Program Transformation for Numerical Precision

Matthieu Martel

Laboratoire ELIAUS-DALI
Université de Perpignan Via Domitia
52 avenue Paul Alduy
66860 Perpignan Cedex, France
matthieu.martel@univ-perp.fr

Abstract

This article introduces a new program transformation in order to enhance the numerical accuracy of floating-point computations. We consider that a program would return an exact result if the computations were carried out using real numbers. In practice, roundoff errors due to the finite representation of values arise during the execution. These errors are closely related to the way formulas are evaluated. Indeed, mathematically equivalent formulas, obtained using laws like associativity, distributivity, etc., may lead to very different numerical results in the computer arithmetic. We propose a semantics-based transformation in order to optimize the numerical accuracy of programs. This transformation is expressed in the abstract interpretation framework and it aims at rewriting pieces of numerical codes in order to obtain results closer to what the computer would output if it used the exact arithmetic.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Program Verification—Validation; D.3.4 [*Programming Languages*]: Processors—Optimization; F.3.2 [*Logics and Meaning of Programs*]: Semantics of Programming Languages—Program Analysis; G.1.0 [*Mathematics of Computing*]: Numerical Analysis—Computer Arithmetic

General Terms Algorithms, Languages, Theory, Verification

Keywords Program Transformation, Numerical Precision, Abstract Interpretation, Compiler Optimizations, Floating-Point Numbers

1. Introduction

This article introduces a new program transformation to enhance the numerical accuracy of floating-point computations which comply to the IEEE754 Standard [1]. We consider that a program would return an exact result if the computations were carried out using real numbers. Indeed, many programmers and compiler designers use floating-point arithmetic as if it were real arithmetic. However, in practice, round-off errors arise at run-time [8, 20]. This is because floating-point arithmetic differs strongly from real number arithmetic: the values have a finite number of digits and algebraic laws like associativity, distributivity, commutativity do not hold. The round-off errors are closely related to the way the formulas are written. For example, the evaluation by a computer of mathematically equivalent expressions (e.g. $x \times (1 + x)$ and $x + x^2$) possibly leads to very different results. Our program transformation aims at manipulating the mathematical expressions in order to produce new expressions which are more accurate and mathematically equivalent to the original ones.

Understanding the reasons why the implementation of a formula is numerically bad and how to improve it is usually difficult because computer arithmetics are particularly not intuitive. So, it is necessary to provide tools to the programmers, in order to help them to increase the numerical quality of their codes. During the last few years, static analyses by abstract interpretation [5] of the numerical accuracy of floating-point computations have been introduced [9, 15, 17] and implemented in the Fluctuat tool [10, 11] which is used in many 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. More generally, program transformations to detect numerical errors are runtime are discussed in [3] and other methods, not based on static analysis, are compared in [16]. Concerning the transformation of programs in order to enhance their numerical accuracy, there only exists non-automatic methods dedicated to specific classes of formulas, for example to improve the evaluation of polynomial expressions [4, 13].

Semantics-based program transformation [6, 12] for floating-point arithmetic expressions has been introduced [18]. This method enables one to automatically rewrite a general formula into another mathematically equivalent and more precise formula. In [19], it has been shown that the transformation could also be applied to the case of fixed-point computations [21, 22].

In this article, we extend the transformation for numerical accuracy to the case of entire programs. While previous work [18, 19] has only focused on how to rewrite arithmetic expressions, we consider a simple imperative language with variables, conditionals and loops. It is then possible to transform computations carried out among many lines of code, to modify the expressions assigned to intermediary variables and to use standard compile-time techniques like loop unrolling to improve the numerical accuracy inside the body of loops. Intuitively, the main originality of the program transformation is to allow, in the non-standard and abstract semantics, to evaluate either eagerly or lazily the arithmetic expressions assigned to variables. Then, when a variable is read, the expression may be either evaluated or inserted into the larger expression which uses the variable. This mechanism permits to recombine the operations among many assignments. We use P. Cousot and R. Cousot's

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PEPM'09, January 19–20, 2009, Savannah, Georgia, USA.
Copyright © 2009 ACM 978-1-60558-327-3/09/01. . . $5.00

framework for semantics-based program transformation [6] by abstract interpretation [5] and we propose an offline transformation. The methodology of [6] enables us to define a semantics transformation that would be far more difficult to obtain at the syntactic level, since there is no strong syntactic relation between the source and transformed expressions.

This article is organized as follows. Section 2 informally presents the program transformation. Section 3 introduces the numerical domains used by our semantics. The non-standard semantics of programs is given in Section 4 and the abstract semantics is introduced in Section 5. Section 6 presents the program transformation based on the abstract semantics. Finally, experimental results are given in Section 7 and Section 8 concludes.

2. Overview

This section briefly illustrates the kind of code transformations we wish. Basically, we aim at optimizing the numerical precision of floating-point computations, i.e. we want to minimize the difference between the result output by the machine and the mathematical result that we would obtain if the computer used exact arithmetic. For example, let us consider the arithmetic expression `e=(a*((b+c)+d))` and let us assume that the inputs belong to the following intervals:

$$a = [56789, 98765] \qquad b = [0, 1] \\ c = [0, 5e^{-8}] \qquad d = [0, 5e^{-8}] \tag{1}$$

Assuming that we are using the IEEE754 single precision, we want to perform the following transformation:

$$\texttt{e=(a*((b+c)+d))} \longrightarrow \texttt{e'=((a*b)+(a*(c+d)))}$$

The sums are parsed in order to first add the smallest terms: this limits the errors since, in IEEE754 single precision, we have $\texttt{b} + \texttt{c} = \texttt{b}$. Furthermore, the product is distributed and this avoids the multiplication of the round-off errors of the additions by a large value and, consequently, this also reduces the final error. Using the domains introduced in Section 3 to compute an over-approximation of the error attached to the result of a floating-point computation, we obtain a bound on the maximal error arising during the evaluation of the expression (for any concrete set of inputs in the intervals given in Equation (1)). The errors for the source and transformed expressions are:

- Error bounds on `e`: [-1.5679E-2,1.5680E-2],
- Error bounds on `e'`: [-7.8125E-3,7.8126E-3].

The error on the transformed expression is approximately half the error on the original expression.

The transformation of arithmetic expressions has been defined in [18]. In this article, we generalize this work to the case of full programs, with variables, conditionals and loops. For example, it is well known that a sequence of numbers has to be sorted increasingly before being added. If we take

$$s = \sum_{i=0}^{4} x_i \tag{2}$$

with $x_i = [2^i, 2^{i+1}]$, we obtain the results of Figure 1, where `a`, `b`, `c`, `d` and `e` stand for x_0, x_1, x_2, x_3 and x_4, respectively. Depending on the user-defined parameters giving the precision of the transformation, all the formulas of Figure 1 can be automatically found by our technique.

In a program, a sequence of additions would typically arise inside a loop. The usual way to implement Equation (2), assuming that the x_i are stored in an array `x[]` of `N` elements (`N` is supposed even) is given on the left-hand-side of Figure 2. It would not be reasonable to expect that a general and automatic semantics-based

Expression	Error bound
`(((e+d)+c)+b)+a`	[-7.6293E-6,7.6294E-6]
`(b+a)+(c+(e+d))`	[-5.9604E-6,5.9605E-6]
`(c+(b+a))+(e+d)`	[-4.5299E-6,4.5300E-6]
`(d+(c+(a+b)))+e`	[-3.5762E-6,3.5763E-6]

Figure 1. Error bounds on the evaluation in IEEE754 single precision of mathematically equivalent expressions.

```
int i=0;                     int i=0;
float s=0.0;                 float s=0.0;
while (i<N) {        ⟶      while (i<N) {
  s=s+x[i];                    l=x[i]+x[i+1];
  i=i+1;                       s=s+l;
}                              i=i+2;
                             }
```

Figure 2. Example of program transformation in the case of an array of decreasingly sorted numbers.

code transformation inserts a sort procedure in the new program. However, the program given on the right-hand-side of Figure 2 already significantly improves the precision of the computations. In the new program, the body of the loop has been unrolled by a factor 2 and the computation $s = (s + x_i) + x_{i+1}$ has been rewritten as $s = s + (x_i + x_{i+1})$. This corresponds to what was done for expressions, in Figure 1 for example. For technical reasons, an intermediary variable ℓ has been introduced and the new program computes $\ell = x_i + x_{i+1}$ and then $s = s + \ell$. We use a standard compile-time technique to unroll the loops [2] before applying the transformation for numerical precision: the body is replicated some number n of times (called the factor) and the loop counter is incremented by n instead of 1. Currently, the factor is a user-defined parameter of the transformation but there also exists standard heuristics to find efficient unrolling factors (e.g. based on the size of the code).

Assuming that $x_i = [2^{\texttt{N}-i-1}, 2^{\texttt{N}-i}]$ and that $\texttt{N}= 4$, we obtain, in IEEE754 single precision, the following error bounds on `s` depending on the program we use:

- Original program: $[0.0, 2.861022950E - 6]$,
- Transformed program: $[0.0, 2.145767212E - 6]$,
- Best bound (uses a sort): $[0.0, 1.668930054E - 6]$.

