Available online at www.sciencedirect.com

SciVerse ScienceDirect

Electronic Notes in Theoretical Computer Science 287 (2012) 3–16

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Accurate Evaluation of Arithmetic Expressions (Invited Talk)

Matthieu Martel[a,b,c,1]

[a] *Université de Perpignan Via Domitia, Digits, Architectures et Logiciels Informatiques*
52 avenue Paul Alduy, F-66860, Perpignan, France

[b] *Université Montpellier II*
Laboratoire d'Informatique Robotique et de Microélectronique de Montpellier
UMR 5506, 161 rue Ada, F-34095, Montpellier, France

[c] *CNRS, Laboratoire d'Informatique Robotique et de Microélectronique de Montpellier*
UMR 5506, 161 rue Ada, F-34095, Montpellier, France

Abstract

In this article, we focus on the synthesis of arithmetic expressions that can be evaluated efficiently on computers in the sense that they do not create overflows, are accurate and do not use unnecessary ressources. We consider several computer arithmetics for integers, floating-point and fixed-point numbers and intervals and we show how to synthetize new expressions, mathematically equivalent to the original ones and more efficient. Our approach is based on abstract interpretation. We introduce two abstractions to represent in polynomial size sets of mathematically equivalent expressions. Then, we extract optimized expressions by searching the most accurate expression among the expressions contained in the abstract structures. We focus on the correctness of the synthesis which consists of showing that the new expressions cannot be distinguished from the source expressions when an observational abstraction is used.

Keywords: Abstract Interpretation, Code Synthesis, Computer Arithmetic.

1 Introduction

During the last decade, static analysis techniques based on abstract interpretation [2] have reached an industrial level of maturity. Tools like Astrée [8], Clousot [9] or Fluctuat [4] have been successfully used on real case studies. These tools are able to compute subtle properties on codes such as accurate ranges for floating-point variables. However, when a run-time error or an unexpected behavior of the program is detected, these tools do not indicate how to fix the code. Then, a natural extension of this work is to propose bug corrections to the programmer. Such techniques have

[1] Email:matthieu.martel@univ-perp.fr

doi:10.1016/j.entcs.2012.09.002

recently been proposed to repair integer expressions and relations between integer expressions [10] and to improve the accuracy of floating-point expressions [11].

In this article, we focus on the synthesis of expressions well-suited for the computer arithmetic in the sense that their evaluation by the machine is efficient, accurate and does not rise run-time errors. We consider that the expressions written in source codes by programmers are mathematical formulas which would return the expected results if the computers used exact arithmetics (with mathematical integer or real numbers). Then we synthesize new expressions which are mathematically equivalent to the original ones and whose evaluation in the computer arithmetic raises less errors. We consider four computer arithmetics: the integer arithmetic is subject to overflows which can be avoided in certain cases is the expressions are transformed [10]. In addition to overflows, the floating-point arithmetic [1,14] is subject to accuracy problems due to the rounding of the operations. The accuracy of expressions can be improved if the expressions are replaced by mathematically equivalent ones [11,6]. In the fixed-point arithmetic [5], the evaluation of equivalent versions of an expression may require more or less ressources depending on the size of the intermediary results. Finally, the interval arithmetic [12] introduces over-approximations because of the lack of relations between variables. In many cases, these over-approximations can be limited by transformation of the source expression.

Our work is based on P. and R. Cousot's framework for program transformation [3]. We introduce non-standard semantics and an observational abstraction for the integer, floating-point, fixed-point and interval arithmetics. The synthesis is correct if it generates a new expression which cannot be distinguished from the source expression when the observational abstraction is used. Because in general there exists too many expressions mathematically equivalent to a source expression, we present two abstractions of the sets of equivalent expressions [11,6]. Finally, the synthesis consists of selecting a expression among the abstract sets of equivalent expressions. This selection is based on an abstract interpretation of the non-standard semantics introduced for the four arithmetics.

This article is organized as follows. In Section 2, we introduce the computer arithmetic and, for each of them, we discuss how the synthesis of a new equivalent expression may improve the evaluation. Section 3 is dedicated to the correctness of the synthesis. It introduces the non-stndard semantics and obsevational abstractions. In Section 4, we describe the abstraction of sets of equivalent expressions. Finally, the synthesis itself is described in Section 5 and Section 6 concludes.

2 Computer Arithmetics and Expression Synthesis

In this section, we review several computer arithmetics and, for each of them, we present how to synthesize expressions that can be evaluated efficiently. We start with our simpliest arithmetic which is the integer arithmetic. Then we discuss the cases of the floating-point and fixed-point arithmetics and, finally, we end this section by examining the case of interval arithmetic.

The signed integer arithmetic enables one to represent exactly any integer number between a minimal value $\mathbf{m}$ and a maximal value $\mathbf{M}$. For example, in many languages, the `int` format corresponds to the integer numbers between $\mathbf{m} = -2^{31}$ and $\mathbf{M} = 2^{31} - 1$. When the result of some operation is out of the interval $[\mathbf{m}, \mathbf{M}]$ then it wraps around this interval. This is a source of error in programs, as noted by F. Logozzo and T. Ball who propose code repairs for integer expressions and relations [10]. Formally, the elementary operations are defined by:

$$a \oplus b = \begin{cases} \mathbf{m} \oplus (a + b - (\mathbf{M} + 1)) \text{ if } a + b > \mathbf{M} \\ \mathbf{M} \oplus (a + b - (\mathbf{m} - 1)) \text{ if } a + b < \mathbf{m} \\ a + b \quad \text{otherwise} \end{cases} \qquad a \otimes b = \begin{cases} a \oplus (a \otimes (b - 1)) \text{ if } b \geq 1 \\ -a \oplus (a \otimes (-b - 1)) \text{ if } b < 1 \end{cases} \tag{1}$$

For example, with 32 bits signed integers, if $x = 2^{30}$ and $y = -2^{15}$ then

$$\frac{2 \times x}{3} + y = -715860650 \quad \text{and} \quad 2 \times \frac{x}{3} + y = 715795114.$$

Both expressions are mathematically equivalent while the closest integer to the exact result is 715795115. Because, in the signed integer arithmetic, errors may arise when intermediary results are out of the range $[\mathbf{m}, \mathbf{M}]$, the synthesis of expressions has to generate expressions which are mathematically equivalent to the original ones and whose evaluation introduces the smallest intermediary results, in absolute value.

