Towards a Compiler for Reals

EVA DARULOVA and VIKTOR KUNCAK, Ecole Polytechnique Federale de Lausanne

Numerical software, common in scientific computing or embedded systems, inevitably uses a finite-precision approximation of the real arithmetic in which most algorithms are designed. In many applications, the round-off errors introduced by finite-precision arithmetic are not the only source of inaccuracy, and measurement and other input errors further increase the uncertainty of the computed results. Adequate tools are needed to help users select suitable data types and evaluate the provided accuracy, especially for safety-critical applications.

We present a source-to-source compiler called Rosa that takes as input a real-valued program with error specifications and synthesizes code over an appropriate floating-point or fixed-point data type. The main challenge of such a compiler is a fully automated, sound, and yet accurate-enough numerical error estimation. We introduce a unified technique for bounding roundoff errors from floating-point and fixed-point arithmetic of various precisions. The technique can handle nonlinear arithmetic, determine closed-form symbolic invariants for unbounded loops, and quantify the effects of discontinuities on numerical errors. We evaluate Rosa on a number of benchmarks from scientific computing and embedded systems and, comparing it to the state of the art in automated error estimation, show that it presents an interesting tradeoff between accuracy and performance.

CCS Concepts: • **Software and its engineering → Formal software verification**; *Specification languages*; Source code generation;

Additional Key Words and Phrases: Roundoff error, floating-point arithmetic, fixed-point arithmetic, verification, compilation, sensitivity analysis, discontinuity error, loops

ACM Reference Format:
Eva Darulova and Viktor Kuncak. 2017. Towards a compiler for reals. ACM Trans. Program. Lang. Syst. 39, 2, Article 8 (March 2017), 28 pages.
DOI: http://dx.doi.org/10.1145/3014426

1. INTRODUCTION

Much of today's software is numerical in nature. While domains such as scientific computing and embedded systems may appear to differ considerably, they have in common that many of their algorithms are designed in real arithmetic but ultimately have to be implemented in finite precision. This inevitable approximation introduces roundoff errors, which individually may be small but can quickly accumulate or get magnified

This work is supported in part by the European Research Council (ERC) project "Implicit Programming".
A preliminary version of one part of this work appeared in the conference paper "Sound Compilation of Reals," presented at the 2014 ACM SIGPLAN-SIGACT International Conference on Principles of Programming Languages (POPL). The current submission is a complete rewrite (except for algorithm 6) of the preliminary version, presents new and improved techniques, as well as new experimental evaluation.
Authors' addresses: E. Darulova, Campus E1.5, 66125 Saarbrucken, Germany; email: eva@mpi-sws.org; V. Kuncak, EPFL IC LARA INR318, Station 14, CH-1015 Lausanne, Switzerland; email: viktor.kuncak@epfl.ch.

DOI: http://dx.doi.org/10.1145/3014426

to become unacceptable. Additional sources of imprecision, such as measurement and truncation errors, often increase the uncertainty on the computed results further. Many of the developers of numerical software are not experts in numerical analysis, and we thus need adequate tools to help them understand whether the computed values meet the accuracy requirements and remain meaningful in the presence of errors. This is particularly important for safety-critical systems.

Today, however, accuracy in numerical computations is implicit. Many developers write floating-point code as if it was real valued, making it easy to forget to verify the result's accuracy. This also introduces a mismatch between the real-valued algorithms one writes on paper and the low-level implementation details of finite-precision arithmetic. Furthermore, a finite-precision source code semantics prevents the compiler from applying many optimizations (soundly), as, for instance, associativity no longer holds. Finally, with a lack of tool support, programmers will often choose a "safe" default data type, often double floating-point precision, whether or not it is appropriate.

We propose a different strategy. The developer writes the program in a real-valued specification language and makes numerical error bounds explicit in pre- and postconditions. Our source-to-source compiler then determines an appropriate data type for this code that fulfills the specification but is as efficient as possible, and generates the corresponding target program. Our tool supports floating-point as well as fixed-point arithmetic, the latter of which is challenging to generate manually as the programmer has to determine fixed-point formats statically.