Observe that the error bound obtained using the transformed code is closer to the best error bound than to the error bound of the original code.

3. Numerical Domains

This section briefly surveys the aspects of floating-point arithmetic useful to the comprehension of the rest of this article. It also defines the precision measure that we aim at improving by program transformation.

The IEEE754 Standard specifies the representation of numbers and the semantics of the elementary operations for floating-point arithmetic [1, 8, 20]. It is implemented in most of modern general-purpose processors. First of all, a floating-point number x in base β is defined by

$$x = s \cdot (d_0.d_1 \dots d_{p-1}) \cdot \beta^e = s \cdot m \cdot \beta^{e-p+1} \tag{6}$$

where $s \in \{-1, 1\}$ is the sign, $m = d_0 d_1 \dots d_{p-1}$ is the mantissa with digits $0 \le d_i < \beta$, $0 \le i \le p - 1$, p is the precision and e is the exponent, $e_{min} \le e \le e_{max}$. A floating-point number x is normalized whenever $d_0 \ne 0$. Normalization avoids multiple

$$(E1^\natural) \quad \frac{v = v_1 \bullet v_2}{\langle v_1 \bullet v_2, \tau\rangle \ \to^\natural \ v} \qquad\qquad \frac{\tau(x) = v}{\langle x, \tau\rangle \ \to^\natural \ v} \quad (E2^\natural)$$

$$(E3^\natural) \quad \frac{\langle e_1, \tau\rangle \ \to^\natural \ e_1'}{\langle e_1 \bullet e_2, \tau\rangle \ \to^\natural \ e_1' \bullet e_2} \qquad\qquad \frac{\langle e_2, \tau\rangle \ \to^\natural \ e_2'}{\langle v_1 \bullet e_2, \tau\rangle \ \to^\natural \ v_1 \bullet e_2'} \quad (E4^\natural)$$

$$(x_1^\sharp, \varepsilon_1^\sharp) + (x_2^\sharp, \varepsilon_2^\sharp) = \Big(\uparrow_\circ^\sharp (x_1^\sharp + x_2^\sharp), \varepsilon_1^\sharp + \varepsilon_2^\sharp + \downarrow_\circ^\sharp (x_1^\sharp + x_2^\sharp)\Big) \tag{3}$$

$$(x_1^\sharp, \varepsilon_1^\sharp) - (x_2^\sharp, \varepsilon_2^\sharp) = \Big(\uparrow_\circ^\sharp (x_1^\sharp - x_2^\sharp), \varepsilon_1^\sharp - \varepsilon_2^\sharp + \downarrow_\circ^\sharp (x_1^\sharp - x_2^\sharp)\Big) \tag{4}$$

$$(x_1^\sharp, \varepsilon_1^\sharp) \times (x_2^\sharp, \varepsilon_2^\sharp) = \Big(\uparrow_\circ^\sharp (x_1^\sharp \times x_2^\sharp), x_1^\sharp \times \varepsilon_2^\sharp + x_2^\sharp \times \varepsilon_1^\sharp + \varepsilon_1^\sharp \times \varepsilon_2^\sharp + \downarrow_\circ^\sharp (x_1^\sharp \times x_2^\sharp)\Big) \tag{5}$$

Figure 3. The operational semantics of arithmetic expressions and the abstract semantics of the elementary operations for the floating-point arithmetic.

representations of the same number. IEEE754 Standard introduces a few values for p, e_{min} and e_{max}. For example, single precision numbers are defined by $\beta = 2$, $p = 23$, $e_{min} = -126$ and $e_{max} = +127$. The IEEE754 Standard also specifies special values (denormalized numbers, infinites and NaN) which are not used in this article.

Let $\uparrow_\circ : \mathbb{R} \to \mathbb{F}$ be the function which returns the round-off of a real number following the rounding mode $\circ \in \{\circ_{-\infty}, \circ_{+\infty}, \circ_0, \circ_\sim\}$ (towards $\pm\infty$, 0 or to the nearest). $\uparrow_\circ$ is fully specified by the IEEE754 Standard which also requires, for any elementary operation $\lozenge$, that:

$$x_1 \lozenge_{\mathbb{F},\circ} x_2 = \uparrow_\circ (x_1 \lozenge_\mathbb{R} x_2) \tag{7}$$

Equation (7) states that the result of an operation between floating-point numbers is the round-off of the exact result of this operation. In this article, we also use the function $\downarrow_\circ : \mathbb{R} \to \mathbb{R}$ which returns the round-off error. We have:

$$\downarrow_\circ (r) = r - \uparrow_\circ (r) \tag{8}$$

Enhancing the quality of the implementation of a formula $f(\mathbf{x})$ then consists of minimizing the round-off error on the result. In other words, using the notation of Equation (8), we aim at minimizing $\downarrow_\circ (f(\mathbf{x}))$, for all the possible vectors of inputs $\mathbf{x}$.

We introduce a non-standard semantics for the arithmetic expressions whose syntax is given by:

$$e ::= v \mid x \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \times e_2 \tag{9}$$

In Equation (9), $x \in \mathsf{Var}$ denotes a variable and v denotes a value. The semantics of expressions is related to the measure of the numerical accuracy. A value $v \in \mathbb{D}$ is a pair (x, ε) where x denotes the computer number, i.e. the floating-point number manipulated by the machine, and ε measures the numerical quality of x. In practice, ε denotes the distance between a real number $x_\mathbb{R}$ and the floating-point number x corresponding to the round-off of $x_\mathbb{R}$ (i.e. $\downarrow_\circ (x_\mathbb{R})$ as defined in Equation (8)). In addition, we consider a left-to-right evaluation order for the expressions, as given by the straightforward reduction rules of Figure 3, where $\tau : \mathsf{Var} \to \mathbb{D}$ is an environment for the variables and where $\bullet$ stands for one of the operations +, - or ×.

In this article, we also consider abstract values $(x^\sharp, \varepsilon^\sharp) \in \mathbb{D}^\sharp$ where $x^\sharp$ and $\varepsilon^\sharp$ are intervals whose bounds are floating-point numbers. A value $(x^\sharp, \varepsilon^\sharp)$ abstracts a set of concrete values $\{(x_i, \varepsilon_i),\ i \in I\}$ by intervals in a component-wise way. The reduction rules of Figure 3 are left unchanged for the abstract semantics.

The abstract semantics of arithmetic operations, given in Figure 3, computes how the round-off errors are propagated. The abstract function $\uparrow_\circ^\sharp$ correspond to the concrete function $\uparrow_\circ$. We have:

$$\uparrow_\circ^\sharp ([\underline{x}, \overline{x}]) = [\uparrow_{-\infty} (\underline{x}), \uparrow_{+\infty} (\overline{x})] \tag{10}$$

The function $\downarrow_\circ^\sharp$ is a safe abstraction of $\downarrow_\circ$, i.e. $\forall x \in [\underline{x}, \overline{x}],\ \downarrow_\circ (x) \in \downarrow_\circ^\sharp ([\underline{x}, \overline{x}])$. For example, if the current rounding mode $\circ$ is to the nearest, one may choose

$$\downarrow_\circ^\sharp ([\underline{x}, \overline{x}]) = [-y, y] \quad \text{with } y = \frac{1}{2}\mathrm{ulp}\big(\max(|\underline{x}|, |\overline{x}|)\big) \tag{11}$$

where the *u*nit in the *l*ast *p*lace $\mathrm{ulp}(x)$ is the weight of the least significant digit of the floating-point number x [8]. For an addition, the errors on the operands are added to the error due to the round-off of the result, as specified by Equation (7). For a subtraction, the errors on the operands are subtracted. Finally, the semantics of the multiplication comes from the development of $(x_1^\sharp + \varepsilon_1^\sharp) \times (x_2^\sharp + \varepsilon_2^\sharp)$. The semantics of the other elementary operations (division and square root) relies on a power series development and is more complicated. It is fully explicited in [16, 17].

For example, let us consider the expression:

$$\mathtt{E} = (\mathtt{a} + (\mathtt{b} + (\mathtt{c} + \mathtt{d}))) \times \mathtt{e} \tag{12}$$

and let us assume that the variables belong to the ranges: $\mathtt{a} \in [-14, -13]$, $\mathtt{b} \in [-3, -2]$, $\mathtt{c} \in [3, 3.5]$, $\mathtt{d} \in [12.5, 13.5]$ and $\mathtt{e} = 2$. Using the semantics of Figure 3, we obtain that:

$$E_{\text{float}} = \big([-3, 4], [-2.861022949 \cdot 10^{-6}, 0]\big)$$

This value indicates that the result returned by the machine always belongs to the interval $[-3, 4]$ and that, for any combination of inputs taken in the correct ranges, the round-off error on the result is always less than $2.861022949 \cdot 10^{-6}$, in absolute value.

Given a value $(x^\sharp, \varepsilon^\sharp)$, the indicator $\varepsilon^\sharp$ measures the accuracy of the implementation of a formula in floating-point arithmetic, assuming that the inputs belong to certain ranges. In the next sections we introduce a program transformation enabling one to improve this measure.