The floating-point arithmetic is defined by the IEEE754 Standard [1]. Floating-point numbers are used to encode real numbers. However, because they are finite representations of they mathematical cousins, roundoff errors arise during computations and these approximations may, in some cases, significantly falsify the result of the evaluation. A floating-point number x is defined by

$$x = s \cdot (d_0.d_1 \dots d_{p-1}) \cdot \beta^e = s \cdot m \cdot \beta^{e-p+1} \tag{2}$$

where $s \in \{-1, 1\}$ is the sign, $m = d_0 d_1 \dots d_{p-1}$ is the mantissa with digits $0 \leq d_i < \beta$, $0 \leq i \leq p-1$, p is the precision and e is the exponent, $e_{min} \leq e \leq e_{max}$. The IEEE754 Standard specifies several formats for the floating-point numbers by providing specific values for p, β, e_{min} and e_{max}. It also defines some rounding modes, towads $+\infty$, $-\infty$, 0 and to the nearest. Let us write $\circ_{+\infty}$, $\circ_{-\infty}$, $\circ_0$ and $\circ_\sim$ the rounding functions, the IEEE754 Standard defines the semantics of the elementary operations by:

$$x \circledast_r y = \circ_r(x * y) \tag{3}$$

where $\circledast_r$ denotes a floating-point operation $+$, $-$, $\times$ or $\div$ computed using the rounding mode r and $*$ denotes an exact operation. Because of the roundoff errors, the results of the computations are not exact. For example, the value $e = 2.7182818\dots$ can be computed using Bernouilli's formula:

$$e = \lim_{n \to +\infty} u_n \quad \text{with} \quad u_n = \left(1 + \frac{1}{n}\right)^n, \; n \geq 0.$$

In double precision, $u_8 = 2.718282$ but then the accuracy decreases as n grows: $u_{14} = 2.716110$, $u_{16} = 3.0.35035$ and $u_{17} = 1.0$. For the floating-point arithmetic,

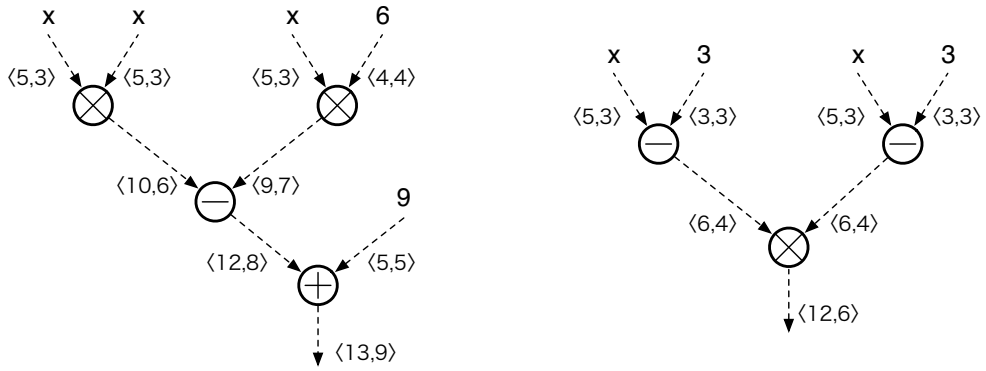

Fig. 1. *Two fixed-point implementations of* $x^2 - 6x + 9$ *where* x *format is* $\langle 5, 3\rangle$.

the synthesis of expressions consists in generating an expression which is mathematically equal to the original one and which minimizes the roundoff error on the result, i.e. the distance $|r - \hat{r}|$ between the exact result r and the floating-point result $\hat{r}$.

There exists no standard for the fixed-point arithmetic comparable to the IEEE754 Standard. A fixed-point format $\langle w, i\rangle$ depends on the total number of bits w used to encode the value and on the location of the fixed-point relative to the most significant bit [5]. In general, the numbers are encoded using two's complement and the sequence of bits $b_{w-1} \dots b_0$ reprensents the value $-b_{w-1} \cdot 2^{i-1} + \sum_{j=2}^{j=w} b_{w-j} \cdot 2^{i-j}$ and the distance between two consecutive numbers is 2^{i-w}. The format $\langle w_r, i_r\rangle$ of the result of an elementary operation depends on the formats $\langle w_1, i_1\rangle$ and $\langle w_2, i_2\rangle$ of its operands:

$$\text{Addition: } \begin{cases} w_r &= i_r + max(w_1 - i_1, w_2 - i_2) \\ i_r &= max(i_1 + \neg s_1 \wedge s_2, i_2 + \neg s_2 \wedge s_1) + 1 \end{cases} \qquad \text{Product: } \begin{cases} w_r &= w_1 + w_2 \\ i_r &= i_1 + i_2 \end{cases} \tag{4}$$

In Equation (4), s_1 and s_2 denote the signs of the operands. Using the formats of Equation (4), the operations are exact and no rounding is needed.

Synthesizing an efficient expression for the fixed-point arithmetic consists of generating an expression equivalent to the original one and which minimizes the size of the implementation, or, in other words, which minimizes the sum of the sizes w of the formats of the intermediary results (the outputs of the operators.) For example, Figure 1 displays two implementations of the polynomial $x^2 - x + 9$, with x in the format $\langle 5, 3\rangle$. The first scheme corresponds to the direct implementation and requires 68 bits to store the intermediary results while the second scheme implements the equivalent formula $(x - 3) \times (x - 3)$ and necessitates 40 bits only.

Our last computer arithmetic is the interval arithmetic [12]. Intervals are commonly used to bound the exact result of computations carried out with floating-point numbers. Given two intervals $[\underline{x}, \overline{x}]$ and $[\underline{y}, \overline{y}]$ whose bounds are floating-point

numbers, the elementary operations are defined by:

$$[\underline{x},\overline{x}] \boxplus [\underline{y},\overline{y}] = [\underline{x} \oplus_{-\infty} \underline{y}, \overline{x} \oplus_{+\infty} \overline{y}] \qquad [\underline{x},\overline{x}] \boxtimes [\underline{y},\overline{y}] = \left[\min \left\{ \begin{array}{l} \underline{x} \otimes_{-\infty} \underline{y} \\ \underline{x} \otimes_{-\infty} \overline{y} \\ \overline{x} \otimes_{-\infty} \underline{y} \\ \overline{x} \otimes_{-\infty} \overline{y} \end{array} \right\}, \max \left\{ \begin{array}{l} \underline{x} \otimes_{+\infty} \underline{y} \\ \underline{x} \otimes_{+\infty} \overline{y} \\ \overline{x} \otimes_{+\infty} \underline{y} \\ \overline{x} \otimes_{+\infty} \overline{y} \end{array} \right\} \right] \quad (5)$$

Interval arithmetic suffers from the decorellation of the variables (the absence of relations) and from the wrapping effect. For example, because of the decorrelations, the value of the function $f(x) = \frac{x}{x-2}$ is $[1.5, 4]$ when $x = [3, 4]$. However, the function f is mathematically equal to $g(x) = 1 + \frac{2}{x-2}$ and $g([3, 4]) = [2, 3]$. While both results are correct, $g([3, 4])$ is clearly more accurate than $f([3, 4])$. Synthesizing efficient expressions for the interval arithmetic then consists of generating expressions whose evaluation yields intervals of small width, in order to optimize the accuracy of the results.

3 Correctness of the Synthesis

The synthesized expressions being possibly very different from the original ones, the correctness of the process is based on semantics and not on syntax. We use the framework for program transformation introduced by P. and R. Cousot [3]. In Figure 2, we introduce four non-standard small-step operational semantics for the evaluation of expressions, where $*$ stands for any elementary operation. These semantics are denoted $\rightarrow_{\texttt{int}}$, $\rightarrow_{\texttt{float}}$, $\rightarrow_{\texttt{fixed}}$ and $\rightarrow_{[]}$ and they are related to the integer, floating-point, fixed-point and interval arithmetics, respectively.

