to find in polynomial time a more accurate expression which minimize the abstract error $\mu^\sharp$ using the semantics of equations (5) and (6). We say that the abstract error $\mu_1^\sharp = [\underline{\mu_1^\sharp}, \overline{\mu_1^\sharp}]$ is less than the abstract error $\mu_2^\sharp = [\underline{\mu_2^\sharp}, \overline{\mu_2^\sharp}]$ if

$$\max(|\underline{\mu_1^\sharp}|, |\overline{\mu_1^\sharp}|) \leq \max(|\underline{\mu_2^\sharp}|, |\overline{\mu_2^\sharp}|)\ . \quad (9)$$

2.4 Related Work

During the last ten years, several static analyses of the numerical accuracy of floating-point computations have been introduced. While these methods compute an over-approximation of the worst error arising during the executions of a program, they operate on final codes, during the verification phase and not at implementation time. Static analyses based on abstract interpretation [4, 5] have been proposed and implemented in the `Fluctuat` tool [15, 16] which has been used in several industrial contexts. A main advantage of this method is that it enables one to bound safely all the errors arising during a computation, for large ranges of inputs. It also provides hints on the sources of errors, that is on the operations which introduce the most important precision loss. This latter information is of great interest to improve the accuracy of the implementation. Darulova and Kuncak have proposed a tool, `Rosa`, which uses a static analysis coupled to a SMT solver to compute the propagation of errors [11].

Another related research axis concerns the compile-time optimization of programs to improve the accuracy of the floating-point computation in function of given ranges for the inputs, without modifying the formats of the numbers [13]. The `Sardana` tool takes arithmetic expressions and optimize them using a source-to-source transformation. Herbie optimizes the arithmetic expressions of Scala codes. While `Sardana` uses a static analysis to select the best expression, Herbie uses dynamic analysis (a set or random runs). A comparison of these tools is given in [10].

2.5 Intraprocedural Transformation

In this section, we focus on the transformation of commands which is done using a set of rewriting rules formally defined in [7, 9]. Our language is made of assignments, conditionals, loops and sequences of commands. The syntax is

$$c ::= id = e \mid c_1\ ;\ c_2 \mid \mathsf{if}\ e\ c_1\ c_2 \mid \mathsf{while}_\Phi\ e\ \mathsf{do}\ c \mid \mathsf{nop}. \quad (10)$$

The transformation relies on several hypotheses. First of all, programs need to be in static single assignment form (SSA form) [6]. Our tool automatically puts input programs in SSA form before the transformation of commands for numerical accuracy. The principle of this intermediary representation is that every variable may be assigned only once in the

source code and must be assigned before its use. The next assumption is that we optimize a reference variable (the observed output of the program) chosen by the user. Our transformation is defined by rules using states $\langle c, \delta, C, \beta\rangle$ where

- c is a command, as defined in Equation (10),
- δ is an environment $\delta : \mathcal{V} \to$ Expr which maps variables to expressions. Intuitively, this environment records the expressions assigned to variables in order to inline them later on in larger expressions,
- $C \in$ Ctx is a single hole context [17]. It records the program englobing the current expression to be transformed,
- $\beta \subseteq \mathcal{V}$ is a list of assigned variables that should not be removed from the code. Initially, $\beta = \{\nu\}$, *i.e.*, the target variable ν must not be removed.

The environment δ is used to discard assignments from programs and to re-insert the expressions when the variables are read, in order to build larger expressions. In addition, in the following, $\nu \in \mathcal{V}$ denotes the reference variable.

Let us consider first assignments. If (i) the variable v of some assignment $v = e$ does not exist in the domain of δ, if (ii) $v \notin \beta$ and if (iii) $v \neq \nu$ then we memorize e in δ and we remove the assignment from the program. Otherwise, if one of the conditions (i), (ii) or (iii) is not satisfied then we rewrite this assignment by inlining the variables saved in δ in the concerned expression and we call the tool based on APEGs (see Section 2.3) to transform the resulting large expression. Note that, when transforming programs by inlining expressions in variables, we get larger formulas. The basic idea, in our implementation, when dealing with too large expressions, is to create intermediary variables and to assign to them the sub-expressions obtained by slicing the global expression at a given level of the syntactic tree. The last step consists of re-inserting these intermediary variables into the main program.

Example 2.2. *For example, let us consider the program below in which three variables x, y and z are assigned. We assume that z is the variable that we aim at optimizing and $a = 0.1$, $b = 0.01$, $c = 0.001$ and $d = 0.0001$ are constants.*

$$\begin{array}{ll} & \langle \mathtt{x = a + b\ ;\ y = c + d\ ;\ z = x + y}\ ,\ \delta,\ []\rangle \\ \to_z & \langle \mathtt{nop\ ;\ y = c + d\ ;\ z = x + y}, \delta' = \delta[\mathtt{x} \mapsto \mathtt{a + b}],\ []\rangle \\ \to_z & \langle \mathtt{nop\ ;\ nop\ ;\ z = x + y}, \delta'' = \delta'[\mathtt{y} \mapsto \mathtt{c + d}],\ []\rangle \\ \to_z & \langle \mathtt{nop\ ;\ nop\ ;\ z = ((d + c) + b) + a},\ \delta'',\ []\rangle \end{array} \quad (11)$$

In Equation (11), the environment δ and the context C are initially empty and the list β contains the reference variable `z`*. We 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 on the variable y. For the last step, we may not remove z because it is the reference variable. Instead, we substitute, in z, x and y by their values in δ and we transform the expression using the technique described in Section 2.3.* ■

Our tool also transforms conditionals. If a certain condition is always true or false, then we keep only the right branch, otherwise, we transform both branches of the conditional. When it is necessary, we re-inject variables that have been discarded from the main program.