Clearly, one of the key challenges of such a compiler is to determine how close a finite-precision representation is to its ideal implementation in real numbers. While sound static analyses exist that compute accurate error bounds for linear operations [Goubault and Putot 2011; Darulova and Kuncak 2011], the presence of nonlinear arithmetic remains a challenge. Roundoff errors and error propagation depend on the ranges of variables in complex and non-obvious ways; even determining these ranges precisely for nonlinear code is hard. Furthermore, due to numerical errors, the control flow in the finite-precision implementation may diverge from the ideal real-valued one, taking a different branch and producing a result that is far off of the expected one. Quantifying discontinuity errors is difficult due to nonlinearity and many correlations between the branches but also due to a lack of smoothness or continuity of the underlying functions that arise in practice [Chaudhuri et al. 2011]. Finally, in general, roundoff errors grow in loops with every iteration, making it impossible to find a non-trivial fixpoint. Even if an iteration bound is known, loop unrolling approaches scale poorly when applied to nonlinear code.

We have addressed these challenges and present here our results towards the goal of a verifying compiler for real arithmetic. In particular, we present

—a real-valued implementation and specification language for numerical programs with uncertainties; we define its semantics in terms of verification constraints that they induce (Section 3).
—an approximation procedure for computing precise range bounds for nonlinear expressions that combines Satisfiability Modulo Theories (SMT) solving with interval arithmetic (Section 4).
—an approach for *sound* and fully *automatic* error estimation for nonlinear expressions for floating-point as well as fixed-point arithmetic of various precisions. We handle roundoff and propagation errors separately with affine arithmetic [de Figueiredo and Stolfi 2004] and a first-order Taylor approximation, respectively. While providing accuracy, this separation also allows us to provide the programmer with useful information about the numerical stability of the computation (Section 5).

—an extension of the error estimation to programs with simple loops, where we developed a technique to express the total error as a function of the number of loop iterations (Section 6).
—a sound and scalable technique to estimate discontinuity errors that crucially relies on the use of a nonlinear SMT solver (Section 7).
—an approach that chooses, based on the error analysis, a data type from a finite set of floating-point or fixed-point precisions that satisfies a given error specification (Section 2).
—an open-source implementation in a tool called Rosa that we evaluate on a number of benchmarks from the scientific computing and embedded systems domain and compare to state-of-the-art tools.

2. A COMPILER FOR REALS

We first introduce Rosa's specification language and give a high-level overview of the technical challenges and our solutions on a number of examples.

A Real-Valued Specification Language. Rosa is a "verifying" source-to-source compiler that takes as input a program written in a real-valued non-executable specification language and outputs appropriate finite-precision code. The input program is written in a functional subset of the Scala programming language and consists of a number of methods over the `Real` data type. Figures 1, 3, and 4 show three such example methods. Pre- and postconditions (`require` and `ensuring` clauses) allow the user to explicitly state possible absolute errors on method inputs and maximum tolerable absolute errors on the output(s), respectively. Taking into account all uncertainties and their propagation, Rosa chooses a data type from a range of floating-point and fixed-point precisions and emits the corresponding implementation in the Scala programming language.

By writing programs in a real-valued source language, programmers can reason about the correctness of the real-valued algorithm and leave the low-level implementation details to the automated sound analysis in Rosa. Besides this separation of concerns, the real-valued semantics also serves as an unambiguous ideal baseline against which to compute errors. We believe that making numerical errors explicit in pre- and postconditions attached directly to the source code makes it less likely that these will be forgotten or overlooked. Finally, such a language opens the door to *sound* compiler optimizations exploiting properties that are valid over reals but not necessarily over finite-precision—as long as the accuracy specification is satisfied. We leave these, however, to future work and focus here on the error analysis.

Compilation Algorithm. If a method has a full specification (with a pre- and postcondition), then Rosa analyzes the numerical computation and selects a suitable finite-precision data type that fulfills this specification and synthesizes the corresponding code. The user specifies which data types are acceptable from a range of floating-point and fixed-point precisions, ordered by preference or, for example, performance. Rosa searches through the data types, applying a static analysis for each and tries to find the first in the list which satisfies the specification. While this analysis is currently repeated for each data type, parts of the computation can be shared, and we plan to optimize the compilation process in the future.