For the integer arithmetic $\rightarrow_{\texttt{int}}$, a non standard value is a pair $(\hat{v}, v) \in \texttt{int} \times \mathbb{Z}$ where `int` denotes the set of computer signed integers (for example the 32 bits or 64 bit integers) and $\mathbb{Z}$ denotes the set of signed mathemtical integers. Intuitively, a value $(\hat{v}, v)$ gives both the exact value v and its computer approximation $\hat{v}$. A state $\langle e, m, \rho \rangle \in \text{Expr} \times \mathbb{N} \times \text{Env}_{\texttt{int}}$ of the integer non-standard semantics is made of an expression e, of an environment $\rho : \text{Var} \rightarrow (\texttt{int} \times \mathbb{Z})$ of $Env_{\texttt{int}}$ mapping variables to non-standard values and of a non-negative integer $m \in \mathbb{N}$ indicating the maximal value encountered during the evaluation of the expression, in absolute value. The integer m has to be minimized during the synthesis of a new expression in order to keep the intermediary results inside the range $[\mathbf{m}, \mathbf{M}]$ introduced in Section 2.

For the floating-point arithmetic $\rightarrow_{\texttt{float}}$, a non-standard value is a pair $(\hat{v}, v) \in \texttt{float} \times \mathbb{R}$ where `float` is the set of floating-point numbers (one of the IEEE754 formats) and $\mathbb{R}$ the set of real numbers. In Figure 2, we assume that the floating-point operations are carried out using the rounding mode to the nearest. A state $\langle e, \rho \rangle$ of the non-standard floating-point semantics is made of an expression e and of an environment $\rho \in \text{Var} \rightarrow (\texttt{float} \times \mathbb{R})$ mapping variables to non-standard values. Intuitively, the synthesis of an efficient expression for the floating-point arithmetic has to minimize the quantity $|\hat{v} - v|$, i.e. the difference between the computer and exact results.

For the fixed-point arithmetic $\rightarrow_{\texttt{fixed}}$, a non standard value is a pair $(v^{\langle w,i \rangle}, v) \in \texttt{fixed} \times \mathbb{R}$ where `fixed` denotes the set of fixed-point numbers. We consider that in

$$\frac{\rho(x) = (\hat{v}, v) \quad m' = \max\left(|v|, m\right)}{\langle x, m, \rho\rangle \rightarrow_{\texttt{int}} \langle(\hat{v}, v), m', \rho\rangle}$$

$$\frac{\hat{v} = \hat{v_1} \circledast \hat{v_2} \quad v = v_1 * v_2 \quad m' = \max(|v_1|, |v_2|, |v|, m)}{\langle(\hat{v_1}, v_1) * (\hat{v_2}, v_2), m, \rho\rangle \rightarrow_{\texttt{int}} \langle(\hat{v}, v), m', \rho\rangle}$$

$$\frac{\langle e_1, m, \rho\rangle \rightarrow_{\texttt{int}} \langle e_1', m', \rho\rangle \quad m'' = \max(m, m')}{\langle e_1 * e_2, m, \rho\rangle \rightarrow_{\texttt{int}} \langle e_1' * e_2, m'', \rho\rangle}$$

$$\frac{\langle e_2, m, \rho\rangle \rightarrow_{\texttt{int}} \langle e_2', m', \rho\rangle \quad m'' = \max(m, m')}{\langle(\hat{v_1}, v_1) * e_2, m, \rho\rangle \rightarrow_{\texttt{int}} \langle(\hat{v_1}, v_1) * e_2', m'', \rho\rangle}$$

$$\frac{\rho(x) = (\hat{v}, v)}{\langle x, \rho\rangle \rightarrow_{\texttt{float}} \langle(\hat{v}, v), \rho\rangle} \qquad \frac{\hat{v} = \hat{v_1} \circledast_{\sim} \hat{v_2} \quad v = v_1 * v_2}{\langle(\hat{v_1}, v_1) * (\hat{v_2}, v_2), \rho\rangle \rightarrow_{\texttt{float}} \langle(\hat{v}, v), \rho\rangle}$$

$$\frac{\langle e_1, \rho\rangle \rightarrow_{\texttt{float}} \langle e_1', \rho\rangle}{\langle e_1 * e_2, \rho\rangle \rightarrow_{\texttt{float}} \langle e_1' * e_2, \rho\rangle} \qquad \frac{\langle e_2, \rho\rangle \rightarrow_{\texttt{float}} \langle e_2', \rho\rangle}{\langle(\hat{v_1}, v_1) * e_2, \rho\rangle \rightarrow_{\texttt{float}} \langle(\hat{v_1}, v_1) * e_2', \rho\rangle}$$

$$\frac{\rho(x) = (v^{\langle w,i\rangle}, v)}{\langle x, W, \rho\rangle \rightarrow_{\texttt{fixed}} \langle(v^{\langle w,i\rangle}, v), W, \rho\rangle}$$

$$\frac{v^{\langle w,i\rangle} = v_1^{\langle w_1,i_1\rangle} \circledast v_2^{\langle w_2,i_2\rangle} \quad v = v_1 * v_2 \quad W' = W + w_1 + w_2 + w}{\langle(v_1^{\langle w_1,i_1\rangle}\rangle, v_1) * (v_2^{\langle w_2,i_2\rangle}, v_2), W, \rho\rangle \rightarrow_{\texttt{fixed}} \langle(v^{\langle w,i\rangle}, v), W', \rho\rangle}$$

$$\frac{\langle e_1, W, \rho\rangle \rightarrow_{\texttt{fixed}} \langle e_1', W', \rho\rangle}{\langle e_1 * e_2, W, \rho\rangle \rightarrow_{\texttt{fixed}} \langle e_1' * e_2, W', \rho\rangle}$$

$$\frac{\langle e_2, W, \rho\rangle \rightarrow_{\texttt{fixed}} \langle e_2', W', \rho\rangle}{\langle(v_1^{\langle w_1,i_1\rangle}, v_1) * e_2, W, \rho\rangle \rightarrow_{\texttt{fixed}} \langle(v_1^{\langle w_1,i_1\rangle}, v_1) * e_2', W', \rho\rangle}$$

$$\frac{\rho(x) = (\hat{v}, v)}{\langle x, \rho\rangle \rightarrow_{[]} \langle(\hat{v}, v), \rho\rangle} \qquad \frac{\hat{v} = v_1 \boxtimes \hat{v_2} \quad v = \{x * y \ : \ x \in v_1, \ y \in v_2\}}{\langle(\hat{v_1}, v_1) * (\hat{v_2}, v_2), \rho\rangle \rightarrow_{[]} \langle(\hat{v}, v), \rho\rangle}$$

$$\frac{\langle e_1, \rho\rangle \rightarrow_{[]} \langle e_1', \rho\rangle}{\langle e_1 * e_2, \rho\rangle \rightarrow_{[]} \langle e_1' * e_2, \rho\rangle} \qquad \frac{\langle e_2, \rho\rangle \rightarrow_{[]} \langle e_2', \rho\rangle}{\langle(\hat{v_1}, v_1) * e_2, \rho\rangle \rightarrow_{[]} \langle(\hat{v_1}, v_1) * e_2', \rho\rangle}$$

Fig. 2. *Non standard semantics for the integer, floating-point, fixed-point and interval arithmetics.*

`fixed`, each fixed-point number has its own format $\langle w, i\rangle$, as introduced in Section 2. A state $\langle e, W, \rho\rangle \in \text{Expr} \times \mathbb{N} \times \text{Env}_{\texttt{fixed}}$ is made of an expression e, a non-negative integer W and an environment $\rho \in \text{Var} \to (\texttt{fixed} \times \mathbb{R})$ mapping variables to non-standard values. Intuitively, W records the total number of bits required to represent all the intermediary results during the evaluation of the expression. The synthesis of a new expression for the fixed-point arithmetic has to minimize W.