Example 2.3. *Let us take another example to explain how we transform conditionals.*

$x_1 = a;$
if$_{\Phi(y_3,y_1,y_2)}$ $x_1 > 1.0$ *then*
  $y_1 = x_1 + 2.0 + 1.0;$
else
  $y_2 = x_1 - 2.0 - 1.0;$
  $\nu = y_3;$

if$_{\Phi(y_3,y_1,y_2)}$ $x_1 > 1.0$ *then*
  $y_1 = x_1 + 2.0 + 1.0;$
else
  $y_2 = x_1 - 2.0 - 1.0;$
  $\nu = y_3;$

Initially, the formal environment δ and the context C are empty and the black list β contains the target variable, ν. The first step consists of avoiding the assignment $x_1 = a$ from the program and storing it in δ. Then, we transform recursively the new program. This program is semantically incorrect since the test is undefined. So we re-inject the statement $x_1 = a$ in the program and add x_1 to the list β in order to avoid an infinite loop in the transformation. Note that at this step, the new environment δ' is $\delta[x_1 \mapsto a]$, β contains the variable x_1 and $C = x_1 = a; []$. In this case, we add the re-injected variable into the black list β to avoid its deletion from the program in the future by applying the corresponding rule. Now, δ is empty and β contains x_1 and ν. Finally, we transform both y_1 and y_2 by applying the partial evaluation techniques and we return y_3 that corresponds to y_1 or y_2 depending on the executed branch. At this stage, we have that C is $x_1 = a$; $\mathsf{if}_\Phi\ x_1 > 3$ then [] else []. ■

For a sequence $c_1;\ c_2$, 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, we return $\langle c_1'\ ;\ c_2', \delta'', \beta''\rangle$.

Other transformations have been defined for while loops. A first rule makes it possible to transform the body of the loop assuming that the variables of the condition have not been stored in δ. In this case, the body is optimized in the context $C[\mathsf{while}_\Phi\ e\ \mathsf{do}\ []]$ where C is the context of the loop. A second rule 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 that must be re-inserted. Secondly, the variables of V are removed from the domain of δ and added to β. The resulting command is obtained by transforming $c'; \mathsf{while}_\Phi\ e\ \mathsf{do}\ c$ with δ' and $\beta \cup V$.

Remark. *Note that, our tool takes as input ranges for the free variables which are either introduced by the user or coming from embedded sensors that transform physical measurements into digital data values. In the rest of this article, these inputs are assigned to global variables thanks to the primitive* `assert` *at the beginning of each program.*

$$\frac{f(u)\{c; return\ v\} \in \texttt{P}}{\langle z = f(e), \delta, C, \beta\rangle \rightarrow_\vartheta \langle u = e; c; z = v, \delta, C, \beta\rangle} \quad (F1 - inlining)$$

$$\frac{\gamma = [\![e]\!]^\sharp\sigma^\sharp \qquad \sigma^\sharp = [\![C[c]]\!]^\sharp\iota^\sharp \qquad g()\{c'; return\ v\} \quad f(u)\{c; return\ v\} \in \texttt{P} \qquad \langle c, \delta[u \mapsto \gamma], [], \{v\}\rangle \rightarrow_v^* \langle c', \delta, C, \beta\rangle}{\langle z = f(e), \delta, C, \beta\rangle \rightarrow_\vartheta \langle z = g(), \delta, C, \beta\rangle} \quad (F2 - specialization)$$

$$\frac{f(u)\{c; return\ v\} \in \texttt{P} \qquad \langle c, \delta[u \mapsto e], [], \{v\}\rangle \rightarrow_z^* \langle c', \delta, [], \{v\}\rangle \qquad g(V)\{c'; return\ v\} \qquad V = Var(e)}{\langle z = f(e), \delta, C, \beta\rangle \rightarrow_\vartheta \langle z = g(V), \delta, C, \beta\rangle} \quad (F3 - laziness)$$

Figure 3: Transformation rules used to deal with functions.

3 FUNCTION TRANSFORMATION

For our function transformation, we start by applying the interprocedural transformation rules given in Figure 3 and then the intraprocedural transformation rules discussed in Section 2.5. The general syntax of a program is

$$\begin{aligned} \mathsf{Prog} \quad \ni \quad \texttt{P} \ &::= \ \texttt{f} \mid \texttt{P f} \\ \texttt{f} \quad &::= \ f(u) \ \{ \ c; \ \ return\ v \ \} \ . \end{aligned} \quad (12)$$

In Equation (12), $f(u)\ \{\ c;\ return\ v\ \}$ defines a function whose argument is u, whose body is the command c and which returns v. Recall that a command c is defined by Equation (10) and a program `P` is a sequence of functions. For the sake of simplicity, in our formal definitions, we only consider functions with one single argument. However, in our implementation we support functions with many arguments. The generalization is straightforward.