Rosa can also be used as an analysis tool. By providing one data type (without necessarily a postcondition), Rosa will perform the analysis and report the results consisting of a real-valued range and maximum absolute error of the result to the user. These analysis result are also reported during the regular compilation process, as they may yield useful insights.

Finite-Precision Data-Types. Rosa supports floating-point as well as fixed-point arithmetic with various precisions. We will review here the most important details related to our work; more thorough treatments can be found in Goldberg [1991] and Yates [2013]. An IEEE754 [IEEE 2008] floating-point number consists of a sign bit and a number of exponent and mantissa bits: 8 and 23 and then 11 and 52 for single and double precision, respectively. Higher precisions, such as double-double or quad-double, can be implemented in software. In addition, the floating-point standard also defines special numbers including positive and negative infinity and not a number (NaN). NaN in general signals an invalid operation such as division by zero. The exponent can be positive and negative and, since it is finite, overflow and underflow occur when a value exceeds the representable range. On overflow, positive or negative infinity is returned, whereas underflow results in zero. The large relative error introduced by underflow can be mitigated by introducing denormal numbers. These have a non-standard representation and fill the gap between the smallest standard floating-point number and zero.

The floating-point standard defines several rounding modes of which we consider here (the usually default) rounding to nearest. When numbers are rounded correctly and assuming no denormals occur, the result of a binary floating-point arithmetic operation $\circ_F$ satisfies

$$x \circ_F y = (x \circ_R y)(1 + \delta), \quad |\delta| \leq \epsilon_M, \quad \circ \in \{+, -, *, /\}, \tag{1}$$

where $\circ_R$ is the ideal operation in real numbers and ϵ_M is the machine epsilon that determines the upper bound on the relative error and is precision dependent. Square root satisfies an analogous expression. Rosa uses this abstraction for determining individual roundoff errors. Note that Rosa's static analysis considers ranges of values and as long as that range does not consist solely of denormals, Equation (1) provides a sound way of computing absolute roundoff errors, even when the range includes denormals.

In this work, we regard overflow, a variable range of only denormal numbers and NaNs as faults. Rosa's static analysis determines when these potentially occur and issues a corresponding error message. Our analysis can be straightforwardly extended to handle ranges of only denormal values by augmenting Equation (1); however, we found that in all our examples this case never occurred, and we thus choose the simpler solution.

Floating-point arithmetic requires dedicated support, either in hardware or in software, because exponents and with it the radix points vary dynamically and need to be aligned before arithmetic operations at runtime. Depending on the application, this support may be too costly. An alternative is fixed-point arithmetic, which can be implemented with integers only but which in return requires that the radix point alignments are precomputed at compile time. While no standard exists, fixed-point values are usually represented as (signed) bit vectors with an integer and a fractional part, separated by an implicit radix point. At runtime, the alignments are then performed by bit-shift operations as illustrated in Figure 2, which is the fixed-point implementation of the `sine` function from Figure 1. These shift operations can also be handled by special language extensions for fixed-point arithmetic [ISO/IEC 2008]; they do require, however, the manual specification of the radix point position, which is the challenging part of the compilation.

Fixed-point arithmetic can provide more precision than floating-point arithmetic for the same bit length when the range of values is known and relatively small and thus a large exponent range is not necessary. The unused exponent bits that would be "wasted" by a floating-point representation can be used for more precision in fixed-points.

```
def sineWithError(x: Real): Real = {
  require(x > -1.57079632679 && x < 1.57079632679 && x +/- 1e-11) // preconditio

  x - (x*x*x)/6.0 + (x*x*x*x*x)/120.0 - (x*x*x*x*x*x*x)/5040.0

} ensuring(res => res +/- 1.001e-11) // postcondition
```

Fig. 1. Approximation of sine with a Taylor expansion.

For fixed-point arithmetic, we assume a uniform bit length for the entire program. We further use truncation as the rounding mode for arithmetic operations. The absolute roundoff error at each operation is determined by the fixed-point format, that is, the (implicit) number of fractional bits available. For more details, please see Anta et al. [2010], whose fixed-point semantics we follow. Furthermore, fixed-point arithmetic does not support the square root in a standardized way, as the underlying integers do not provide that operation. If such a function was added as, for example, a library function with an error specification, then Rosa could be straightforwardly extended to support it as well.