4. Non-Standard Semantics

In this section, we introduce the non-standard semantics, denoted $\to$, that we use to define the program transformation. Intuitively, the non-standard semantics is a small-step operational semantics which is non-deterministic in the sense that, from a state s, there exists in general many elementary reduction steps $s \to s_1$, $s \to s_2$, ..., $s \to s_n$. This non-determinism comes from the fact that any arithmetic expression e may be transformed into a mathematically

equivalent one e' using standard rules such as associativity, commutativity or distributivity. In addition, in the semantics of commands, we allow two ways of evaluating the expressions which appear in the right-hand side of the assignments: eagerly or lazily (i.e. the expressions may be fully, partly or not evaluated at all before being assigned to variables in the environment).

(i) $(e_1 + e_2) + e_3 \equiv e_1 + (e_2 + e_3)$
(ii) $e_1 + e_2 \equiv e_2 + e_1$
(iii) $e \equiv e + 0$
(iv) $(e_1 \times e_2) \times e_3 \equiv e_1 \times (e_2 \times e_3)$
(v) $e_1 \times e_2 \equiv e_2 \times e_1$
(vi) $e \equiv e \times 1$
(vii) $e_1 \times (e_2 + e_3) \equiv e_1 \times e_2 + e_1 \times e_3$

Figure 5. Example of relation which may be used for the transformation of mathematical expressions.

From a formal point of view, we consider the programs generated by the grammar of Equation (13).

$$\begin{array}{rcl} e & ::= & v \mid x \mid e_1 \texttt{ + } e_2 \mid e_1 \texttt{ - } e_2 \mid e_1 \texttt{ x } e_2 \\ b & ::= & \texttt{true} \mid \texttt{false} \mid x \texttt{ == } v \mid x \texttt{ < } v \\ c & ::= & \texttt{skip} \mid c_1 \texttt{ ; } c_2 \mid x \texttt{ = } e \mid \texttt{if } b\ c_1\ c_2 \\ & & \mid \texttt{while } b \texttt{ do } c \end{array} \quad (13)$$

The arithmetic expressions have been defined in Section 3 and the boolean expressions are boolean values or elementary comparisons. The commands define the core of an imperative programming language with sequences, assignments, conditionals and loops.

The semantics of the arithmetic expressions is given in Figure 4. In the reduction rules, $\bullet \in \{ +, -, \times \}$ denotes any elementary operation. A label $\ell \in \mathsf{Lab}$ is attached to each value occurring in the expressions and we use three environments. The first environment $\tau : \mathsf{Var} \to \mathbb{D}$ binds variables to values. Next, $\rho : \mathsf{Lab} \to \mathsf{Expr}$ maps any label ℓ to the expression e whose evaluation has lead to v^ℓ. Finally, the environment $\sigma : \mathsf{Expr} \to \mathbb{D}$ maps expressions to the result of their evaluation in the domain $\mathbb{D}$. We let Env_ρ, Env_σ and Env_τ denote the sets of such environments. The information collected by these environments is useful in the abstract semantics of Section 5.

Initially, a unique label is attached to each value occurring in an expression and a fresh label is associated to the result of each operation. For example, assuming that initially $\rho(\ell_1) = 1^{\ell_1}$, $\rho(\ell_2) = 2^{\ell_2}$ and $\rho(\ell_3) = 3^{\ell_3}$, the expression $(1^{\ell_1} + (2^{\ell_2} + 3^{\ell_3}))$ is evaluated as follows in the non-standard semantics:

$\langle (1^{\ell_1} + (2^{\ell_2} + 3^{\ell_3})), \rho, \sigma, \tau \rangle \to \langle 1^{\ell_1} + 5^{\ell_4}, \rho', \sigma', \tau \rangle \to \langle 6^{\ell_5}, \rho'', \sigma'', \tau \rangle$

where $\rho' = \rho[\ell_4 \mapsto 2^{\ell_2} + 3^{\ell_3}]$, $\sigma' = \sigma[2^{\ell_2} + 3^{\ell_3} \mapsto 5]$, $\rho'' = \rho'[\ell_5 \mapsto 1^{\ell_1} + (2^{\ell_2} + 3^{\ell_3})]$ and $s'' = \sigma'[1^{\ell_1} + (2^{\ell_2} + 3^{\ell_3}) \mapsto 6]$. In this example, for the sake of simplicity, numbers are integers instead of values of $\mathbb{D}$.

The rules $(E1)$ to $(E3)$ correspond to the rules $(E1^\natural)$ to $(E4^\natural)$ given in Figure 3 for the standard semantics. The originality of the semantics $\to$, concerning the arithmetic expressions, comes from Rule $(E4)$ which states that an expression may be rewritten into a mathematically equivalent one before or after a reduction step. The relation $\equiv$ that we use in practice is given in Figure 5 but it could be extended to other mathematical laws. Our relation $\equiv$ allows to transform expressions by associativity, commutativity, distributivity, and addition or product with neutral elements. Because commutativity is included in $\equiv$, we do not need, in the non-standard semantics, a rule corresponding to Rule $(E4^\natural)$: in the non-standard semantics, the evaluation of the rightmost operand is handled by a combination of Rule $(E3)$ and Rule $(E4)$.

For the sake of simplicity, we only consider simple boolean expressions of the form $x \texttt{ == } v$ or $x \texttt{ < } v$. In the most general case, the evaluation of tests should also introduce non-determinism by permitting to rewrite the conditions into other mathematically equivalent conditions. For instance, we could use the relation $\equiv_\prec$ that allows one to add a constant to both terms of an equality or inequality and to multiply by a constants both terms of an equality or inequality (by reverting the order in case of a product by a negative value).

The semantics of commands is given in Figure 4. It is a very standard semantics, excepted the combination of the rules $(C1)$ and $(C2)$: Rule $(C1)$ defines a lazy evaluation of the expression assigned to a variable x while Rule $(C2)$ allows an eager evaluation: contrarily to Rule $(C2)$, which performs some actual operations in the evaluation of e, Rule $(C1)$ permits to bind x to the expression e itself. As a result, in our semantics, the expressions may be evaluated either eagerly or lazily and the choice between these two strategies is non-deterministic. This mechanism enables us to insert e into a bigger expression e' and to rewrite later the resulting expression using $\equiv$. The semantics of commands does not directly affect the environments ρ and σ.

5. Abstract Semantics

Because of the non-determinism, there is possibly an exponential number of execution paths for a program in the non-standard semantics. Beside of the classical abstraction of data, the abstract semantics aims at reducing the combinatorial explosion by only allowing a polynomial number of possibilities while conservatively approximating the non-standard semantics.

$$\begin{array}{rcl} \mathsf{Expr}_0^\sharp \ni \eta_0 & ::= & v^\ell \mid \top_\eta \\ \mathsf{Expr}_k^\sharp \ni \eta_k & ::= & \eta_{k-1} \mid \eta_{k-1} \texttt{ + } \eta_{k-1} \mid \eta_{k-1} \texttt{ - } \eta_{k-1} \\ & & \mid \eta_{k-1} \texttt{ x } \eta_{k-1} \end{array}$$

$$\begin{array}{rcll} \ulcorner v^\ell \urcorner^k & = & v^\ell & k \geq 0 \\ \ulcorner \top_\eta \urcorner^k & = & \top_\eta & k \geq 0 \\ \ulcorner e_1 \bullet e_2 \urcorner^0 & = & \top_\eta & \\ \ulcorner e_1 \bullet e_2 \urcorner^k & = & \ulcorner e_1 \urcorner^{k-1} \bullet \ulcorner e_2 \urcorner^{k-1} & k \geq 1 \end{array}$$

Figure 6. Abstract expressions and the abstraction function.

Concerning the arithmetic expressions, the abstract semantics is mainly designed to limit the number of reduction paths. We introduce abstract expressions as well as a coarser relation $\equiv_k$ which reduces the number of possibilities allowed by $\equiv$. From a formal point of view, the abstract semantics of expressions, given in Figure 7, is obtained by applying the following modifications to the non-standard semantics:

(i) To limit the combinatorial explosion of the number of traces due to the Rule $(E4)$, we introduce the set $\mathsf{Expr}_k^\sharp$ of abstract expressions of height at most k and the abstraction function $\ulcorner . \urcorner^k : \mathsf{Expr} \to \mathsf{Expr}_k^\sharp$. The set $\mathsf{Expr}_k^\sharp$ and the function $\ulcorner . \urcorner^k$ are defined in Figure 6. The special element $\top_\eta$ denotes any expression. Note that, in abstract expressions, labels are attached to values (and only values). In the abstract semantics, a new label, generated dynamically, is attached to the new values coming from the result of intermediary computations.