For a sake of simplicity, we write $f(u)\ \{c;\ return\ v\}$ instead of $\mathsf{Type}\ f(\mathsf{Type}\ u)\ \{\ c;\ return\ v\}$ in Equation (12) and Figure 3, with $\mathsf{Type} \in \{Float, Double, \dots\}$. In addition, the grammar of arithmetic expressions is extended to function calls. Equation (7) becomes

$$\text{Expr} \quad \ni \quad e \quad ::= \quad id \mid cst \mid e{+}e \mid e{-}e \mid e{\times}e \mid e{\div}e \mid f(e). \quad (13)$$

We assume that any program p has a function named *main* which is the first function called when executing p, and the returned variable v is the target variable, i.e. $v = \nu$.

Basically, our interprocedural program transformation follows the same objective as the intraprocedural one. We aim at creating large arithmetic expressions which can be recombined into more accurate ones as explained in Section 2.3. The more the expressions are the more opportunities we have to rewrite them. In the case of functions, we may either inline the body of a function into the caller or evaluate lazily the arguments, especially when they correspond to large expressions. We also use a specialization rule with respect to the arguments of the function since this also improves the accuracy in many contexts. In this section, we detail these three transformation rules formally given in Figure 3.

3.1 Inlining Functions

The first rule $(F1)$ of Figure 3 consists of inlining the body of the function into the calling function. This makes possible to create larger expressions in the caller. Then the new program can be more optimized by applying the intraprocedural transformation rules previously seen in Section 2.5.

Rule $(F1)$ is used for a call $z = f(e)$ in the body of the calling function. We assume that `f` has one formal parameter `u`, a body `c` and returns a value `v`. Rule $(F1)$ states that the body of the calling function is transformed as follows. A new assignment is inserted in order to relate the formal parameter `u` to the expression `e` corresponding to the expression of argument. Then the body `c` of the called function `f` is inserted in the code of the `caller`, followed by a last assignment $z = v$ in order to assign the result of the function to the relevant variable `z`. Note that, if `f` is a function of more than one argument, then we process similarly by creating a sequence $u_1 = e_1, \dots, u_n = e_n$ of assignments before `c`.

EXAMPLE 3.1. *Let us consider the example of Figure 4 to illustrate how Rule $(F1)$ is applied. The original program contains a call to a function `callee()`. The principle here is to inline the body of the function `callee()` within the `caller()` function in the source code. In the code of Figure 4, we present the original program, the new program after applying the interprocedural rules and then by using the intraprocedural optimization, we obtain the final improved program.* ■

3.2 Specialization of Functions

The second transformation rule $(F2)$ is mainly used when we deal with a small number of calls to a large function in the original program. The idea is to pass the values of the function when the variability of the interval is small (for example whenever it contains less than ω floating-point numbers). By variability, we mean that the distance between the lower bound and the upper bound of an interval is small. If the variability of the interval $i = [\underline{i}, \overline{i}]$ is smaller than a parameter ω then, we apply $(F2)$, we substitute the variable `u` of the function `f` by the value γ and we return $\texttt{callee}_y$ as mentioned in Figure 3. In practice, we choose $\omega = 2^4 \times ulp(max(|\underline{i}|, |\overline{i}|)$ where ulp(x) is defined in Section 2.2. Note that, we conserve the original function in our code for the case when the condition on the variability is not satisfied.

Let us consider a call `z = f(e)` to a function `f(u){c; return v}`. More precisely, let γ be the abstract value obtained by abstract interpretation of the program for the

```
assert x = [100.0, 200.0]
double caller(){
 y = (x * x) + 15.0 ;
 z = callee(y) ;
 return z ; }
double callee(double u){
 v = (55.123 * u * u * u) + (12.453
   * u * u) + (239.078 * u) + 0.3 ;
 return v ; }
```
(1)

```
assert x = [100.0,200.0]
double caller(){
 y = (x * x) + 15.0 ;
 u = y ;
 v = (55.123 * u * u * u)
   + (12.453 * u * u)
   + (239.078 * u) + 0.3 ;
 z = v ;
 return z ; }
```
(2)

```
assert x = [100.0, 200.0]
double caller(){
 v = ((((239.078 * (15.0 + (x * x))) + 0.3)
   + ((12.452 * (15.0 + (x * x))) * (15.0
   + (x * x)))) + (((55.123 * (15.0 + (x * x)))
   * (15.0 + (x * x))) * (15.0 + (x * x)))) ;
 z = v ;
 return z ; }
```
(3)

Figure 4: Example of program transformed by our tool. Left: The original program. Middle: The program obtained using the interprocedural transformation rule (F1). Right: The optimized program obtained by using the intraprocedural transformation rules.

```
assert a = [10.0, 20.0]
double caller(){
 x = 2.0 ; y = 3.0 * x + 9.0 ;
 z = callee(y) ;
 return z ; }
double callee(double  u){
 v = (a * a * u * u * u)
   + (a * u * u)
   + ((a * 0.5) * u) + 0.3 ;
 return v ; }
```
(1)

```
assert a = [10.0, 20.0]
double caller(){
 x = 2.0 ; y = 15.0 ;
 z = callee_y() ;
 return  z ; }
double callee_y(){
 v = (a * a * 15.0 * 15.0 * 15.0)
   + (a * 15.0 * 15.0)
   + ((a * 0.5) * 15.0) + 0.3 ;
 return v ; }
```
(2)

```
assert a = [10.0, 20.0]
double caller(){
 x = 2.0 ; y = 15.0 ;
 z = callee_y() ;
 return z ; }
double callee_y(){
 v = ((0.5 * (15.0 * a))
   + ((0.3 + ((a * 15.0) * 15.0))
   + ((((a * a) * 15.0) * 15.0) * 15.0))) ;
 return v ; }
```
(3)