2.1. Example 1: Straight-Line Nonlinear Computations

We illustrate Rosa's compilation process on the example in Figure 1, which shows the code of a method that computes the sine function with a seventh-order Taylor approximation. The `Real` data type denotes an ideal real-valued variable without uncertainty or roundoff. The `require` clause or precondition specifies the real-valued range of the input parameter `x` as well as an initial absolute error of 1e-11, which may stem from previous computations or measurement imprecisions.

We would like to make sure that the error on the result does not grow too large, so we constrain the result's absolute error by 1.001e-11. We do not specify a data type preference, so Rosa considers all (currently) supported ones: 8-, 16-, and 32-bit fixed-point arithmetic as well as single, double, double-double, and quad-double floating-point precision (in this order). Rosa determines that 8-bit fixed-point precision potentially overflows and that 16- or 32-bit fixed-point and single-precision floating-point arithmetic is not accurate enough (total error of 2.54e-4, 3.90e-9, and 2.49e-7, respectively). For double floating-point precision, Rosa determines an error of 1.000047e-11, which is sufficiently small so it generates code with the `Double` data type.

If we had specified a somewhat larger error, say, 5e-9, then Rosa would determine that 32-bit fixed-point arithmetic is sufficient and generate the code in Figure 2. (All intermediate results fit into 32 bits, but as we need up to 64 bits to perform the arithmetic operations, we simulate the arithmetic here with the 64-bit `Long` data type.) Note that the generated code includes a precondition, which is checked at runtime and throws an exception if the given condition is violated. The generated precondition checks that inputs satisfy the range bounds for which Rosa has computed error bounds, as otherwise these would not be sound. While the real-valued specification talks about real-valued ranges and error bounds, the generated precondition can (and does) only check the *finite-precision* input ranges (and not the errors), that is, the ranges include the errors but cannot check for them directly.

In order to determine which data type is appropriate, Rosa performs a static analysis that computes a sound estimate of the worst-case absolute error. Since roundoff errors and error propagation depend on the ranges of (all intermediate) computations, Rosa needs to compute these as accurately as possible. We developed a technique that combines interval arithmetic [Moore 1966] with a decision procedure for nonlinear arithmetic, providing accuracy as well as automation (Section 4). Using a decision procedure allows Rosa to take into account arbitrary additional constraints on the inputs,

```
/*
  @param x (x +/- 1.e-11)
  @return ((-1.0003675439019308 <= result && result <= 1.0003675439019308 &&
        (result +/- 3.8907801077969955e-09)))
*/
def sineWithError(x : Long): Long = {
  require(-1.5707963268 <= x && x <= 1.5707963268)

  val _tmp1 = ((x * x) >> 31)
  val _tmp2 = ((_tmp1 * x) >> 30)
  val _tmp3 = ((_tmp2 << 30) / 1610612736l)
  val _tmp4 = ((x << 1) - _tmp3)
  val _tmp5 = ((x * x) >> 31)
  val _tmp6 = ((_tmp5 * x) >> 30)
  val _tmp7 = ((_tmp6 * x) >> 31)
  val _tmp8 = ((_tmp7 * x) >> 31)
  val _tmp9 = ((_tmp8 << 28) / 2013265920l)
  val _tmp10 = ((_tmp4 + _tmp9) >> 1)
  val _tmp11 = ((x * x) >> 31)
  val _tmp12 = ((_tmp11 * x) >> 30)
  val _tmp13 = ((_tmp12 * x) >> 31)
  val _tmp14 = ((_tmp13 * x) >> 31)
  val _tmp15 = ((_tmp14 * x) >> 30)
  val _tmp16 = ((_tmp15 * x) >> 31)
  val _tmp17 = ((_tmp16 << 23) / 1321205760l)
  val _tmp18 = (((_tmp10 << 1) - _tmp17) >> 1)
  val result = _tmp18
}
```

Fig. 2. Sine function implemented in fixed-point arithmetic.

which, for example, neither interval nor affine arithmetic [de Figueiredo and Stolfi 2004] can. Rosa decomposes the total error into roundoff errors and propagated initial errors and computes each with a different method (Section 5). Rosa uses affine arithmetic to keep track of accumulated roundoff errors, while propagated errors are soundly estimated with a first-order Taylor approximation. The latter is fully automatically and accurately computed with our interval and nonlinear decision procedure combination.