(ii) Rule $(E4^\sharp)$ of Figure 7 is substituted to Rule $(E4)$ where $\equiv_k$ is a new relation which identifies mathematically equivalent abstract expressions. In other words, the relation $\equiv_k$ is the quotient relation $\equiv / \sim_k$ where $\sim_k$ is defined by:

$$e \sim_k e' \iff \ulcorner e \urcorner^k = \ulcorner e' \urcorner^k.$$

$$(E1)\quad \frac{v = v_1 \bullet v_2 \quad \rho' = \rho[\ell \mapsto \rho(\ell_1) \bullet \rho(\ell_2)] \quad \sigma' = \sigma[\rho(\ell_1) \bullet \rho(\ell_2) \mapsto v] \quad \ell \text{ fresh}}{\langle v_1^{\ell_1} \bullet v_2^{\ell_2}, \rho, \sigma, \tau\rangle \rightarrow \langle v^{\ell}, \rho', \sigma', \tau\rangle}$$

$$(E2)\quad \frac{\tau(x) = e}{\langle x, \rho, \sigma, \tau\rangle \rightarrow \langle e, \rho, \sigma, \tau\rangle} \qquad \frac{\langle e_1, \rho, \sigma, \tau\rangle \rightarrow \langle e_1', \rho', \sigma', \tau\rangle}{\langle e_1 \bullet e_2, \rho, \sigma, \tau\rangle \rightarrow \langle e_1' \bullet e_2, \rho', \sigma', \tau\rangle}\quad (E3)$$

$$(E4)\quad \frac{e \equiv e_1 \quad \langle e_1, \rho, \sigma, \tau\rangle \rightarrow \langle e_2, \rho', \sigma', \tau\rangle \quad e_2 \equiv e'}{\langle e, \rho, \sigma, \tau\rangle \rightarrow \langle e', \rho', \sigma', \tau\rangle}$$

$$(C1)\quad \frac{\tau' = \tau[x \mapsto e]}{\langle x = e, \rho, \sigma, \tau\rangle \rightarrow \langle \texttt{skip}, \rho, \sigma, \tau'\rangle} \qquad \frac{\langle e, \rho, \sigma, \tau\rangle \rightarrow \langle e', \rho', \sigma', \tau\rangle}{\langle x = e, \rho, \sigma, \tau\rangle \rightarrow \langle x = e', \rho', \sigma', \tau\rangle}\quad (C2)$$

$$(C3)\quad \frac{\langle c_1, \rho, \sigma, \tau\rangle \rightarrow \langle \texttt{skip}, \rho', \sigma', \tau'\rangle}{\langle c_1 \texttt{ ; } c_2, \rho, \sigma, \tau\rangle \rightarrow \langle c_2, \rho', \sigma', \tau'\rangle} \qquad \frac{\langle c_1, \rho, \sigma, \tau\rangle \rightarrow \langle c_1', \rho', \sigma', \tau'\rangle}{\langle c_1 \texttt{ ; } c_2, \rho, \sigma, \tau\rangle \rightarrow \langle c_1' \texttt{ ; } c_2, \rho', \sigma', \tau'\rangle}\quad (C4)$$

$$(C5)\quad \frac{\langle b, \rho, \sigma, \tau\rangle \rightarrow b'}{\langle \texttt{if } b\ c_1\ c_2, \rho, \sigma, \tau\rangle \rightarrow \langle \texttt{if } b'\ c_1\ c_2, \rho, \sigma, \tau\rangle}$$

$$(C6)\quad \frac{b = \texttt{true}}{\langle \texttt{if } b\ c_1\ c_2, \rho, \sigma, \tau\rangle \rightarrow \langle c_1, \rho, \sigma, \tau\rangle} \qquad \frac{b = \texttt{false}}{\langle \texttt{if } b\ c_1\ c_2, \rho, \sigma, \tau\rangle \rightarrow \langle c_2, \rho, \sigma, \tau\rangle}\quad (C7)$$

$$(C8)\quad \frac{\langle b, \rho, \sigma, \tau\rangle \rightarrow b'}{\langle \texttt{while } b \texttt{ do } c, \rho, \sigma, \tau\rangle \rightarrow \langle \texttt{while } b' \texttt{ do } c, \rho, \sigma, \tau\rangle}$$

$$(C9)\quad \frac{b = \texttt{true}}{\langle \texttt{while } b \texttt{ do } c, \rho, \sigma, \tau\rangle \rightarrow \langle c \texttt{ ; while } b \texttt{ do } c, \rho, \sigma, \tau\rangle} \qquad \frac{b = \texttt{false}}{\langle \texttt{while } b \texttt{ do } c, \rho, \sigma, \tau\rangle \rightarrow \langle \texttt{skip}, \rho, \sigma, \tau\rangle}\quad (C10)$$

Figure 4. Non-standard semantics of expressions and commands.

(iii) We use two abstract environments to manage the expressions: a first environment $\rho^\sharp\ :\ \mathsf{Lab} \rightarrow \wp(\mathsf{Expr}^\sharp)$ maps the labels attached to values to abstract expressions. This indicates how the value has been calculated. The second environment $\sigma^\sharp\ :\ \mathsf{Expr}^\sharp \rightarrow \mathbb{D}^\sharp$ maps abstract expressions to abstract values. The set $\mathbb{D}^\sharp$ denotes the abstract domain of floating-point numbers with error terms introduced in Section 3. The function $\sigma^\sharp(\eta)$ indicates the range of values in which are evaluated the expressions abstracted by η and encountered during the execution.

(iv) Intuitively, following the framework of semantics-based program transformations introduced in [6], our technique consists of, first, computing fully the abstract semantics. This semantics is non-deterministic because of Rule $(E4^\sharp)$ but the abstract expressions make the number of reductions polynomial. Next, using the information collected by the environments $\rho^\sharp$ and $\sigma^\sharp$, the result of each trace has a bound on the round-off errors and we select the best one. Finally, from the best abstract trace, we aim at building a new program, mathematically equivalent to the original one. We use the actions attached to the transitions: an action ω indicates which arithmetic operation has been performed at this step. Actions are used to rebuild a new program from a trace.

In summary, in the abstract semantics, we have rules of the form:

$$\langle e, \rho^\sharp, \sigma^\sharp, \tau^\sharp\rangle^{\ell} \xrightarrow{\omega}_k \langle e', \rho'^\sharp, \sigma'^\sharp, \tau^\sharp\rangle^{\ell'} \qquad (14)$$

where $\rho^\sharp, \sigma^\sharp, \tau^\sharp$ are environments, k is a user-defined parameter for the maximal height of the abstract expressions, ℓ and ℓ' are labels and ω is a sequence of actions. Actions and labels play a crucial role for the generation of the new program. They are presented in details in Section 6.

The abstract semantics resulting from the ideas detailed in the enumeration above is given in Figure 7. Its correctness is given, in the case of the floating-point arithmetic, in [18]. In Figure 7, $\bullet$ denotes one of the elementary operations `+`, `-` or `×` and the join operators $\sqcup$ are based on the order relations defined at the end of this section. The abstract environment for program variables maps a variable to a set of expressions:

$$\tau^\sharp\ :\ \mathsf{Var} \rightarrow \wp(\mathsf{Expr})$$

Note that $\tau^\sharp$ is not modified by the evaluation of an expression. Also note that Rule $(E2^\sharp)$ introduces (a limited amount of) non-determinism by allowing to substitute any expression $e \in \tau^\sharp(x)$ to the variable x.

The abstract semantics of commands is given in the bottom of Figure 7. It is a big step semantics whose rules have the form:

$$\langle c, \rho^\sharp, \sigma^\sharp, \tau^\sharp\rangle \xrightarrow{\ \omega\ }{}_k \langle c', \rho'^\sharp, \sigma'^\sharp, \tau'^\sharp\rangle \qquad (15)$$

Again, in Equation (15), ω denotes a sequence of actions. The rules $(C3^\sharp)$ to $(C5^\sharp)$ define the abstract semantics of control flow statements in a usual way, the actions on the transitions excepted. For a conditional, both branches are analyzed in the relevant environment. The environment $\tau^\sharp_{|b=\texttt{true}}$ (resp. $\tau^\sharp_{|b=\texttt{false}}$) is the restriction of the environment $\tau^\sharp$ to the cases which make the condition be true (resp. false). For a while loop, the body is executed in the relevant environment and the old and new environments are joined in order to run the next iteration. Note that the semantics requires that the sequence of actions ω corresponding to the execution of the body of the loop is the same at each iteration. With this condition, the same sequence of commands is executed at each iteration and it is possible to fold the loop when the transformed program is generated.