Figure 5: Example of program transformed by our tool. Left: The original program. Middle: The program obtained using the interprocedural transformation rule (F2). Right: The optimized program obtained by using the intraprocedural transformation rules.

```
assert a = [10.0, 20.0]
double caller(){
 x = 2.0 ; y = 15.0 * x - 1.0 ;
 z = callee(y) ;
 return z ; }
double callee(double u){
 v = (a * a * u * u * u) + (a * u * u)
   + ((a * 0.5) * u) + 0.3 ;
 return v ; }
```
(1)

```
assert a = [10.0, 20.0]
double caller(){
 x = 2.0 ; y = 15.0 * x - 1.0 ;
 z = callee_lazy(x) ;
 return z ; }
double callee_lazy(double x){
 x = 2.0 ; u = 15.0 * x - 1.0 ;
 v = (a * a * u * u * u) + (a * u * u) + ((a * 0.5) * u) + 0.3 ;
 return v ; }
```
(2)

```
assert a = [10.0, 20.0]
double caller(){
 x = 2.0 ; y = 15.0 * x - 1.0 ;
 z = callee_x() ;
 return z ; }
double callee_x(){
 x = 2.0 ;
 v = (((a * ((15.0 * x - 1.0) * (15.0 * x - 1.0)))
   + (a * (a * (((15.0 * x - 1.0) * (15.0 * x - 1.0))
   * (15.0 * x - 1.0))))) + (0.3 + ((a * 0.5)
   * (15.0 * x - 1.0)))) ;
 return v ; }
```
(3)

```
assert a = [10.0, 20.0]
double caller(){
 x = 2.0 ; y = 15.0 * x - 1.0 ;
 z = callee_x() ;
 return z ; }
double callee_x(){
 x = 2.0 ;
 v =  (((((x * 15.0) - 1.0) * ((x * 15.0) - 1.0)) * a)
   + (((a * (a * ((((x * 15.0) - 1.0) * ((x * 15.0) - 1.0))
   * ((x * 15.0) - 1.0)))) + ((a * 0.5) * ((x * 15.0) - 1.0))) + 0.3)) ;
 return v ; }
```
(4)

Figure 6: Example of program transformed by our tool. Top left: The original program. Top right: The program obtained using the interprocedural transformation rule (F3). Down left: The program obtained using the interprocedural transformation rule (F2). Down right: The optimized program obtained by using the intraprocedural transformation rules.

expression `e`, $\gamma = [\![e]\!]^\sharp \sigma^\sharp$ where $\sigma^\sharp$ is the abstract environment given by the analysis of the program, *i.e.*, the analysis of `c` in the global context `C`, $\sigma^\sharp = [\![C[c]]\!]^\sharp \iota^\sharp$ ($\iota^\sharp$ is a global environment for the free variables of the program corresponding to entries provided, *i.e.*, by sensors). Let us assume that the intraprocedural rules transform `c` into `c'` assuming that the constant expression γ is assigned to `u` during the transformation and that `v` is the target variable of this transformation.

Then we create a new function with zero argument `g(){c'; return v}` and in the caller code, we substitute the assignment `z=g()` to `z = f(u)`. Note that this process is easily generalizable to functions of more than one arguments. The new black list β' contains the variable `z` in addition to the original *target variable* ν.

EXAMPLE 3.2. *Let us illustrate the application of Rule* $(F2)$ *on the code given in the left side of Figure 5. We substitute the parameters of the function* `callee()` *by the parameters of the calling function in the* `caller()`*. That means that we substitute the value of* `u` *of the function* `callee()` *by the value of* `y` *of the call function, so the new function is named* $\texttt{callee}_y\texttt{()}$ *without parameters (see the code given in the middle of Figure 5). We say that we have specialized the function* `callee()` *with its effective parameters. Next, we apply on the transformed program the intraprocedural transformation rules of Section 2.5 to improve its accuracy as shown in the right side of Figure 5.* ■

REMARK. *In our case, the functions are specialized according to their call site, and they are left separated. In other words, several calls to Rule* $(F2)$ *for the same function* `callee()` *yield several specialized versions of* `callee()`*. In future work, we aim at merging them when they are too many.*

3.3 Formal Expression Passing

The last rule $(F3)$ consists of substituting the formal expression of the parameters of a given call function to the formal parameters inside the body of the called function. It can be seen as a lazy evaluation of the parameters in the caller. By applying this rule, we obtain the new function $\texttt{callee}_{lazy}\texttt{()}$ whose parameters are the variables of the expressions of the call to the function. Then we rewrite the new function $\texttt{callee}_{lazy}\texttt{()}$ by using the intraprocedural transformation rules to optimize the accuracy of the computations.