2.2. Example 2: Loops with Constant Ranges

In general, numerical errors in loops grow with every iteration, and thus standard abstract interpretation approaches compute a trivial fixpoint of infinity. The only alternative today to compute a sound absolute error bound for complex code is by unrolling. It turns out, however, that our separation of errors into roundoff and propagation errors allows us to express the total error as a function of the number of loop iterations. We have further identified a class of loops for which we can derive a closed-form expression of the loop error bounds. This expression, on one hand, constitutes an inductive invariant and, on the other hand, can be used to compute concrete error bounds. While this approach is limited to loops where the variable ranges are bounded, our experiments show that this approach can already analyze interesting loops that are out of reach for current tools.

Figure 3 shows such an example: a Runge-Kutta order-2 simulation of a pendulum. `t` and `w` are the angle the pendulum forms with the vertical and the angular velocity,

```
def sine(x: Real): Real = {
  require(-20 <= x && x <= 20)
  x - x*x*x/6 + x*x*x*x*x/120
}

def pendulum(t: Real, w: Real, n: LoopCounter): (Real, Real) = {
  require(-2 <= t && t <= 2 && -5 <= w && w <= 5 &&
    -2.01 <= ~t && ~t <= 2.01 && -5.01 <= ~w && ~w <= 5.01)

  if (n < 100) {
    val h: Real = 0.01
    val L: Real = 2.0
    val m: Real = 1.5
    val g: Real = 9.80665

    val k1t = w
    val k1w = -g/L * sine(t)
    val k2t = w + h/2*k1w
    val k2w = -g/L * sine(t + h/2*k1t)
    val tNew = t + h*k2t
    val wNew = w + h*k2w

    pendulum(tNew, wNew, n++)
  } else {
    (t, w)
  }
}
```

Fig. 3. Simulation of a pendulum.

```
def jetApproxGoodFitWithError(x: Real, y: Real): Real = {
 require(-5<=x && x<=5 && -5<=y && y<=5 && x +/- 0.001 && y +/- 0.001)
  if (y < x) {
    -0.317581 + 0.0563331*x + 0.0966019*x*x + 0.0132828*y +
     0.0372319*x*y + 0.00204579*y*y
  } else {
    -0.330458 + 0.0478931*x + 0.154893*x*x + 0.0185116*y -
     0.0153842*x*y - 0.00204579*y*y
  }
}
```

Fig. 4. Approximation of a complex embedded controller.

respectively. We approximate the sine function with its fifth-order Taylor series polynomial. We focus here on *roundoff* errors between the system following the real-valued execution and the system following the same dynamics but implemented in finite precision (we do not attempt to capture truncation errors due to the numerical integration or due to the Taylor approximation of sine). After 100 iterations, for instance, Rosa determines that the error on the results is at most 8.82e-14 and 1.97e-13 (for `t` and `w`, respectively) when implemented in double-precision floating-point arithmetic and 7.38e-7 and 1.65e-6 in 32-bit fixed-point arithmetic.

2.3. Example 3: Discontinuities

Embedded systems often use piecewise approximations of more complex functions. In Figure 4, we show a possible piecewise polynomial approximation of a fairly complex jet

```
P ::= def mName(args): res = {
        require(⋀ A)
        ( L | D )
      } ensuring(res => ⋀ A)
A ::= x +/- const | S
S ::= S ∧ S | S ∨ S | ¬ S | C
L ::= if (n < const) mName(B, n + 1) else args
D ::= if (C) D else D | B
B ::= val x = F; B | (F, F, ...) | F
F ::= - F | F + F | F - F | F ∗ F | F / F | √F | X
C ::= F ≤ F | F < F | F ≥ F | F > F
X ::= x | const
```

Fig. 5. Rosa's input language.

engine controller. We obtained this approximation by fitting a polynomial to a sample of values of the original function. Note that the resulting function is not continuous.