The rules $(C1^\sharp)$ and $(C2^\sharp)$ are abstract versions of the rules $(C1)$ and $(C2)$ of the non-standard semantics. Rule $(C2^\sharp)$ simply performs an eager step in the evaluation of the expression. Rule $(C1^\sharp)$ adds the current version of the expression e to the environ-

$$
(E1^\sharp)\quad \frac{\begin{array}{l} v^\sharp = \bigsqcup\left\{v'^\sharp \,:\, v'^\sharp = \left(\sigma^\sharp(\eta_1) \bullet^\sharp \sigma^\sharp(\eta_2)\right),\ \ \eta_1 \in \rho^\sharp(\ell_1),\ \eta_2 \in \rho^\sharp(\ell_2)\right\} \\ E = \bigsqcup\left\{\eta \,:\, \eta = \ulcorner \eta_1 \bullet \eta_2 \urcorner^k,\ \eta_1 \in \rho^\sharp(\ell_1),\ \eta_2 \in \rho^\sharp(\ell_2)\right\} \\ \sigma'^\sharp = \left| \begin{array}{l} \sigma^\sharp(\eta) \sqcup \left(\sigma^\sharp(\eta_1) \bullet \sigma^\sharp(\eta_2)\right) \text{ if } \eta_1 \in \rho^\sharp(\ell_1),\ \eta_2 \in \rho^\sharp(\ell_2),\ \eta = \ulcorner \eta_1 \bullet \eta_2 \urcorner^k \\ \sigma^\sharp(\eta) \text{ otherwise} \end{array}\right. \end{array}}{\langle v_1^{\ell_1} \bullet v_2^{\ell_2}, \rho^\sharp, \sigma^\sharp, \tau^\sharp\rangle^\ell \xrightarrow{\ \ell\, =\, \ell_1 \bullet \ell_2\ }_k \langle v^\ell, \rho^\sharp[\ell \mapsto \rho^\sharp(\ell) \cup E], \sigma'^\sharp, \tau^\sharp\rangle^\ell}
$$

$$
(E2^\sharp)\quad \frac{e \in \tau^\sharp(x) \quad \ell' \text{ fresh}}{\langle x, \rho^\sharp, \sigma^\sharp, \tau^\sharp\rangle^\ell \xrightarrow{\ x = \ell \rightsquigarrow \ell'\ }_k \langle e, \rho^\sharp, \sigma^\sharp, \tau^\sharp\rangle^{\ell'}}
$$

$$
(E3^\sharp)\quad \frac{\langle e_1, \rho^\sharp, \sigma^\sharp, \tau^\sharp\rangle^{\ell'} \xrightarrow{\omega}_k \langle e_1', \rho'^\sharp, \sigma'^\sharp, \tau^\sharp\rangle^{\ell'} \quad \ell' \text{ fresh}}{\langle e_1 \bullet e_2, \rho^\sharp, \sigma^\sharp, \tau^\sharp\rangle^\ell \xrightarrow{\omega}_k \langle e_1' \bullet e_2, \rho'^\sharp, \sigma'^\sharp, \tau^\sharp\rangle^\ell}
$$

$$
(E4^\sharp)\quad \frac{e \equiv_k e_1 \qquad \langle e_1, \rho^\sharp, \sigma^\sharp, \tau^\sharp\rangle^\ell \xrightarrow{\omega}_k \langle e_2, \rho'^\sharp, \sigma'^\sharp, \tau^\sharp\rangle^\ell \qquad e_2 \equiv_k e'}{\langle e, \rho^\sharp, \sigma^\sharp, \tau^\sharp\rangle^\ell \xrightarrow{\omega}_k \langle e', \rho'^\sharp, \sigma'^\sharp, \tau^\sharp\rangle^\ell}
$$

$$
(C1^\sharp)\quad \frac{\tau'^\sharp = \tau^\sharp\left[x \mapsto \tau^\sharp(x) \cup e\right] \quad \forall e' \in \tau^\sharp(x),\ e \not\equiv_k e'}{\langle x = e, \rho^\sharp, \sigma^\sharp, \tau^\sharp\rangle \xrightarrow{\ \texttt{skip}\ }_k \langle \rho^\sharp, \sigma^\sharp, \tau'^\sharp\rangle}
$$

$$
(C2^\sharp)\quad \frac{\langle e, \rho^\sharp, \sigma^\sharp, \tau^\sharp\rangle^x \xrightarrow{\ \ \omega\ \ }_k \langle e', \rho''^\sharp, \sigma''^\sharp, \tau^\sharp\rangle^\ell \quad \langle x = e', \rho''^\sharp, \sigma''^\sharp, \tau^\sharp\rangle \xrightarrow{\ \ \omega'\ \ }_k \langle \rho'^\sharp, \sigma'^\sharp, \tau'^\sharp\rangle}{\langle x = e, \rho^\sharp, \sigma^\sharp, \tau^\sharp\rangle \xrightarrow{\ \omega\ ;\ \omega'\ }_k \langle \rho'^\sharp, \sigma'^\sharp, \tau'^\sharp\rangle}
$$

$$
(C3^\sharp)\quad \frac{\langle c_1, \rho^\sharp, \sigma^\sharp, \tau^\sharp\rangle \xrightarrow{\ \ \omega_1\ \ }_k \langle \rho''^\sharp, \sigma''^\sharp, \tau''^\sharp\rangle \quad \langle c_2, \rho''^\sharp, \sigma''^\sharp, \tau''^\sharp\rangle \xrightarrow{\ \ \omega_2\ \ }_k \langle \rho'^\sharp, \sigma'^\sharp, \tau'^\sharp\rangle}{\langle c_1\ ;\ c_2, \rho^\sharp, \sigma^\sharp, \tau^\sharp\rangle \xrightarrow{\ \omega_1\ ;\ \omega_2\ }_k \langle \rho'^\sharp, \sigma'^\sharp, \tau'^\sharp\rangle}
$$

$$
(C4^\sharp)\quad \frac{\langle c_1, \rho^\sharp_{|b=\texttt{true}}, \sigma^\sharp_{|b=\texttt{true}}, \tau^\sharp_{|b=\texttt{true}}\rangle \xrightarrow{\ \ \omega_1\ \ }_k \langle \rho_1^\sharp, \sigma_1^\sharp, \tau_1^\sharp\rangle \quad \langle c_2, \rho^\sharp_{|b=\texttt{false}}, \sigma^\sharp_{|b=\texttt{false}}, \tau^\sharp_{|b=\texttt{false}}\rangle \xrightarrow{\ \ \omega_2\ \ }_k \langle \rho_2^\sharp, \sigma_2^\sharp, \tau_2^\sharp\rangle}{\langle \texttt{if}\ b\ c_1\ c_2, \rho^\sharp, \sigma^\sharp, \tau^\sharp\rangle \xrightarrow{\ \texttt{if}\ b\ \omega_1\ \omega_2\ }_k \langle \rho_1^\sharp \sqcup \rho_2^\sharp, \sigma_1^\sharp \sqcup \sigma_2^\sharp, \tau_1^\sharp \sqcup \tau_2^\sharp\rangle}
$$

$$
(C5^\sharp)\quad \frac{\begin{array}{c} \langle c, \rho^\sharp_{|b=\texttt{true}}, \sigma^\sharp_{|b=\texttt{true}}, \tau^\sharp_{|b=\texttt{true}}\rangle \xrightarrow{\ \ \omega\ \ }_k \langle \rho''^\sharp, \sigma''^\sharp, \tau''^\sharp\rangle \\ \langle \texttt{while}\ b\ \texttt{do}\ c, \rho''^\sharp \sqcup \rho^\sharp_{|b=\texttt{true}}, \sigma''^\sharp \sqcup \sigma^\sharp_{|b=\texttt{true}}, \tau''^\sharp \sqcup \tau^\sharp_{|b=\texttt{true}}\rangle \xrightarrow{\ \texttt{while}\ b\ \texttt{do}\ \omega\ }_k \langle \rho'^\sharp, \sigma'^\sharp, \tau'^\sharp\rangle \end{array}}{\langle \texttt{while}\ b\ \texttt{do}\ c, \rho^\sharp, \sigma^\sharp, \tau^\sharp\rangle \xrightarrow{\ \texttt{while}\ b\ \texttt{do}\ \omega\ }_k \langle \rho'^\sharp \sqcup \rho^\sharp_{|b=\texttt{false}}, \sigma'^\sharp \sqcup \sigma^\sharp_{|b=\texttt{false}}, \tau'^\sharp \sqcup \tau^\sharp_{|b=\texttt{false}}\rangle}
$$

Figure 7. Abstract semantics of expressions and commands.

ment $\tau^\sharp$ if and only if e is not in relation by $\equiv_k$ with an expression already present in $\tau^\sharp(x)$.