Let `f(u){c; return v}` be a function of the program `P` and `z = f(e)` a call to `f` in same calling function. In Rule $(F3)$ of Figure 3, `c'` correspond to the code obtained by intraprocedural transformation of `c` in a formal environment mapping the formal parameter `u` to the formal expression `e` at the call site. The variable to be optimized is the returned variable `v`. This corresponds to the intraprocedural transformation $\langle c, \delta[u \mapsto e], [], \{v\}\rangle \rightarrow_z^* \langle c', \delta, [], \{v\}\rangle$. Let `g` be the new function obtained by transformation. The parameters of `g` are `Var(e)`, the free variables of `e`. For functions of many arguments, the same process is applied to each argument.

In addition, in the calling site `z = f(e)`, we substitute `Var(e)` to `e` in the arguments. For example, if `Var(e) = x` the call becomes `z = f(x)`.

REMARK. *Note that, in practice, we apply the inlining Rule* $(F1)$ *after the laziness Rule* $(F3)$*.*

EXAMPLE 3.3. *To understand the use of Rule* $(F3)$*, we take the example of Figure 6. We apply to it the third interprocedural transformation rule presented in Figure 3. As we observe, we have replaced the parameter* `u` *of the called function* `callee()` *with the formal expression corresponding to* `y` *in the main function. We create a new function named* $\texttt{callee}_{lazy}\texttt{()}$ *with a new parameter* `x` *corresponding to the variable* `z` *used in the expression assigned to* `y`*. Next, in this example, Rule* $(F2)$ *may be interestingly used after Rule* $(F3)$*. Its application gives the down left code of Figure 6. Finally, we call the intraprocedural transformation tool to optimize the intermediary program, in other words, the body of the new function* $\texttt{callee}_x\texttt{()}$*, as shown in the code of Figure 6.* ■

3.4 Choice of the Transformation Rule

In this section, we introduce our heuristic to choose the interprocedural transformation rule that is applied to the program in order to improve its numerical accuracy. We have defined a function `RuleSelector` which selects the appropriate rule to be used. This function includes a variable named `inlineAllowed` that allows or not the inlining of the callee function within the caller function. In other words, it allows or not the use of the first interprocedural transformation rule $(F1)$. The `inlineAllowed` allows to inline the function `f` by computing the expression

$$\frac{size(\texttt{f}) \times numCall(\texttt{f})}{size(\texttt{P})} \leq inlineFactor \tag{14}$$

- $size(\texttt{f})$ is the number of lines of the function `f`,
- $numCall(\texttt{f})$ is the number of the calls to `f` in the caller function,
- $size(\texttt{P})$ is the total lines number of the program `P`.

If the expression described by Equation (14) is less than a defined coefficient `inlineFactor` then we inline the function, otherwise the inlining is not allowed. In our case, the value of `inlineFactor` is initialized to 5. This value may be increased or decreased depending on the maximal code size of the transformed program that the user is ready to accept.

Second, if the first interprocedural transformation rule $(F1)$ is not applied, we use either the specialization function rule $(F2)$ or the lazy evaluation rule $(F3)$.

The choice between rules $(F2)$ and $(F3)$ is done as follows. Recall that in Section 3.2, we have given conditions on when to apply Rule $(F2)$: basically, it is used when the width of the intervals of the values of the arguments are small enough (in our implementation the intervals must contain at most $\omega = 2^4$ floating-point numbers). We apply Rule $(F2)$ when this condition is satisfied. Otherwise Rule $(F3)$ is applied. In our implementation, the application of Rule $(F3)$ is always followed by the application of Rule $(F2)$ when the conditions of this latter are satisfied (parameter ω). In each case, the interprocedural transformation is followed by the application of the intraprocedural rules to the body of all the functions.

4 EXPERIMENTS

In this section we discuss some experiments made with our tool on the interprocedural program transformation. To perform our tests, we have taken a set of four examples presented in previous work [7–9]: The first two examples are coming from embedded systems (Odomerty and PID controller) while

```
assert sl = [0.52,0.53]
double main(){
  x = 0.0 ; y = 0.0 ; arg  = 0.0 ; delta_d  = 0.0 ;
  delta_dl = 0.0 ; delta_dr = 0.0 ; delta_theta = 0.0 ;
  sr = 0.62831853071 ; t = 0.0; x = 0.0 ; y = 0.0 ;
  inv_l = 0.1 ; c = 12.34 ; theta = -0.985 ;
  while (t < 1.0) {
    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) ;
    z = cos(arg) ; x = x + (delta_d * z) ;
    q = sin(arg) ; y = y + (delta_d * q) ;
    theta = theta + delta_theta ; t = t + 0.1 ;
   } ; return x ;
}
double cos(double a){
  res = 1.0 - (a * a * 0.5) + ((a * a * a * a) * 0.0416) ;
  return res;
}
double sin(double u){
  res = u - ((u * u * u) * 0.1666)
      + ((u * u * u * u * u) * 0.0083) ;
  return res;
}
```

Figure 7: Original Odometry program.

the latter two examples are numerical algorithms (Newton-Raphson and Runge-Kutta). Recall that the target variable in each case is the variable returned by the main function.

We show in Table 1 the results obtained on our set of examples. The errors given are the worst errors computed by static analysis for all the possible entries given by the intervals (using the order relation of Equation (9)). The error on the original program is compared to the error on the programs obtained by intraprocedural and interprocedural transformation. The intraprocedural transformation is applied on each function separately. Note that for the intraprocedural transformation, the numbers differ from the ones given in previous articles because the programs have been rewritten in several functions (the intraprocedural transformation is then less efficient than on a single block).