A precise constraint encoding the difference between the real-valued and finite-precision computation when they take different paths includes variables that are tightly correlated. This makes it hard for SMT solvers to cope with and makes linear approaches imprecise. We explore the separation of errors idea again in order to soundly estimate errors due to conditional branches and separate the real-valued difference from finite-precision artifacts. The individual error components are easier to handle individually yet preserve enough accuracy.

In our example, the real-valued difference between the two branches is bounded by 0.0428 (making it arguably a reasonable approximation given the large possible range of results). However, this is not a sound estimate for the discontinuity error in the presence of roundoff and initial errors (in our example 0.001). With Rosa, we can confirm that the discontinuity error is bounded by 0.0450 with double floating-point precision, with all errors taken into account.

3. PROBLEM DEFINITION

Clearly, computing error bounds is the main technical challenge of our compiler for `Reals`. Before we describe our solution in detail, we first precisely define the error computation problem that Rosa is solving internally.

Formally, an input program consists of one or more methods given by the grammar in Figure 5. `args` denotes possibly multiple arguments, and the return value (`res`) can be a single value or a tuple (denoted, e.g., as `(x, y)`). The method body may consist of an arithmetic expression with the operators $+, -, *, /, \sqrt{}$, as well as immutable variable declarations (`val t1 = ...`), non-recursive function calls, and possibly nested conditionals. Note that the square root is only supported for floating-point arithmetic. Non-recursive method calls are handled either by inlining the postcondition or the whole method body and by proving the corresponding precondition. The specification language is functional, so we represent loops as recursive functions (denoted `L`), where `n` denotes the integer loop iteration counter.

The precondition given by the **`require`** clause defines a range bound for each input variable as `a <= x && x <= b` representing $x \in [a, b]$. Optionally, initial absolute errors can be specified with the `+/-` operator. If no errors are given explicitly, then we assume roundoff as the initial error. Additionally, the `require` clause may specify further constraints on the inputs, such as `x*x + y*y <= 20.0`.

Overall Problem Definition. Let us denote by P a real-valued function representing our program and by x its input, where x is possibly multivariate. Denote by $\tilde{P}$ the

corresponding finite-precision implementation of the program that has the same syntax tree but with operations interpreted in finite-precision arithmetic. Let $\tilde{x}$ denote the input to this finite-precision program. The technical challenge of Rosa is to estimate the difference

$$\max_{x,\tilde{x}} \ |P(x) - \tilde{P}(\tilde{x})|, \tag{2}$$

that is, the worst-case absolute error on the result of the program. This error bound is crucial for selecting an implementation data type.

The error bound `x +/- λ` defines the relationship $|x - \tilde{x}| \leq \lambda$. The domains of x and $\tilde{x}$, over which Equation (2) is to be evaluated, are given by the precondition in the **`require`** clause. The real-valued range bounds $x_i \in [a_i, b_i]$ are given directly and the finite-precision ones $\tilde{x}_i \in [c_i, d_i] = [a_i - \lambda, b_i + \lambda]$ are derived from the real-valued range and the error bound.

Straight-line Nonlinear Code (F). When P consists of a nonlinear arithmetic expression only, the syntactic program corresponds to a real-valued mathematical expression that is the input to our core error computation procedure. Concretely, the input consists of a real-valued function $f : \mathbb{R}^m \to \mathbb{R}^n$ over some inputs $x_i \in \mathbb{R}$, representing the arithmetic expressions `F`, with corresponding finite-precision counter-parts $\tilde{f}$ and $\tilde{x}$. Note that for our analysis all variables are real valued; the finite-precision variable $\tilde{x}$ is considered as a noisy version of x. Rosa performs the error computation with respect to some fixed target precision in floating-point or fixed-point arithmetic; this choice provides error bounds for each individual arithmetic operation. Then Equation (2) reduces to bounding the absolute error on the result of evaluating $f(x)$ in finite precision arithmetic (Section 5): $\max_{x,\tilde{x}} |f(x) - \tilde{f}(\tilde{x})|$.