Finally, a non standard value $(\hat{v}, v) \in (\texttt{float} \times \texttt{float}) \times \wp(\mathbb{R})$ for the interval arithmetic is made of an inteval $\hat{v}$ with floating-point bounds and a subset v of $\mathbb{R}$ ($\wp(\mathbb{R})$ denotes the powerset of $\mathbb{R}$.) Intuitively, v is used to compute the exact image of the points belonging to the input intervals. In other words, if $e(x, y)$ is an expression depending on two variables $x \in [a, b]$ and $y \in [c, d]$, we aim at computing in the non-standard semantics the exact image $I = \{e(x, y) \ : a \leq x \leq b, \ c \leq y \leq d\}$

of e. For the interval arithmetic, the synthesis of new expressions has to minimize the width of the interval corresponding to the result of the computation.

Given a set Θ of environments, the collecting semantics $[\![e]\!]_{\texttt{int}}\ \Theta$, $[\![e]\!]_{\texttt{float}}\ \Theta$, $[\![e]\!]_{\texttt{fixed}}\ \Theta$ and $[\![e]\!]_{[]}\ \Theta$ correspond to the sets of maximal traces starting by the states $\{\langle e, 0, \rho\rangle \ :\ \rho \in \Theta\}$ for the integer arithmetic, $\{\langle e, \rho\rangle \ :\ \rho \in \Theta\}$ for the floating-point arithmetic, $\{\langle e, 0, \rho\rangle \ :\ \rho \in \Theta\}$ for the fixed-point arithmetic and $\{\langle e, \rho\rangle \ :\ \rho \in \Theta\}$ for the interval arithmetic.

In order to define the correctness of the synthesis, we also introduce observational abstractions $\alpha_{\mathcal{O}}$ of the states [3]. It is correct to replace an expression e by another expression e' if for any Θ, $[\![e]\!]\ \Theta = S$, $[\![e']\!]\ \Theta = S'$ and $\{\alpha_{\mathcal{O}}(s) \ :\ s \in S\} = \{\alpha_{\mathcal{O}}(s') \ :\ s' \in S'\}$ where $[\![e]\!]\ \Theta$ is one of our four collecting semantics. For the synthesis of expressions, the observational abstraction discards the computer results of the states and conserve only the mathematical results. Then, the synthesis is correct if the new expression always returns the same mathematical result than the source expression. In all our semantics, values are pairs $(\hat{v}, v)$ where $\hat{v}$ is a value representable in machine and v is a mathematical value. We define $\alpha_{\mathcal{O}}$ as the second projection, i.e. $\alpha_{\mathcal{O}}\big((\hat{v}, v)\big) = v$ for any non-standard semantics. The abstraction $\alpha_{\mathcal{O}}$ is then extended to states by projecting the values inside the expressions and environments.

4 Abstraction of Equivalent Expressions

In general, the number of expressions equivalent to an original expression e by associativity, commutativity, distributivity and factorization is exponential in the size of e. For example, the number of ways to evaluate the polynomial $\underbrace{(x-1)\times\ldots\times(x-1)}_{n \text{ times}}$ is $2.3\cdot 10^6$ for $n=5$ and $1.3\cdot 10^9$ for $n=6$ [13]. In this section, we introduce two abstractions, polynomial in size, of the set of mathematically equivalent expressions.

The first abstraction consists of identifying the expressions whose syntactic trees are equal up to depth k. This abstraction is a simplified version of the abstraction introduced in [11]. In the present article, we introduce an under-approximation of the set of equivalent expressions while the more complicated abstraction introduced in [11] was a complete covering of the mathematically equivalent expressions. Because we limit ourselves to the expressions whose syntactic trees are equal up to depth k, the application of algebraic laws like associativity, commutativity, etc. yields a limited number of expressions related to the user defined parameter k. We start by introducing a special expression $\top \in \text{Expr}$ and the function $\ulcorner.\urcorner^k \ :\ \text{Expr} \to \text{Expr}$ which discards the deepest level of the syntactic tree of an expression [11]:

$$\begin{array}{llll} \ulcorner v\urcorner^k = v & \text{if} & k \geq 0 & \qquad \ulcorner e_1 * e_2\urcorner^0 = \top & \text{if} & k = 0 \\ \ulcorner x\urcorner^k = x & \text{if} & k \geq 0 & \qquad \ulcorner e_1 * e_2\urcorner^k = \ulcorner e_1\urcorner^{k-1} * \ulcorner e_2\urcorner^{k-1} & \text{if} & k \geq 1 \end{array} \tag{6}$$

Let $\mathcal{R} \subseteq \text{Expr} \times \text{Expr}$ be a binary relation on the set of expressions. We use $\mathcal{R}$ to identify mathematically equivalent expressions. For example, $\mathcal{R}$ may contain

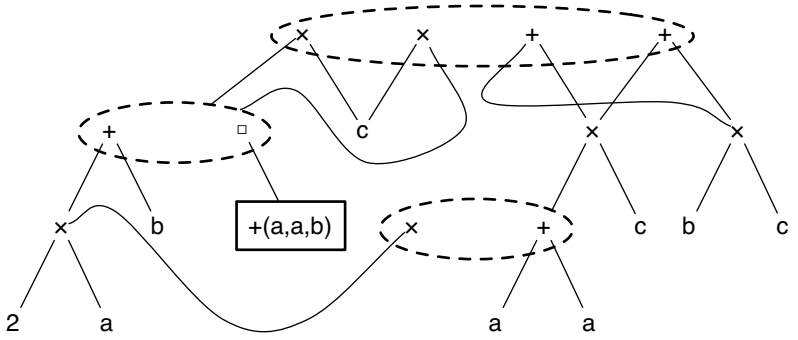

Fig. 3. *APEG for the expression* $e = \big((a + a) + c\big) \times c$.

associativity or distributivity:

$$\Big\{\big(e_1 + (e_2 + e_3), (e_1 + e_2) + e_3\big) \ : \ e_1,\ e_2,\ e_3 \in \mathrm{Expr}\Big\} \subseteq \ \mathcal{R} \tag{7}$$

$$\Big\{\big(e_1 \times (e_2 + e_3), e_1 \times e_2 + e_1 \times e_3\big) \ : \ e_1,\ e_2,\ e_3 \in \mathrm{Expr}\Big\} \subseteq \ \mathcal{R} \tag{8}$$

Note that we do not require $\mathcal{R}$ to be transitive. To generate a subset of the expressions equivalent to a source expression e, we use the transition $\Rightarrow_k$ of Equation (9) which relates states $\langle E, K\rangle \in \wp(Expr) \times \wp(Expr)$:

$$\frac{e \in E \quad e \ \mathcal{R} \ e' \quad \ulcorner e' \urcorner^k \notin K}{\langle E, K\rangle \rightarrow_k \langle \{e'\} \cup E, \{\ulcorner e' \urcorner^k\} \cup K\rangle}. \tag{9}$$

Using Equation (9) and the initial state $\langle\{e\}, \{\ulcorner e \urcorner^k\}\rangle$, we may generate a maximal set E of expressions all equivalent to e and such that for any pair $e_1, e_2 \in E$, $\ulcorner e_1 \urcorner^k \neq \ulcorner e_2 \urcorner^k$. The set E is an under-approximation of the set of expressions mathematically equivalent to e.