In intraprocedural mode, the geometric mean improvement is $25,69\%$. In the interprocedural mode, the geometric mean improvement is $33,17\%$. The interprocedural rules are then clearly useful. Note that, we have used the geometric mean to compute the mean improvement of our benchmark results because the arithmetic mean may lead to mistaken conclusions.

4.1 Odometry

The first example consists in the Odometry program introduced in [7]. At the difference, the program given in Figure 7 contains two functions cos and sin that compute respectively the cosine and the sine using Taylor series expansions. Figure 8 gives the transformed program obtained by applying our interprocedural transformation on the program given in Figure 7. The accuracy of the transformed Odometry program is improved by 29.39% when using intraprocedural mode and by 39.98% while using interprocedural mode.

```
assert sl = [0.52,0.53]
double main() {
  t = 0.0 ; theta = -0.985 ; y = 0.0 ; x = 0.0 ;
  arg = 0.0 ; delta_theta = 0.0 ; delta_d = 0.0 ;
  delta_dr = 0.0 ; delta_dl = 0.0 ;
  while (t < 1.0) {
    delta_dl = (12.34 * sl) ; delta_dr = 7.753450668961398 ;
    delta_d = (0.5 * (delta_dl + delta_dr)) ;
    delta_theta = (0.1 * (delta_dr - delta_dl)) ;
    arg = (theta + (delta_theta * 0.5)) ;
    z = cosTMP_1() ; x = (x + (delta_d * z)) ;
    q = sinTMP_2() ; y = (y + (delta_d * q)) ;
    theta = (theta + delta_theta) ; t = (t + 0.1) ;
  } ; return x ;
}
double sinTMP_2() {
  res = ((((0.5 * delta_theta) + theta) - (0.1666 * (((0.5
      * delta_theta) + theta) * (((0.5 * delta_theta) + theta)
      * ((0.5 * delta_theta)+theta)))))+((((((0.0083 * (theta
      + (delta_theta * 0.5))) * (theta + (delta_theta * 0.5)))
      * (theta + (delta_theta * 0.5))) * (theta + (delta_theta
      * 0.5))) * (theta + (delta_theta * 0.5)))) ;
  return res ;
}
double cosTMP_1() {
  res = ((1.0 - (0.5 * (((0.5 * delta_theta) + theta) * ((0.5
      * delta_theta) + theta)))) + ((((0.0416 * (theta
      + (delta_theta * 0.5))) * (theta + (delta_theta * 0.5)))
      * (theta + (delta_theta * 0.5))) * (theta
      + (delta_theta * 0.5)))) ;
  return res ;
}
```

Figure 8: Transformed Odometry program.

```
double main(){
  eold = 0.0 ; t = 0.0 ; i = 0.0 ; c = 5.0 ; kp = 9.4514 ;
  kd = 2.8454 ; dt = 0.2 ; invdt = 5.0 ; m = 0.0 ;
  while (t < 10.0) {
    e = c - m ; p = kp * e ; i = integral(i,m,c,dt) ;
    d = kd * invdt * (e - eold) ; r = p + i + d ;
    eold = e ; m = m + r * 0.01 ; t = t + dt ;
  } ; return m ;
}
double integral(double ii, double mm,double cc,double ddt){
  ki = 0.69006 ; res = ii + (ki * ddt * (cc-mm)) ;
  return res ;
}
```

Figure 9: Original PID program.

Recall from Section 2.2 that we compute a bound on the worst error on the results for all the considered inputs (here the intervals for $\mathtt{s}_l$ and $\mathtt{s}_r$ which correspond to the speeds of the left and right wheel of the robot in radians per second). This relative error is 39.98% smaller for our returned variable x in the interprocedural transformed program.

4.2 PID Controller

The PID [2] is a widely used algorithm in embedded and critical systems, like aeronautic and avionic systems. It keeps a physical parameter (m) at a specific value known as the *setpoint* (c). In other words, it tries to correct a measure by maintaining it at a defined value. The original PID program is given in Listing 9. The function integral computes the integral of the error by the rectangle rule. The error being the difference between the *setpoint* and the measure, the

```
double main() {
  t = 0.0 ; m = 0.0 ; eold = 0.0 ; i = 0.0 ;
  while (t < 10.) {
    p = ((5.0 - m) * 9.4514) ; i = integral(i,m,5.0,0.2) ;
    d = (((5.0 - m) - eold) * 14.226999999999997) ;
    eold = (5.0 - m) ; m = (m + (0.01 * ((p + i) + d))) ;
    t = (t + 0.2) ;
  } ; return m ;
}
double integral(double ii,double mm,double cc,double ddt) {
  res = (ii + ((0.69006 * (cc - mm)) * ddt)) ; return res ; }
```

Figure 10: Transformed PID program.

```
double main(){
  a = 0.0 ; a0 = 0.0 ;
  while (a0 <= 3.0) {
    u = f(a) ; w = f-prime(a) ; a_n = a - (u / w) ;
    a = a_n ; a0 = a0 + 0.1 ;
  } ; return a_n ;
}
double f(double x){
  res = (x * x * x * x * x * x) - (10.0 * x * x * x * x * x)
      + (40.0 * x * x * x) - (80.0 * x * x)
      + (80.0 * x) - (32.0) ;
  return res ;
}
double f-prime(double v){
  res = (5.0 * v * v * v * v * v) - (40.0 *v * v * v * v)
      + (120.0 * v * v) - (160.0 * v) + (80.0) ;
  return res ; }
```

Figure 11: Original Newton-Raphson's method.

controller computes a correction based on the integral i and derivative d of the error and also from a proportional error term p. The weighted sum of these three terms contributes to improve the reactivity, robustness and speed of the `PID`.