Concerning the arithmetic expressions, the safety of the abstract semantics with respect to the non-standard semantics has been proved in [18]. In the rest of this section, we focus on the correctness of the abstract semantics of commands. First, we relate the environments used in the non-standard semantics and in the abstract semantics by the Galois connections:

$$
\langle \wp(\mathsf{Env}_\rho), \subseteq\rangle \underset{\alpha_k^\rho}{\overset{\gamma_k^\rho}{\leftrightarrows}} \langle \mathsf{Env}^\sharp_{\rho,k}, \sqsubseteq_\rho\rangle, \tag{16}
$$

$$
\langle \wp(\mathsf{Env}_\sigma), \subseteq\rangle \underset{\alpha_k^\sigma}{\overset{\gamma_k^\sigma}{\leftrightarrows}} \langle \mathsf{Env}^\sharp_{\sigma,k}, \sqsubseteq_\sigma\rangle, \tag{17}
$$

$$
\langle \wp(\mathsf{Env}_\tau), \subseteq\rangle \underset{\alpha_k^\tau}{\overset{\gamma_k^\tau}{\leftrightarrows}} \langle \mathsf{Env}^\sharp_{\tau,k}, \sqsubseteq_\tau\rangle. \tag{18}
$$

The partial order as well as the abstraction and concretization functions for the first kind of environments are defined by:

$$
\rho_1^\sharp \sqsubseteq_\rho \rho_2^\sharp \iff \forall \ell \in \mathsf{Dom}(\rho_1^\sharp),\ \rho_1^\sharp(\ell) \subseteq \rho_2^\sharp(\ell), \tag{19}
$$

$$
\alpha_k^\rho(R) = \rho^\sharp \ :\ \forall \ell \in \mathcal{L},\ \rho^\sharp(\ell) = \cup_{\rho \in R} \ulcorner \rho(\ell) \urcorner^k, \tag{20}
$$

$$
\gamma_k^\rho(\rho^\sharp) = \{\rho \in \mathsf{Env}_\rho \ :\ \forall \ell \in \mathcal{L},\ \ulcorner \rho(\ell) \urcorner^k \in \rho^\sharp(\ell)\}. \tag{21}
$$

The environment $\rho_1^\sharp$ is smaller than $\rho_2^\sharp$ if, for any label ℓ, the set $\rho_1^\sharp(\ell)$ is a subset of $\rho_2^\sharp(\ell)$. The abstraction $\alpha_k^\rho(R)$ of a set $R = \{\rho_1, \rho_2, \ldots, \rho_n\}$ of environments is the abstract environment $\rho^\sharp$ which maps any label ℓ to the set of abstract expressions $\ulcorner e \urcorner^k$ such that $\rho_i(\ell) = e$ for some $1 \le i \le n$. Conversely, γ_k^ρ is the set of environments ρ which map ℓ to an expression e such that $\ulcorner e \urcorner^k = \rho^\sharp(\ell)$. Similarly, we have for the second kind of environments:

$$
\sigma_1^\sharp \sqsubseteq_\sigma \sigma_2^\sharp \iff \forall \eta \in \mathsf{Dom}(\sigma_1^\sharp),\ \sigma_1^\sharp(\eta) \sqsubseteq^\sharp_{\mathbb{D}} \sigma_2^\sharp(\eta), \tag{22}
$$

$$
\alpha_k^\sigma(S) = \sigma^\sharp \ :\ \forall \eta \in \mathsf{Expr}_k^\sharp,\ \sigma^\sharp(\eta) = \alpha(\{\sigma(\eta),\ \sigma \in S\}), \tag{23}
$$

$$
\gamma_k^\sigma(\sigma^\sharp) = \{\sigma \in \mathsf{Env}_\sigma \ :\ \forall e \in \mathsf{Expr},\ \sigma(e) \in \gamma(\sigma^\sharp(\ulcorner e \urcorner^k))\}. \tag{24}
$$

The environment $\sigma_1^\sharp$ is smaller than $\sigma_2^\sharp$ if $\sigma_1^\sharp$ maps any abstract expression η to an abstract value smaller than $\sigma_2^\sharp(\eta)$. The order $\sqsubseteq^\sharp_{\mathbb{D}}$ is the component-wise order on the abstract domain of values $\mathbb{D}^\sharp$. The abstraction α_k^σ and concretization γ_k^σ are based on the Galois connection mentioned in Section 3 to relate concrete and abstract values.

Finally, for the third kind of environments, we have:

$$
\tau_1^\sharp \sqsubseteq_\tau \tau_2^\sharp \iff \forall x \in \mathsf{Dom}(\tau_1^\sharp),\ \tau_1^\sharp(x) \sqsubseteq^\sharp_{\mathbb{E}} \tau_2^\sharp(x), \tag{25}
$$

$$
\alpha_k^\tau(T) = \tau^\sharp \ :\ \forall x \in \mathsf{Var},\ \tau^\sharp(x) = \cup_{\tau \in T} \ulcorner \tau(x) \urcorner^k, \tag{26}
$$

$$
\gamma_k^\tau(\tau^\sharp) = \{\tau \in \mathsf{Env}_\tau \ :\ \forall x \in \mathsf{Var},\ \ulcorner \tau(x) \urcorner^k \in \tau^\sharp(x)\}. \tag{27}
$$

The correctness of the abstract semantics is based on the following proposition.

PROPOSITION 1. *Let* $\langle c,\rho,\sigma,\tau\rangle \longrightarrow^n \langle \mathit{skip},\rho',\sigma',\tau'\rangle$ *denote a sequence of reduction steps of length* n *in the non-standard semantics. Then, if* $\alpha_k^\rho(\rho) \sqsubseteq_\rho \rho^\sharp$, $\alpha_k^\sigma(\sigma) \sqsubseteq_\sigma \sigma^\sharp$ *and* $\alpha_k^\tau(\tau) \sqsubseteq_\tau \tau^\sharp$ *then*

$$\langle c,\rho^\sharp,\sigma^\sharp,\tau^\sharp\rangle \xrightarrow{\ \omega\ }_k \langle \rho'^\sharp,\sigma'^\sharp,\tau'^\sharp\rangle$$

such that $\alpha_k^\rho(\rho') \sqsubseteq_\rho \rho'^\sharp$, $\alpha_k^\sigma(\sigma') \sqsubseteq_\sigma \sigma'^\sharp$ *and* $\alpha_k^\tau(\tau') \sqsubseteq_\tau \tau'^\sharp$.

Because of the abstract expressions, for a given program, the number of reduction paths in the abstract semantics is polynomial of degree k, where k is the user parameter used in the function $\ulcorner . \urcorner^k$.

6. Program Transformation

The code transformation consists of generating a new program from the best abstract trace. For an arithmetic expression, the best trace is the trace which minimizes the greatest round-off error [18]. However, for programs with many variables, the way to determine the best trace is not unique and it may depend on user requirements. In particular, we have to consider the following alternatives:

- the measure of the errors may concern either all the variables of the program or a (possibly user-defined) subset of them, for example the only variables storing the final results of a computation,
- for a given set of relevant variables, different measures can be used: we may either minimize the worst error associated to variables, or minimize the mean error, or use a least square method.

These choices modify significantly the results of the transformation but they introduce no theoretical difficulty and we consider them rather as implementation details.

Given a measure $\prec^\sharp$ of the quality of traces as discussed in the previous paragraph, the transformation is based on the minimal abstract trace $\langle c,\rho^\sharp,\sigma^\sharp,\tau^\sharp\rangle \xrightarrow{\ \omega\ }_k \langle \rho'^\sharp,\sigma'^\sharp,\tau'^\sharp\rangle$, i.e. the trace which yields the minimal state in the sense of $\prec^\sharp$. Remark that, since $\xrightarrow{\omega}_k$ uses abstract values of $\mathbb{D}^\sharp$, the transformation minimizes the worst error ε which may occurs during an evaluation. In other words, it minimizes the precision lost which may arise during an evaluation in the worst case, that is for the most pessimistic combination of data.

$$(G1)\quad \mathbf{P}[\![(\ell = \ell_1 \bullet \ell_2,\iota)]\!] = (\iota(\ell) = \iota(\ell_1)\bullet\iota(\ell_2),\iota)$$

$$(G2)\quad \mathbf{P}[\![(x = \ell \rightsquigarrow \ell',\iota)]\!] = (\mathtt{skip},\iota[\ell'\mapsto x,\ell\mapsto x])$$

$$(G3)\quad \mathbf{P}[\![(\omega_1 \ ;\ \omega_2,\iota)]\!] = \left|\begin{array}{l}\mathtt{let}\ (c_1,\iota_1) = \mathbf{P}[\![(\omega_1,\iota)]\!]\\ \mathtt{and}\ (c_2,\iota') = \mathbf{P}[\![(\omega_2,\iota_1)]\!]\\ \mathtt{in}\ (c_1\ ;\ c_2,\iota')\end{array}\right.$$

$$(G4)\quad \mathbf{P}[\![(\mathtt{if}\ b\ \omega_1\ \omega_2,\iota)]\!] = \left|\begin{array}{l}\mathtt{let}\ (c_1,\iota_1) = \mathbf{P}[\![(\omega_1,\iota)]\!]\\ \mathtt{and}\ (c_2,\iota') = \mathbf{P}[\![(\omega_2,\iota_1)]\!]\\ \mathtt{in}\ (\mathtt{if}\ b\ c_1\ c_2, i')\end{array}\right.$$

$$(G5)\quad \mathbf{P}[\![(\mathtt{while}\ b\ \mathtt{do}\ \omega,\iota)]\!] = \left|\begin{array}{l}\mathtt{let}\ (c,\iota') = \mathbf{P}[\![(\omega,\iota)]\!]\\ \mathtt{in}\ (\mathtt{while}\ b\ \mathtt{do}\ c',\iota')\end{array}\right.$$

Figure 8. Generation of the new program.