The second abstraction is based on the notion of Abstract Program Equivalence Graph (APEG for short) [6]. The APEGs are an extension of the Equivalence Program Expression Graphs (EPEGs) introduced by R. Tate *et al.* [15,16]. An APEG is defined inductively as follows:

(i) A value v or a variable x is an APEG,

(ii) An expression $p_1 * p_2$ is an APEG, where p_1 and p_2 are APEGs and $*$ is a binary operator,

(iii) 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,

(iv) A non-empty set $\{p_1, \ldots, p_n\}$ of APEGs is an APEG where p_i, $1 \leq i \leq n$, is not a set of APEGs itself. The set $\{p_1, \ldots, p_n\}$ is called equivalence class.

An example of APEG is given in Figure 3. When an equivalence class (denoted by a dotted ellipse in Figure 3) 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 3, the leafs corresponding to the variables a, b and c are duplicated while, it practice, they are defined only once in the structure.

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

(i) If p is a value v or a variable x then $\mathcal{A}(p) = \{v\}$ or $\mathcal{A}(p) = \{x\}$,

(ii) If p is an expression $p_1 * p_2$ then $\mathcal{A}(p) = \bigcup_{e_1 \in \mathcal{A}(p_1),\ e_2 \in \mathcal{A}(p_2)} e_1 * e_2$,

(iii) If p is a box $\boxed{*(p_1, \ldots, p_n)}$ then $\mathcal{A}(p)$ contains all the parsings of $e_1 * \ldots * e_n$ for all $e_1 \in \mathcal{A}(p_1), \ldots, e_n \in \mathcal{A}(p_n)$,

(iv) If p is an equivalence class $\{p_1, \ldots, p_n\}$ then $\mathcal{A}(p) = \bigcup_{1 \leq i \leq n} \mathcal{A}(p_i)$.

For instance, the APEG p of Figure 3 represents all the following expressions:

$$\mathcal{A}(p) = \left\{ \begin{array}{c} \big((a+a)+b\big) \times c,\ \big((a+b)+a\big) \times c,\ \big((b+a)+a\big) \times c, \\ \big((2 \times a)+b\big) \times c,\ c \times \big((a+a)+b\big),\ c \times \big((a+b)+a\big), \\ c \times \big((b+a)+a\big),\ c \times \big((2 \times a)+b\big),\ (a+a) \times c + b \times c, \\ (2 \times a) \times c + b \times c,\ b \times c + (a+a) \times c,\ b \times c + (2 \times a) \times c \end{array} \right\} \tag{10}$$

In comparison, with the first abstraction introduced at the beginning of this section, one may under-approximate the set of expressions equivalent to $e = c \times \big((a+a)+b\big)$ by the set

$$S_1 = \Big\{ c \times \big((a+a)+b\big),\ c \times (a+a) + c \times b \Big\} \quad \text{if } k = 1, \tag{11}$$

and by the set

$$S_2 = \left\{ \begin{array}{c} \big((a+a)+b\big) \times c,\ \big(a+(a+b)\big) \times c, \\ (a+a) \times c + b \times c,\ a \times c + (a+b) \times c \end{array} \right\} \quad \text{if } k = 2. \tag{12}$$

In their article on EPEGs, R. Tate *et al.* use rewritting rules to extend the structure up to saturation [15,16]. 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 4. An alternative technique for APEG construction is to use dedicated algorithms. Such algorithms, working in polynomial time, have been proposed in [6].

The abstractions defined previously in this section do not introduce expressions that are not mathematically equivalent to the source expression. Then, for synthesis, it is correct to select any expression belonging to the abstraction of a set of equivalent expressions. The selection criteria used at synthesis time are discussed in Section 5. We end this section by formalizing the correctness of the abstractions.

Let $\mathcal{R} \subseteq \text{Expr} \times \text{Expr}$ be the binary relation on the set of expressions introduced ealier in this section to identify mathematically equivalent expressions (see equations

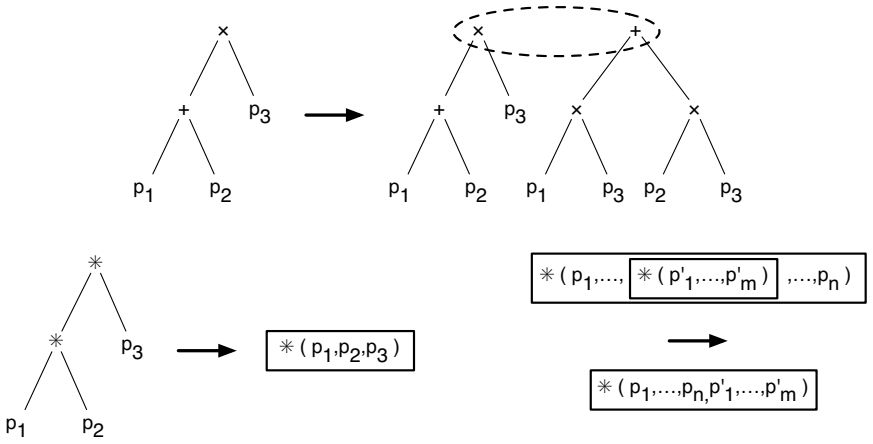

Fig. 4. *Some rules for APEG construction by pattern matching.*

(7) and (8).) The set of expressions equivalent to an original expression e can be generated by the following rule $\Rightarrow \in \wp(\text{Expr}) \times \wp(\text{Expr})$:

$$\frac{e \in E \quad e \mathcal{R} e'}{E \to \{e'\} \cup E} \tag{13}$$

The set $\mathcal{E}(e)$ of expressions equivalent to e using the relations contained in $\mathcal{R}$ is such that the sequence $\{e\} \to^* \mathcal{E}(e)$ of transitions is maximal (i.e. $\mathcal{E}(e) \to E'$ implies $E' = \mathcal{E}(e)$.)