Figure 10 gives the transformed program obtained by applying our interprocedural transformation on the program given in Figure 9. The accuracy of the transformed PID program is improved by 13.24% if we use intraprocedural mode and by 18.45% if we use interprocedural transformation rules. It means that the relative error is 18.45% smaller for our returned variable `m` in the interprocedural transformed program. No improvement is obtained if we only apply the intraprocedural rules to each function separately.

4.3 Newton-Raphson's Method

In this section, we take the example of the Newton-Raphson method [19]. It is a numerical method used to compute the successive approximations of the zeros of a real-valued function [7]. Note that this numerical method is often used in embedded systems (with a bound number of iterations). We have shown in former work that even if Newton-Raphson method converges quickly in the reals, it may converge very slowly in floating-point arithmetic when the target function is evaluated inaccurately.

In Figure 11, our program contains two functions that compute respectively the polynomial $(x-2)^5$ and its derivative. The original program corresponding to this method is given in Figure 11. By applying the interprocedural transformation, the transformed program obtained is given in Figure 12. The

```
double main() {
  a0 = 0.0 ; a = 0.0 ;
  while (a0 <= 3.0) {
    u = (-32.0 + ((a * 80.0) + (((a * (a * (a * 40.0)))
      + ((a * (a * (a * (a * a)))) - (a * (a * (10.0
      * (a * a)))))) - (a * (a * 80.0))))) ;
    w = (80.0 + (((a * (a * 120.0)) + ((5.0 * ((a * a)
      * (a * a))) - (a * (40.0 * (a * a))))) - (a * 160.0)));
    res = ((u / -w) + a) ; a = a_n ; a0 = (a0 + 0.1) ;
  } ; return res ; }
```

Figure 12: Transformed Newton-Raphson's method program.

```
double main(){
  x = 1.0; y = 1.0; z = 0.0; h = 0.1; t = 0.0; yn = 0.0;
  coef = 0.16666667;
  while (t < 1.0){
    ff = f(x,y) ; k1 = h * ff ; p1 = p(x,h) ;
    p2 = p(y,k1) ; p3 = f(p1,p2) ; k2 = h * p3 ;
    q1 = p(y,k2) ;  q2 = f(p1,q1) ; k3 = h * q2 ;
    s1 = x + h ; s2 = y + k3 ; s3 = f(s1,s2) ; k4 = h * s3 ;
    yn = y + (coef * h * (k1 + (2.0 * k2) + (2.0 * k3) + k4));
    t = t + 0.1; y = yn;
  }; return yn;
}
double f(double u, double v){
  res = 2.0 * u + 3.0 * v ; return res ; }
double p(double xx, double hh){
  res = xx + hh * 0.5 ; return res ; }
```

Figure 13: Original Runge-Kutta Method.

numerical accuracy of the transformed program of Newton-Raphson method when applying the intraprocedural rules is improved by 14.89% (see Table 1, otherwise, by using the interprocedural rules, its accuracy is improved by 21.79%. This significant improvement is mainly due to the correction floating-point errors arising during the computations of the developed form of polynomial which evaluate very poorly close to the root.

```
double main() {
  y = 1.0 ; t = 0.0 ; yn = 0.0 ;
  while (t < 1.0){
    ff = fTMP_3() ; p1 = pTMP_4() ;
    p2 = pTMP_7() ; p3 = fTMP_8() ;
    q1 = pTMP_11() ; q2 = fTMP_12() ;
    s2 = (y + (0.1 * 5.9775)) ; s3 = fTMP_13() ;
    yn = ((((s3 * 0.1) + ((0.1 * (2.0 * 5.9775)) + ((0.1
       * (2.0 * 5.85)) + (ff * 0.1)))) * 0.016666667) + y) ;
    t = (t + 0.1) ; y = yn ;
  } ; return yn ;
}
double fTMP_13() { res = 6.993249999999997 ; return res ; }
double fTMP_12() { res = 5.977499999999998 ; return res ; }
double pTMP_11() { TMP_9 = 1.0 ; TMP_10 = 0.585 ;
  res = (1.0 + (TMP_10 * 0.5)) ; return res ; }
double fTMP_8() { res = 5.849999999999998 ; return res ; }
double pTMP_7() { TMP_5 = 1.0 ; TMP_6 = 0.5 ;
  res = (1.0 + (TMP_6 * 0.5)) ; return res ; }
double pTMP_4() { res = 1.05 ; return res ; }
double fTMP_3() { TMP_1 = 1.0 ; TMP_2 = 1.0 ;
  res = (2.0 + 2.9999999999999999) ; return res ; }
```

Figure 14: Transformed Runge-Kutta Method.