Loops (L). When the body of P is a loop, then the constraint reduces to computing the overall error after k-fold iteration f^k of f, where f corresponds to the arithmetic expression of the loop body. We define for any function H: $H^0(x) = x$, $H^{k+1}(x) = H(H^k(x))$. We are then interested in bounding (Section 6):

$$\max_{x,\tilde{x}} |f^k(x) - \tilde{f}^k(\tilde{x})|.$$

Conditionals (D). For code containing branches, Equation (2) accounts also for the *discontinuity error*. For example, if we let f_1 and f_2 be the real-valued functions corresponding to the **`if`** and the **`else`** branch, respectively, with the branch condition c, then, if $c(x) \wedge \neg\tilde{c}(\tilde{x})$, the discontinuity error is given by $\max_{x,\tilde{x}} |f_1(x) - \tilde{f}_2(\tilde{x})|$, that is, it accounts for the case where the real computation takes the if-branch, and the finite-precision one takes the else-branch. The overall error on P from Equation (2) in this case must account for the maximum of discontinuity errors between all pairs of paths, as well as propagation and roundoff errors for each path (Section 7).

A note on relative error. Our technique soundly overestimates the absolute error of the computation. The relative error can be computed from this and from the range of the result provided that the range does not include zero. Whenever this is the case, Rosa also reports the relative error in addition to the absolute one.

3.1. Assumptions and Extensions

While the input and output language is a subset of Scala, the analysis is programming language agnostic, as long as the IEEE 754 standard is supported and the regular language compiler preserves the computation order. Adapting Rosa to a different front-end or back-end is a straightforward programming exercise.

Rosa currently supports analysis and code generation for floating-point and fixed-point arithmetic of different precisions; currently, these are 8-, 16-, and 32-bit

fixed-point arithmetic as well as single, double, double-double, and quad-double floating-point precision, where the latter two are implemented as software libraries [Bailey et al. 2013]. Extending Rosa to other or non-standard precisions is straightforward, provided that roundoff error bounds are given for each arithmetic operation (they can depend on the variable's ranges).

We further assume a uniform precision throughout the program, although an extension to mixed-precision would be straightforward, provided that the types for each variable are given.

4. COMPUTING RANGES ACCURATELY

The first step to accurately estimating roundoff and propagation errors is to have a procedure to estimate ranges as tightly as possible. This is important, as these errors directly depend on the ranges of all, including intermediate, values. Coarse range estimates may thus result in inaccurate overall error bounds or they can make the analysis impossible due to spurious divisions by zero, for example.

4.1. Interval and Affine Arithmetic

Traditionally, sound or guaranteed computations have been performed with interval arithmetic [Moore 1966]. Interval arithmetic computes a bounding interval for each basic operation as

$$x \circ y = [min(x \circ y), max(x \circ y)] \qquad \circ \in \{+, -, *, /\}$$

and analogously for the square root. Interval arithmetic cannot track correlations between variables (e.g., $x - x \neq 0$) and thus can introduce significant over-approximations of the true ranges, especially when the computations are longer.

Affine arithmetic [de Figueiredo and Stolfi 2004] partially addresses this loss of correlation by representing possible values of variables as affine forms

$$\hat{x} = x_0 + \sum_{i=1}^{n} x_i \epsilon_i,$$

where x_0 denotes the *central value* (of the represented interval) and each *noise symbol* ϵ_i, ranging over $[-1, 1]$, is a formal variable denoting a deviation from this central value. The maximum magnitude of each *noise term* $x_i \epsilon_i$ is given by the corresponding x_i.

The range represented by an affine form is computed as

$$[\hat{x}] = [x_0 - rad(\hat{x}), x_0 + rad(\hat{x})], \qquad rad(\hat{x}) = \sum_{i=1}^{n} |x_i|.$$

Note that the sign of the x_is does not matter in isolation, it does, however, reflect the relative dependence between values. For example, take $x = x_0 + x_1\epsilon_1$, then

$$x - x = x_0 + x_1\epsilon_1 - (x_0 + x_1\epsilon_1) = x_0 - x_0 + x_1\epsilon_1 - x_1\epsilon_1 = 0.$$

If we subtracted $x' = x_0 - x_1\epsilon_1$ instead, then the resulting interval would have width $4 * x_1$ and not zero. Linear operations are performed termwise and are computed exactly, whereas nonlinear ones need to be approximated. We use linear approximations as proposed in de Figueiredo and Stolfi [2004] and as implemented in Darulova and Kuncak [2011].