The key point of the code transformation is the sequences of actions attached to the transitions of the abstract semantics. They are generated by the grammar:

$$\omega \ ::= \ \ell = \ell_1 \bullet \ell_2 \mid x = \ell \rightsquigarrow \ell' \mid \omega_1\ ;\ \omega_2 \mid \mathtt{if}\ b\ \omega_1\ \omega_2 \mid \mathtt{while}\ b\ \mathtt{do}\ \omega \qquad (28)$$

The actions indicate which operation has been executed during a transition. The action $\ell = \ell_1 \bullet \ell_2$, where $\bullet \in \{+,\ -,\ \times\}$, means that an operation between the values labeled ℓ_1 and ℓ_2 is performed and that the result is assigned to ℓ. The labels belong to the set Lab. The label ℓ attached to a state $\langle e,\rho,\sigma,\tau\rangle^\ell$ indicates that the result of the evaluation of the expression e has to be assigned to the label ℓ. In the abstract semantics of expressions, this label is only modified by Rule $(E2^\sharp)$: when a variable is met, we stop evaluating the current expression and we evaluate the expression assigned to x instead. In addition, the action $x = \ell \rightsquigarrow \ell'$ is inserted to create indirections by letting ℓ and ℓ' point to x.

For a given abstract trace $\langle c,\rho^\sharp,\sigma^\sharp,\tau^\sharp\rangle \xrightarrow{\ \omega\ }_k \langle \rho'^\sharp,\ \sigma'^\sharp,\ \tau'^\sharp\rangle$, the code transformation $\mathbf{P}[\![(\omega,\iota)]\!]$, given in Figure 8, is based on the sequence ω of actions and on the environment

$$\iota\ :\ \mathsf{Lab}\cup\mathsf{Var}\to\mathsf{Expr} \qquad (29)$$

which maps a label or variable to either another label or a variable or an expression. We assume that, initially, for any value v^ℓ occurring in the program, $\iota(\ell) = v$.

Concerning the arithmetic expressions, for the sake of simplicity, the code transformation generates a sequence of elementary operations, using the labels as auxiliary variables. In the abstract semantics, an action $\ell = \ell_1 \bullet \ell_2$ arises in Rule $(E1^\sharp)$, when an operation between two values is performed. In this case, $\mathbf{P}$ generates an assignment

$$\ell = \iota(\ell_1)\bullet\iota(\ell)$$

where ℓ is a variable named as the label. The environment ι transforms labels into expressions. For example, if v^{ℓ_1} is a value occurring in the original program and if $\iota(\ell_2) = \ell_3$ then ℓ_3 is a variable named as some label and $\mathbf{P}[\![(\ell = \ell_1 + \ell_2,\iota)]\!] = v + \ell_3$ where $v + \ell_3$ is the syntax of the arithmetic expression adding the value v to the variable ℓ_3.

Recall from Section 5 that an action $x = \ell \rightsquigarrow \ell'$ arises in the abstract semantics in Rule $(E2^\sharp)$, when a variable is read:

$$(E2^\sharp)\quad \frac{e\in\tau^\sharp(x)\quad \ell' \text{ fresh}}{\langle x,\rho^\sharp,\sigma^\sharp,\tau^\sharp\rangle^\ell \xrightarrow{x=\ell\rightsquigarrow\ell'}_k \langle e,\rho^\sharp,\sigma^\sharp,\tau^\sharp\rangle^{\ell'}}$$

This rule works together with Rule $(C2^\sharp)$ which handles the eager assignments:

$$(C2^\sharp)\quad \frac{\langle e,\rho^\sharp,\sigma^\sharp,\tau^\sharp\rangle^x \xrightarrow{\ \omega\ }_k \langle e',\rho''^\sharp,\sigma''^\sharp,\tau^\sharp\rangle^\ell \qquad \langle x = e',\rho''^\sharp,\sigma''^\sharp,\tau^\sharp\rangle \xrightarrow{\ \omega'\ }_k \langle \rho'^\sharp,\sigma'^\sharp,\tau'^\sharp\rangle}{\langle x = e,\rho^\sharp,\sigma^\sharp,\tau^\sharp\rangle \xrightarrow{\ \omega\ ;\ \omega'\ }_k \langle \rho'^\sharp,\sigma'^\sharp,\tau'^\sharp\rangle}$$

At program generation time, when the action $x = \ell \rightsquigarrow \ell'$ is encountered, no command is generated by Rule $(G2)$ but `skip`. Instead, indirections are introduced in the environment ι: the label ℓ is the label attached to the source state of the transition. Because of the labels of the states used in the first premise of $(C2^\sharp)$, the label ℓ of the indirection indicates which variable was being computed and the computation as lead to a variable x. So we set $\iota(\ell) = x$ and $\iota(\ell') = x$. Then, later, when ℓ will be accessed the value of x will be returned. Similarly, instead of being assigned to ℓ', the expression will be assigned to x. For example, assuming that $\iota(x) = 7 \times 5$, the expression $y = x + 3$ will lead to the commands $x = (7 \times 5) + 3\ ;\ y = x$.

The other rules, named $(G3)$, $(G4)$ and $(G5)$ concern the control-flow structures and are more classical. Rule $(G3)$ deals with sequences and Rule $(G4)$ with conditionals. In Rule $(G5)$, the body of the loop is built from the sequence ω corresponding to the actions of the abstract semantics. Note that, Rule $(C5^\sharp)$ requires that the same the sequence of actions ω is performed at each iteration.

We end this section by discussing the correctness of the code transformation. We show that, at an observational level [6], the semantics of the original program c and the semantics of the trans-

formed program c_t are equal. Our observation consists of showing that, for a set of observational variables $\mathcal{V}_\mathcal{O} \subseteq \mathsf{Var}$, c and c_t compute the same thing in the exact arithmetic of real numbers.

Let $\alpha_\mathcal{O}$ be an observational abstraction $\alpha_\mathcal{O} : \mathbb{D} \to \mathbb{R}$ which transforms a floating-point number with errors $(x, \varepsilon) \in \mathbb{D}$ into a real number, i.e. $\alpha_\mathcal{O}(x, \varepsilon) = x + \varepsilon$. We first introduce a proposition concerning the non-standard semantics.

PROPOSITION 2. *Let c be a command and let*

$$\langle c, \rho, \sigma, \tau \rangle \longrightarrow^* \langle \textit{skip}, \rho', \sigma', \tau' \rangle$$

and

$$\langle c, \rho, \sigma, \tau \rangle \longrightarrow^* \langle \textit{skip}, \rho'', \sigma'', \tau'' \rangle$$

be two paths of the non-standard semantics. Then for any variable $x \in \mathcal{V}_\mathcal{O}$, $\alpha_\mathcal{O}(\tau'(x)) = \alpha_\mathcal{O}(\tau''(x))$.

Proposition 2 stems from the fact that, in $\mathbb{D}$, the errors are exactly computed. So, from the perspective of $\alpha_\mathcal{O}$, the traces of $\to_\mathbb{D}$ are identical to the traces of $\to_\mathbb{R}$.

PROPOSITION 3. *Let* $c_t = \mathbf{P}[\![(\omega, \iota)]\!]$ *for some trace*

$$\langle c, \rho^\sharp, \sigma^\sharp, \tau^\sharp \rangle \xrightarrow{\ \ \omega\ \ }_k \langle \rho'^\sharp, \sigma'^\sharp, \tau'^\sharp \rangle.$$

Then for all $\rho \in \gamma_k^\rho(\rho^\sharp)$, $\sigma \in \gamma_k^\sigma(\sigma^\sharp)$ *and* $\tau \in \gamma_k^\tau(\tau^\sharp)$, *if*

$$\langle c, \rho, \sigma, \tau \rangle \longrightarrow^* \langle \textit{skip}, \rho', \sigma', \tau' \rangle$$

and

$$\langle c_t, \rho, \sigma, \tau \rangle \longrightarrow^* \langle \textit{skip}, \rho'', \sigma'', \tau'' \rangle$$

then for any variable $x \in \mathcal{V}_\mathcal{O}$, $\alpha_\mathcal{O}(\tau'(x)) = \alpha_\mathcal{O}(\tau''(x))$.

As a consequence, in the non-standard semantics c and c_t lead to observationally equivalent environments. In particular, this property holds for the minimal trace chosen in order to generate c_t.

7. Experiments

In this section, we show how our program transformation for numerical precision works on some examples. We also discuss how the set of rules defining the relation $\equiv$ between arithmetic expressions can be chosen, in order to allow some transformations useful in practice.

First, we reconsider the program introduced in Section 2 to compute the sum of decreasing terms. Let p denote the original program and p_t the transformed program. Assuming that $x_i = [2^{\mathtt{N}-i-1}, 2^{\mathtt{N}-i}]$, we obtain, for different values of N, the error bounds given in Figure 9. The evolution of both error bounds is also given by the curve of Figure 9. We can see how the transformation improves the quality of the computation.