Code	Initial absolute error of original program	New absolute error after intraprocedural transformation	Percentage of improvement	New absolute error after interprocedural transformation	Percentage of improvement
Odometry	$4.3972539271e^{-14}$	$1.9179826428e^{-14}$	29.39%	$1.5940113894e^{-14}$	39.98%
PID	$5.3075447922e^{-8}$	$3.1161972335e^{-10}$	13.24%	$2.5645650587e^{-10}$	18.45%
Newton	$3.2148412680e^{-14}$	$1.5694371894e^{-14}$	14.89%	$1.0744405055e^{-15}$	21.79%
RK4	$1.2662168373e^{-15}$	$2.6672691579e^{-16}$	75.22%	$2.5951463231e^{-16}$	75.37%

Table 1: Comparison between the accuracy results of programs before and after optimization.

4.4 Runge-Kutta Method

In this section, we aim at transforming the fourth order Runge-Kutta method in order to improve its accuracy using our interprocedural rules presented in Section 3. We give in Figure 13 the original program corresponding to this method with functions. After interprocedural transformation, our tool returns the program given in Figure 14 which accuracy improvement is 75.37%. Note that, when using the intraprocedural transformation, the transformed program is improved by 75.22% as shown on Table 1.

5 CONCLUSION

In this article, we have presented an automatic method to improve the numerical accuracy of computations of interprocedural programs by automatic transformation. It is based on a set of transformation rules which are defined in Figure 3 and which have been implemented in our tool. A heuristic to choose between the rules $(F1)$ to $(F3)$ is also described in Section 3.4. The experimental results applied on various programs either coming from embedded systems or numerical methods, show the efficiency of our transformation in terms of accuracy improvement.

A significant interest would be to extend the current work with a case study concerning a large size numerical application. Another interesting perspective consists of extending our work to optimize the high performance computing programs. In this direction, we aim at solving new numerical accuracy problems like the order in an operation of a distributed system. We are also interested also in studying the compromise between execution time, computation performances, numerical accuracy and the convergence acceleration of numerical methods. 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 about distributed systems, an important issue concerns the reproducibility of the results: different runs of the same application yield different results on different machines due to the variations in the order of evaluation of the mathematical expression. We aim at studying how our technique improves reproducibility.

REFERENCES

[1] ANSI/IEEE. *IEEE Standard for Binary Floating-Point Arithmetic*. SIAM, 2008.

[2] K. J. Åström and T. Hagglund. *PID Controllers, 2nd ed.* Instrument Society of America, 1995.

[3] 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.

[4] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[5] P. Cousout and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *POPL*, pages 178–190. ACM, 2002.

[6] R. Cytron and R. Gershbein. Efficient accomodation of may-alias information in SSA form. In *PLDI*, pages 36–45. ACM, 1993.

[7] N. Damouche, M. Martel, and A. Chapoutot. Intra-procedural optimization of the numerical accuracy of programs. In *FMICS'15*, volume 9128 of *LNCS*, pages 31–46. Springer, 2015.

[8] N. Damouche, M. Martel, and A. Chapoutot. Data-types optimization for floating-point formats by program transformation. In *CoDIT*. IEEE, 2016.

[9] N. Damouche, M. Martel, and A. Chapoutot. Improving the numerical accuracy of programs by automatic transformation. *STTT*, 2016.

[10] N. Damouche, M. Martel, P. Panchekha, C. Qiu, A. Sanchez-Stern, and Z. Tatlock. Toward a standard benchmark format and suite for floating-point analysis. In *NSV'16*, LNCS. Springer, 2016.

[11] E. Darulova and V. Kuncak. Sound compilation of reals. In *POPL'14*, pages 235–248. ACM, 2014.

[12] 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 *FMICS*, volume 5825 of *LNCS*, pages 53–69. Springer, 2009.

[13] P-L. Garoche, F. Howar, T. Kahsai, and X. Thirioux. Testing-based compiler validation for synchronous languages. In *NFM*, volume 8430 of *LNCS*, pages 246–251. Springer, 2014.

[14] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, 1991.

[15] E. Goubault. Static analysis by abstract interpretation of numerical programs and systems, and FLUCTUAT. In *SAS*, volume 7935 of *LNCS*, pages 1–3. Springer, 2013.

[16] E. Goubault, M. Martel, and S. Putot. Some future challenges in the validation of control systems. In *ERTS*, 2006.

[17] E. Hankin. *Lambda Calculi A Guide For Computer Scientists*. Clarendon Press, Oxford, 1994.

[18] A. Ioualalen and M. Martel. A new abstract domain for the representation of mathematically equivalent expressions. In *SAS'12*, volume 7460 of *LNCS*, pages 75–93. Springer, 2012.

[19] Atkinson K. *An Introduction to Numerical Analysis*. J. Wiley & Sons, 1989.

[20] M. Martel. Semantics of roundoff error propagation in finite precision calculations. *Higher-Order and Symbolic Computation*, 19(1):7–30, 2006.

[21] 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.

[22] J. R. Wilcox P. Panchekha, A. Sanchez-Stern and Z. Tatlock. Automatically improving accuracy for floating point expressions. In *PLDI*, pages 1–11. ACM, 2015.

[23] S. Sagiv, T. W. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167(1&2):131–170, 1996.

[24] A. Solovyev, C. Jacobsen, Z. Rakamaric, and G. Gopalakrishnan. Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. In *FM*, volume 9109 of *LNCS*, pages 532–550. Springer, 2015.

[25] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: A new approach to optimization. *Logical Methods in Computer Science*, 7(1), 2011.