Let $\langle E^\sharp, K\rangle$ be the state resulting from a maximal transition path based on Equation (9): $\langle\{e\}, \{\ulcorner e^{\neg k}\}\rangle \to_k \langle E^\sharp, K\rangle$ and let $\mathcal{A}(p)$ be the set of expressions contained inside an APEG p built from e, for example using the rules of Figure 4. Then $E^\sharp$ and $\mathcal{A}(p)$ are under-approximations of $\mathcal{E}(e)$ and there exists the following Galois connexions between the set of equivalent expression and its abstractions:

$$\langle \wp(Expr), \subseteq\rangle \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} \langle \wp(Expr) \times \wp(Expr), \subseteq_\times\rangle \tag{14}$$

$$\langle \wp(Expr), \subseteq\rangle \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} \langle \Pi, \subseteq_\Pi\rangle \tag{15}$$

In equations (14) and (15), $\subseteq_\times$ denotes the component-wise inclusion, Π denotes the set of APEGs and $\subseteq_\Pi$ is the partial order on APEGs. Intuitively, $p_1 \subseteq_\Pi p_2$ if $\mathcal{A}(p_1) \subseteq \mathcal{A}(p_2)$. An inductive definition of $\subseteq_\Pi$ is given in [6]. The concretizations of abstract states $\langle E^\sharp, K\rangle$ or p are defined by the following functions:

$$\gamma_1(\langle E, K\rangle) = \bigcup_{e\in E,\ \{e\}\to E'} E' \qquad \text{and} \qquad \gamma_2(p) = \bigcup_{e\in\mathcal{A}(p),\ \{e\}\to E'} E'. \tag{16}$$

Hence, the abstract sets $\langle E^\sharp, K\rangle$ and p do not contain expressions which are not mathematically equivalent to the others and any expression e' in $E^\sharp$ or $\mathcal{A}(p)$ may be selected in order to synthesize a new expression as it will not be distinguishible from e by the observational abstraction $\alpha_{\mathcal{O}}$ introduced in Section 3.

$$\frac{\rho(x) = [\underline{v}, \overline{v}] \quad m' = \max\left(|\underline{v}|, |\overline{v}|, m\right)}{\langle x, m, \rho\rangle \rightarrow^\sharp_{\texttt{int}} \langle [\underline{v}, \overline{v}], m', \rho\rangle}$$

$$\frac{[\underline{v}, \overline{v}] = [\underline{v_1}, \overline{v_1}] \circledast_{\texttt{int}} [\underline{v_2}, \overline{v_2}] \quad m' = \max(|\underline{v_1}|, |\overline{v_1}|, |\underline{v_2}|, |\overline{v_2}|, |\underline{v}|, |\overline{v}|, m)}{\langle [\underline{v_1}, \overline{v_1}] * [\underline{v_2}, \overline{v_2}], m, \rho\rangle \rightarrow^\sharp_{\texttt{int}} \langle [\underline{v}, \overline{v}], m', \rho\rangle}$$

$$\frac{\langle e_1, m, \rho\rangle \rightarrow^\sharp_{\texttt{int}} \langle e_1', m', \rho\rangle \quad m'' = \max(m, m')}{\langle e_1 * e_2, m, \rho\rangle \rightarrow^\sharp_{\texttt{int}} \langle e_1' * e_2, m'', \rho\rangle}$$

$$\frac{\langle e_2, m, \rho\rangle \rightarrow^\sharp_{\texttt{int}} \langle e_2', m', \rho\rangle \quad m'' = \max(m, m')}{\langle [\underline{v_1}, \overline{v_1}] * e_2, m, \rho\rangle \rightarrow^\sharp_{\texttt{int}} \langle [\underline{v_1}, \overline{v_1}] * e_2', m'', \rho\rangle}$$

$$\frac{\rho(x) = ([\underline{\hat{v}}, \overline{\hat{v}}], [\underline{v}, \overline{v}])}{\langle x, \rho\rangle \rightarrow^\sharp_{\texttt{float}} \langle ([\underline{\hat{v}}, \overline{\hat{v}}], [\underline{v}, \overline{v}]), \rho\rangle} \quad \frac{\langle e_1, \rho\rangle \rightarrow^\sharp_{\texttt{float}} \langle e_1', \rho\rangle}{\langle e_1 * e_2, \rho\rangle \rightarrow^\sharp_{\texttt{float}} \langle e_1' * e_2, \rho\rangle}$$

$$\frac{[\underline{\hat{v}}, \overline{\hat{v}}] = [\underline{\hat{v_1}}, \overline{\hat{v_1}}] \circledast_{\sim} [\underline{\hat{v_2}}, \overline{\hat{v_2}}] \quad [\underline{v}, \overline{v}] = [\underline{v_1}, \overline{v_1}] \circledast_{\downarrow} [\underline{v_2}, \overline{v_2}]}{\langle ([\underline{\hat{v_1}}, \overline{\hat{v_1}}], [\underline{v_1}, \overline{v_1}]) * ([\underline{\hat{v_2}}, \overline{\hat{v_2}}], [\underline{v_2}, \overline{v_2}]), \rho\rangle \rightarrow^\sharp_{\texttt{float}} \langle ([\underline{\hat{v}}, \overline{\hat{v}}], [\underline{v}, \overline{v}]), \rho\rangle}$$

$$\frac{\langle e_2, \rho\rangle \rightarrow^\sharp_{\texttt{float}} \langle e_2', \rho\rangle}{\langle ([\underline{\hat{v_1}}, \overline{\hat{v_1}}], [\underline{v_1}, \overline{v_1}]) * e_2, \rho\rangle \rightarrow^\sharp_{\texttt{float}} \langle ([\underline{\hat{v_1}}, \overline{\hat{v_1}}], [\underline{v_1}, \overline{v_1}]) * e_2', \rho\rangle}$$

$$\frac{\rho(x) = [\underline{v}, \overline{v}]^{\langle w,i\rangle}}{\langle x, W, \rho\rangle \rightarrow^\sharp_{\texttt{fixed}} \langle [\underline{v}, \overline{v}]^{\langle w,i\rangle}, W, \rho\rangle} \quad \frac{\langle e_1, W, \rho\rangle \rightarrow^\sharp_{\texttt{fixed}} \langle e_1', W', \rho\rangle}{\langle e_1 * e_2, W, \rho\rangle \rightarrow^\sharp_{\texttt{fixed}} \langle e_1' * e_2, W', \rho\rangle}$$

$$\frac{[\underline{v}, \overline{v}]^{\langle w,i\rangle} = [\underline{v_1}, \overline{v_1}]^{\langle w_1,i_1\rangle} \circledast_{\texttt{fixed}} [\underline{v_2}, \overline{v_2}]^{\langle w_2,i_2\rangle} \quad W' = W + w_1 + w_2 + w}{\langle [\underline{v_1}, \overline{v_1}]^{\langle w_1,i_1\rangle} * [\underline{v_2}, \overline{v_2}]^{\langle w_2,i_2\rangle}, W, \rho\rangle \rightarrow^\sharp_{\texttt{fixed}} \langle [\underline{v}, \overline{v}]^{\langle w,i\rangle}, W', \rho\rangle}$$

$$\frac{\langle e_2, W, \rho\rangle \rightarrow^\sharp_{\texttt{fixed}} \langle e_2', W', \rho\rangle}{\langle [\underline{v_1}, \overline{v_1}]^{\langle w_1,i_1\rangle} * e_2, W, \rho\rangle \rightarrow^\sharp_{\texttt{fixed}} \langle [\underline{v_1}, \overline{v_1}]^{\langle w_1,i_1\rangle} * e_2', W', \rho\rangle}$$