A variant of the previous example is the well-known Patriot bug [14]: a counter has been repeatedly incremented by 0.1 for a long time. It is well-known that the constant 0.1 is not representable exactly in binary and, as a consequence, the counter diverges. However, at compile-time, a program transformer could detect that the mathematical constant 0.1 cannot be stored exactly in the formats allowed by the IEEE754 Standard. Assuming that the body of the loop has been unrolled at least 5 times, by extending the relation $\equiv$ which governs the program transformation, we could detect that by adding 0.1 five times we obtain 0.5 which is a power of 2, exactly represented in machine. We could obtain a program with the body of the loop unrolled five times, and 0.5 added to the counter. In order to obtain this result, the relation $\equiv$ has to be augmented. Because it is not possible to add the rules $a \bullet b \equiv c$ for every triple (a, b, c) and for every operation $\bullet \in \{ +, -, \times \}$ such that $a \bullet b = c$ in real arithmetic, it is necessary to use a dynamic technique which computes, for any operation $\bullet$ between some constants a and b of the program, the exact result c and adds the rule $a \bullet b \equiv c$ to the relation of Figure 5.

Case	Error for p	Error for p_t
N=4	[0.0,2.861022950E-6]	[0.0,2.145767212E-6]
N=6	[0.0,1.907348633E-5]	[0.0,1.263618470E-5]
N=8	[0.0,1.068115235E-4]	[0.0,6.604194642E-5]
N=10	[0.0,5.493164063E-4]	[0.0,3.254413605E-4]

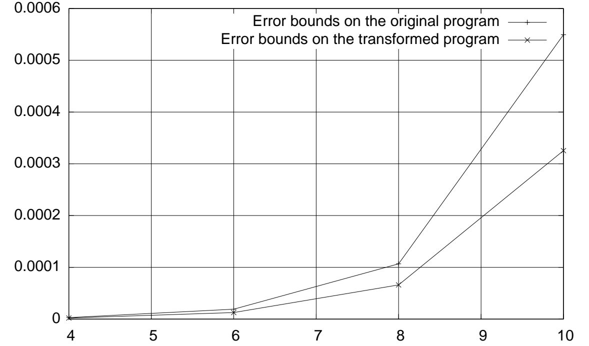

Figure 9. Evolution of the error bounds for the program adding decreasing numbers.

```
float a=[0.0,100.0];         float a=[0.0,100.0];
float b=[0.0,0.001];         float b=[0.0,0.001];
float c=[5.0,10.0];          float c=[5.0,10.0];
float d=2.0;                 float d=2.0;
if (a>0.1) {           ⟶     if (a>0.1) {
  r=d*(a+(b+c));               r=d*(a+(b+c));
  ...                          ...
} else {                     } else {
  r=d*(a+(b+c));               r=d*((a+b)+c);
  ...                          ...
}                            }
```

Case	Error bound on r
$p,\ a > 0.1$	[-8.583068847E-6,-7.629394532E-6]
$p_t,\ a \le 0.1$	[-1.907348632E-6,-9.536743165E-7]
$p,\ a > 0.1$	[-8.583068847E-6,-7.629394532E-6]
$p_t,\ a \le 0.1$	[-9.611248970E-7,-9.536743165E-7]

Figure 10. Example of program transformation with conditionals.

Our next example shows that, depending on the data, different expressions can be selected. We consider the program given in the left-hand side of Figure 10. The precision of the evaluation of the expression r depends on the parsing of the sum $a + b + c$. Our transformation uses the test $a > 0.1$ to restrict the environments in both branches. As a consequence, different parsings are selected in the branches of the conditionals in the transformed program. The precision in all the branches is given in the table of Figure 10.

We could add more complex rules to the relation $\equiv$ in order to generalize what we did in the previous example. Some tests which are not written in the original code could be added in the transformed program, in order to obtain more precise formulas depending on the data. However this technique cannot be applied systematically since it would lead to a combinatorial explosion. This technique can also be related to program specialization (by partial evaluation) where a program is optimized for certain data.

8. Conclusion

In this article, we have defined a code transformation to enhance the numerical precision of floating-point computations. This work

extends previous work on arithmetic expressions to the case of full programs with assignments, conditional and loops. The transformed program may assign to the variables expressions which are very different from the original ones while still computing the same mathematical formulas. The quality of the transformation depends on a user-defined parameter corresponding to the height of the abstract expressions.

In [19], we have shown that the transformation of expressions can be extended to other arithmetic, like the fixed-point arithmetic. This result can be generalized to full programs, for which the techniques introduced in this article can be applied to the case of fixed-point computations.

We believe that it would be very interesting to mix our transformation for numerical precision with other program transformations. First, our transformation can go against usual compiler optimizations which attempt to speedup the execution-time of programs and combined techniques should be studied. Second, we could use our transformation to specialize numerical programs, in the sense of partial evaluation [12], with respect to certain sets of inputs: depending on the data, we chose some parsing of a formula or another. The connections with partial evaluation should be studied more deeply. Finally, other transformations could be mixed, like code watermarking and obfuscation for which round-off errors on floating-point computation could be used, e.g. to encode a watermark [7].

The relation $\equiv$ used in this article identifies expressions which are equal in the reals. These laws enable us to rewrite arithmetic expressions. However, the relation $\equiv$ is not unique and could be extended by many other laws, as shown in Section 7. For example, some laws can be used to improve the precision of floating-point computations, like Sterbenz's theorem for subtraction [23]. In the case of fixed-point arithmetic, yet another set of rules is probably needed. The study of the most useful sets of rules for the most common computer arithmetic is one of our current work direction.

In the future, we also aim at implementing a program transformer able to handle large codes. This tool should work both in floating-point and fixed-point arithmetic since there are many industrial needs in both contexts.

References

[1] ANSI/IEEE. *IEEE Standard for Binary Floating-point Arithmetic*, std 754-1985 edition, 1985.

[2] David F. Bacon, Graham Susan L., and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, 1994.

[3] Christian Bischof, Paul D. Hovland, and Boyana Norris. Implementation of automatic differentiation tools. In *Partial Evaluation and Semantics-Based Program Transformations, PEPM'02*. ACM Press, 2002.

[4] M. Ceberio and V. Kreinovich. Greedy algorithms for optimizing multivariate horner schemes. *ACM-SIGSAM Bulletin*, 38(1):8–15, 2004.

[5] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximations of fixed points. In *Principles of Programming Languages 4*, pages 238–252. ACM Press, 1977.

[6] P. Cousot and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *Conference Record of the Twentyninth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 178–190, Portland, Oregon, 2002. ACM Press, New York, NY.

[7] M. Dalla Preda and R. Giacobazzi. Control code obfuscation by abstract interpretation. In *International Conference on Software Engineering and Formal Methods, SEFM'05*, pages 301–310. IEEE Computer Society Press, 2005.

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

[9] E. Goubault. Static analyses of the precision of floating-point operations. In *Static Analysis Symposium, SAS'01*, number 2126 in Lecture Notes in Computer Science, pages 234–259. Springer-Verlag, 2001.

[10] E. Goubault, M. Martel, and S. Putot. Asserting the precision of floating-point computations: a simple abstract interpreter. In 11th *European Symposium on Programming, ESOP'02*, number 2305 in Lecture Notes in Computer Science, pages 209–212, 2002.

[11] E. Goubault, M. Martel, and S. Putot. Some future challenges in the validation of control systems. In *Proceedings of the European Congress on Embedded Real Time Software (ERTS'06)*, 2006.

[12] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, Int. Series in Computer Science, 1993.

[13] P. Langlois and N. Louvet. How to ensure a faithful polynomial evaluation with the compensated Horner algorithm? In P. Kornerup and J.-M. Muller, editors, *18th IEEE International Symposium on Computer Arithmetic*, pages 141–149. IEEE Computer Society, June 2007.

[14] Information Management and Technology Division. Patriot missile defense : Software problem led to system failure in Dhahran, saudi arabia. Technical Report B-247-094, United State General Accounting Office, 1992.

[15] M. Martel. Propagation of roundoff errors in finite precision computations: a semantics approach. In 11th *European Symposium on Programming, ESOP'02*, number 2305 in Lecture Notes in Computer Science, pages 194–208. Springer-Verlag, 2002.

[16] M. Martel. An overview of semantics for the validation of numerical programs. In *Verification, Model Checking and Abstract Interpretation, VMCAI'05*, number 3385 in Lecture Notes in Computer Science, pages 59–77. Springer-Verlag, 2005.

[17] M. Martel. Semantics of roundoff error propagation in finite precision calculations. *Journal of Higher Order and Symbolic Computation*, 19:7–30, 2006.

[18] M. Martel. Semantics-based transformation of arithmetic expressions. In *Static Analysis Symposium, SAS'07*, number 4634 in Lecture Notes in Computer Science. Springer-Verlag, 2007.

[19] M. Martel. Enhancing the implementation of mathematical formulas for fixed-point and floating-point arithmetics. In *First International Workshop on Numerical Abstractions for Software Verification, NSV'08*, 2008.

[20] D. Monniaux. The pitfalls of verifying floating-point computations. *TOPLAS*, 30(3), May 2008.

[21] R. Rocher, D. Menard, N. Herve, and Sentieys. Fixed-point configurable hardware components. *EURASIP Journal on Embedded Systems (JES)*, 2006, 2006.

[22] R. Rocher, D. Menard, O. Sentieys, and P. Scalart. Analytical accuracy evaluation of fixed-point systems. In *EUSIPCO'07Poznan, Pologne*, 2007.

[23] P. H. Sterbenz. *Floating-point Computation*. Prentice Hall International, 1974.