$$\frac{\rho(x) = [\underline{v}, \overline{v}]}{\langle x, \rho\rangle \rightarrow^\sharp_{[]} \langle [\underline{v}, \overline{v}], \rho\rangle} \quad \frac{[\underline{v}, \overline{v}] = [\underline{v_1}, \overline{v_1}] \circledast_{\leftrightarrow} [\underline{v_2}, \overline{v_2}]}{\langle [\underline{v_1}, \overline{v_1}] * [\underline{v_2}, \overline{v_2}], \rho\rangle \rightarrow^\sharp_{[]} \langle (\hat{v}, v), \rho\rangle}$$

$$\frac{\langle e_1, \rho\rangle \rightarrow^\sharp_{[]} \langle e_1', \rho\rangle}{\langle e_1 * e_2, \rho\rangle \rightarrow^\sharp_{[]} \langle e_1' * e_2, \rho\rangle} \quad \frac{\langle e_2, \rho\rangle \rightarrow^\sharp_{[]} \langle e_2', \rho\rangle}{\langle [\underline{v_1}, \overline{v_1}] * e_2, \rho\rangle \rightarrow^\sharp_{[]} \langle [\underline{v_1}, \overline{v_1}] * e_2', \rho\rangle}$$

Fig. 5. *Abstract semantics for the integer, floating-point, fixed-point and interval arithmetics.*

5 Generation of New Expressions

This section concerns the last step of the synthesis which consists of selecting an expression inside the abstract representations of equivalent expressions. First of all, we introduce abstract semantics, in Figure 5, in order to compare the quality of mathematically equivalent expressions. These semantics abstract the non-standard semantics of Figure 2 in which the mathematical values have been discarded. The abstract state contain intervals instead of scalar values since we aim at synthesizing expressions optimized for large ranges of inputs.

A value of the abstract integer semantics $\rightarrow^\sharp_{\texttt{int}}$ is an interval $[\underline{v}, \overline{v}] \in \texttt{int} \times \texttt{int}$ and an abstract state is a triple $\langle e, m, \rho\rangle \in \text{Expr} \times \texttt{int} \times \text{Env}^\sharp_{\texttt{int}}$ where $\text{Env}^\sharp_{\texttt{int}}$ is the set of environments mapping variables to abstract integer values. The operator $\circledast_{\texttt{int}}$ denotes the operation $*$ between intervals of integers, no rounding is require in this case. An expression e_1 is better than an expression e_2 for an abstract environment $\rho^\sharp$, denoted $e_1 \prec^{\rho^\sharp}_{\texttt{int}} e_2$, if $\langle e_1, 0, \rho^\sharp\rangle \rightarrow^{\sharp *}_{\texttt{int}} \langle [\underline{v_1}, \overline{v_1}], m_1, \rho^\sharp_1\rangle$, $\langle e_2, 0, \rho^\sharp\rangle \rightarrow^{\sharp *}_{\texttt{int}} \langle [\underline{v_2}, \overline{v_2}], m_2, \rho^\sharp_2\rangle$ and $m_1 \leq m_2$. Recall from Section 3 that m gives the maximal value, in absolute value of the intermediary results encountered during the evaluation of the expression.

A value of the abstract floating-point semantics $\rightarrow^\sharp_{\texttt{float}}$ is a pair of intervals $([\hat{\underline{v}}, \hat{\overline{v}}], [\underline{v}, \overline{v}]) \in (\texttt{float} \times \texttt{float}) \times (\texttt{float} \times \texttt{float})$. Intuitively, the first interval is the abstraction of the set of concrete values and the second interval is an under-approximation of the exact results of the computation. Hence, in the abstract semantics of Figure 5, the operations between the first intervals are carried out using the standard rounding mode $\sim$ of the machine (to the nearest in general) while for the second intervals, we use the rounding mode towards inside, denoted $\downarrow$. For instance

$$[\underline{v_1}, \overline{v_1}] \boxplus_\sim [\underline{v_2}, \overline{v_2}] = [\underline{v_1} \oplus_\sim \underline{v_2}, \overline{v_1} \oplus_\sim \overline{v_2}] \tag{17}$$

and

$$[\underline{v_1}, \overline{v_1}] \boxplus_\downarrow [\underline{v_2}, \overline{v_2}] = \left[\min(\underline{v_1} \oplus_{-\infty} \underline{v_2}, \overline{v_1} \oplus_{-\infty} \overline{v_2}), \max(\underline{v_1} \oplus_{+\infty} \underline{v_2}, \overline{v_1} \oplus_{+\infty} \overline{v_2})\right]. \tag{18}$$

An abstract state is a pair $\langle e, \rho\rangle \in \text{Expr} \times \text{Env}^\sharp_{\texttt{float}}$ where $\text{Env}^\sharp_{\texttt{float}}$ denotes the environments mapping variables to abstract floating-point values. An expression e_1 is better than an expression e_2 for an abstract environment $\rho^\sharp$, denoted $e_1 \prec^{\rho^\sharp}_{\texttt{float}} e_2$, if $\langle e_1, \rho^\sharp\rangle \rightarrow^{\sharp *}_{\texttt{float}} \langle([\hat{\underline{v_1}}, \hat{\overline{v_1}}], [\underline{v_1}, \overline{v_1}]), \rho^\sharp_1\rangle$, $\langle e_2, \rho^\sharp\rangle \rightarrow^{\sharp *}_{\texttt{float}} \langle([\hat{\underline{v_2}}, \hat{\overline{v_2}}], [\underline{v_2}, \overline{v_2}]), \rho^\sharp_2\rangle$ and $\max(|\hat{\underline{v_1}} - \underline{v_1}|, |\hat{\overline{v_1}} - \overline{v_1}|) \leq \max(|\hat{\underline{v_2}} - \underline{v_2}|, |\hat{\overline{v_2}} - \overline{v_2}|)$. In other words, $e_1 \prec^{\rho^\sharp}_{\texttt{float}} e_2$ if the error in the work case between the computer and mathematical results is less for e_1 than for e_2.

Concerning the fixed-point semantics $\rightarrow^\sharp_{\texttt{fixed}}$, an abstract value is an interval of fixed-point numbers which all have the same format. Such an interval is denoted $[\underline{v}, \overline{v}]^{\langle w,i\rangle}$. We have:

$$[\underline{v}, \overline{v}]^{\langle w,i\rangle} = \left\{ v^{\langle w,i\rangle} \ : \ \underline{v} \leq v \leq \overline{v} \right\} \tag{19}$$

An abstract state is a triple $\langle e, W, \rho\rangle \in \text{Expr} \times \texttt{int} \times \text{Env}^\sharp_{\texttt{fixed}}$ where $\text{Env}^\sharp_{\texttt{fixed}}$ is the set of environments mapping variables to abstract fixed-point values. The operator $\circledast_{\texttt{fixed}}$ denotes the operation $*$ between intevals whose bounds are fixed-point numbers. The operations are exact and no rounding mode is needed for $\circledast_{\texttt{fixed}}$. An expression e_1 is better than an expression e_2 for an abstract environment $\rho^\sharp$, denoted $e_1 \prec^{\rho^\sharp}_{\texttt{fixed}} e_2$, if $\langle e_1, 0, \rho^\sharp\rangle \rightarrow^{\sharp *}_{\texttt{fixed}} \langle [\underline{v_1}, \overline{v_1}]^{\langle w_1, i_1\rangle}, W_1, \rho^\sharp_1\rangle$, $\langle e_2, 0, \rho^\sharp\rangle \rightarrow^{\sharp *}_{\texttt{fixed}} \langle [\underline{v_2}, \overline{v_2}]^{\langle w_2, i_2\rangle}, W_2, \rho^\sharp_2\rangle$ and $W_1 \leq W_2$. In other words, e_1 is better than e_2 if the number of bits require to store the intermediary results is less for e_1 than for e_2.

A value of the abstract interval semantics $\rightarrow^\sharp_{[]}$ is an interval $[\underline{v}, \overline{v}] \in \texttt{float} \times \texttt{float}$. As in Section 2, the operator $\circledast$ denotes the operation $*$ between intevals of floating-point numbers with the rounding mode towards outside (see Equation (5).) An abstract state is a pair $\langle e, \rho\rangle \in \text{Expr} \times \text{Env}^\sharp_{[]}$ where $\text{Env}^\sharp_{[]}$ denotes the

environments mapping variables to intervals of floating-point values. An expression e_1 is better than an expression e_2 for an abstract environment $\rho^\sharp$, denoted $e_1 \prec_{[]}^{\rho^\sharp} e_2$, if $\langle e_1, r^\sharp\rangle \rightarrow_{[]}^{\sharp *} \langle([\underline{v_1}, \overline{v_1}]), \rho_1^\sharp\rangle$, $\langle e_2, r^\sharp\rangle \rightarrow_{[]}^{\sharp *} \langle([\underline{v_2}, \overline{v_2}]), \rho_2^\sharp\rangle$ and $\overline{v_1} - \underline{v_1} \leq \overline{v_2} - \underline{v_2}$ i.e, $e_1 \prec_{\texttt{float}}^{\rho^\sharp} e_2$ if the width of interval resulting from the evaluation of e_1 is smaller than the width of the interval resulting from the evaluation of e_2.

To synthesize a new expression equivalent to a source expression e using the first abstraction of Section 4, we use the rule of Equation (9) to compute the set $E^\sharp$ such that $\langle\{e\}, \{\ulcorner e \urcorner^k\}\rangle \rightarrow_k^* \langle E^\sharp, K\rangle\}$ and we evaluate all the expressions of $E^\sharp$ with the abstract semantics of Figure 5, for the desired arithmetic. Then we select the expression which yields the smallest result in the sense of $\prec_{\texttt{int}}$, $\prec_{\texttt{float}}$, $\prec_{\texttt{fixed}}$ or $\prec_{[]}$.

Concerning APEGs, the synthesis of a new expression requires special techniques to handle the abstract boxes and to search inside the structure. For boxes, a greedy algorithm has been proposed [6]. It consists of selecting in $\boxed{*(p_1, \ldots, p_n)}$ the best operation $p_i * p_j$, $1 \leq i, j \leq n$, $i \neq j$ in the sense of $\prec_{\texttt{int}}$, $\prec_{\texttt{float}}$, $\prec_{\texttt{fixed}}$ or $\prec_{[]}$ and then repeating the process with the box $\boxed{*\big(p_1, \ldots, p_{i-1}, p_{i+1}, \ldots, p_{j-1}, p_{j+1}, \ldots, p_n, (p_i * p_j)\big)}$. For generic APEGs containing equivalence classes, a limited depth search algorithm with memoization has also been proposed [6]. In its simplest setting, it consists of only considering the best expression of the child equivalence classes when synthesizing an expression for a parent equivalent class.

6 Conclusion

In this article, we have presented a general framework for the synthesis of arithmetic expressions which can be evaluated by computers accurately, without overflow and with limited ressources. We have considered the integer, floating-point, fixed-point and interval arithmetics and two abstractions of the set of mathematically equivalent expressions have been described. A large part of this article has been dedicated to the correctness of the synthesis.

Most of this work has been implemented in a tool, called Sardana [7], which accepts the floating-point and fixed-point arithmetics and which implements APEGs. Many experimentations have been carried out with Sardana and the results are convincing [6].

In the future, we would like to generalize our approach to pieces of code more complicated than simple arithmetic expressions. For example, F. Logozzo and T. Ball have worked on binary relations between integer expressions [10]. More generally, we also aim at modifying control structures like conditionals and loops.

References

[1] ANSI/IEEE. *IEEE Standard for Binary Floating-point Arithmetic*, std 754-2008 edition, 2008.

[2] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximations of fixed points. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252. ACM Press, New York, NY, 1977.

[3] Patrick Cousot and Radhia 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. ACM Press, New York, NY, 2002.

[4] David Delmas, Eric Goubault, Sylvie Putot, Jean Souyris, Karim Tekkal, and Franck Vedrine. Towards an industrial use of fluctuat on safety-critical avionics software. In *Formal Methods for Industrial Critical Systems (FMICS'12)*, pages 53–69, 2009.

[5] Mentor Graphics. *Algorithmic C Datatypes*, software version 2.6 edition, 2011. http://www.mentor.com/esl/catapult/algorithmic.

[6] Arnault Ioualalen and Matthieu Martel. A new abstract domain for the representation of mathematically equivalent expressions. In *Static Analysis Symposium (SAS'12)*, volume 7460 of *Lecture Notes in computer Science*, pages 75–93. Springer Verlag, 2012.

[7] Arnault Ioualalen and Matthieu Martel. Sardana: an automatic tool for numerical accuracy optimization. In *15th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic and Verified Numerical Computations (SCAN'12)*, 2012.

[8] Daniel Kästner, Stephan Wilhelm, Stefana Nenova, Patrick Cousot, Radhia Cousot, Jérôme Feret, Antoine Miné, Laurent Mauborgne, and Xavier Rival. Astrée: Proving the absence of runtime errors. In *Embedded Real Time Software and Systems (ERTSS 2011)*, 2010.

[9] Francesco Logozzo. Practical verification for the working programmer with code contracts and abstract interpretation. In *Proceedings of the 12th Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'11)*, volume 6538 of *Lecture Notes in computer Science*, pages 19–22. Springer Verlag, 2011.

[10] Francesco Logozzo and Tom Ball. Modular and verified automatic program repair. In *Proceedings of the 27th ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'12)*. ACM Press, New York, NY, 2012.

[11] Matthieu Martel. Semantics-based transformation of arithmetic expressions. In *Static Analysis Symposium (SAS'07)*, number 4634 in Lecture Notes in computer Science. Springer Verlag, 2007.

[12] Ramon E. Moore, R. Baker Kearfott, and Michael J. Cloud. *Introduction to Interval Analysis*. SIAM, 2009.

[13] Christophe Mouilleron. *Efficient computation with structured matrices and arithmetic expressions*. PhD thesis, Université de Lyon–ENS de Lyon, November 2011.

[14] Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010.

[15] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: A new approach to optimization. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, (POPL'09)*, pages 264–276. ACM Press, 2009.

[16] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: A new approach to optimization. *Logical Methods in Computer Science*, 7(1), 2011.