Affine arithmetic can thus track linear correlations; it is, however, not generally better than interval arithmetic. For example, consider $x * y$, where $x = [-5, 3]$, $y = [-3, 1]$.

```
def getRange(expr, precondition, precision, maxIterations):
  z3.assertConstraint(precondition)
  [aInit, bInit] = evalInterval(expr, precondition.ranges);

  //lower bound
  if z3.checkSat(expr < aInit + precision) == UNSAT
    a = aInit
    b = bInit
    numIterations = 0
    while (b-a) < precision ∧ numIterations < maxIterations
      mid = a + (b - a) / 2
      numIterations++
      z3.checkSat(expr < mid) match
        case SAT ⇒ b = mid
        case UNSAT ⇒ a = mid
        case Unknown ⇒ break
    aNew = a
  else
    aNew = aInit

  //upper bound symmetrically
  bNew = ...
  return: [aNew, bNew]
```

Fig. 6. Algorithm for computing the range of an expression.

In interval arithmetic, the result is $[-9, 15]$, whereas for affine arithmetic it is $[-13, 15]$:

$$\begin{aligned}(-1 + 4\epsilon_1) * (-1 + 2\epsilon_2) &= 1 - 4\epsilon_1 - 2\epsilon_2 + 8\epsilon_1\epsilon_2\\ [x * y] &= [1 - 14, 1 + 14] = [-13, 15].\end{aligned}$$

4.2. Range Estimation Using Satisfiability Modulo Theories Solvers

While interval and affine arithmetic are reasonably fast for range estimation, they tend to introduce over-approximations, especially if the input intervals are not sufficiently small. We propose a combination of interval arithmetic and a decision procedure for nonlinear arithmetic, implemented in a SMT constraint solver, to obtain improved accuracy while maintaining automation.

Figure 6 shows our algorithm for computing the lower bound of a range. The computation for the upper bound is symmetric. For each range to be computed, our technique computes an initial sound estimate of the range with interval arithmetic. It then performs an initial quick check to test whether the computed first approximation bounds are already tight. If not, then it uses the initial approximation as the starting point and narrows down the lower and upper bounds using a binary search. At each step of the binary search our tool uses the nonlinear nlsat solver within Z3 [De Moura and Bjørner 2008; Jovanović and de Moura 2012] to confirm or reject the newly proposed bound. The search stops when either Z3 returns unknown or times out, the difference between subsequent bounds is smaller than a precision threshold, or the maximum number of iterations is reached. This stopping criterion can be adjusted by the user.

Additional Constraints. In our approach, since we are using Z3 to check the soundness of range bounds, we can assert additional constraints and run the algorithm in Figure 6 with respect to all of these. This is especially useful when taking into account branch conditions from conditionals.

Optimizations. Calling an SMT solver is fairly expensive, so we want to minimize the number of calls. The algorithm in Figure 6 presents parameters to do this: the maximum number of iterations and the precision of the range estimate. Through our experiments we have identified suitable default values, which seem to present a good tradeoff between accuracy and performance. In addition to these two parameters, if we are only interested in the final range, we do not need to call Z3 and the algorithm in Figure 6 for every intermediate expression but use interval arithmetic only instead. In principle, we could call Z3 only on the full final expression; however, we found that this resulted in suboptimal results as this expression often was too complex and Z3 would time out. We found a good compromise in calling Z3 only every 10 arithmetic operations. All of these parameters can be adjusted by the user.

5. SOUNDLY ESTIMATING NUMERICAL ERRORS IN NONLINEAR EXPRESSIONS

Now we can address the first challenge of error estimation for a loop-free nonlinear function without branches: $\max_{x,\tilde{x}} |f(x) - \tilde{f}(\tilde{x})|$, where $|x - \tilde{x}| \leq \lambda$ and where the ranges for x and $\tilde{x}$ are given by the precondition. We will assume $f : \mathbb{R}^m \to \mathbb{R}$ for simplicity of exposition, but the approach extends straightforwardly to $f : \mathbb{R}^m \to \mathbb{R}^n$ by computing the error for each output individually.

5.1. Error Estimation with Affine Arithmetic

Consider first the case when want to estimate roundoff errors only; that is, we are interested in $\max_x |f(x) - \tilde{f}(x)|$, where the input errors are zero. Our procedure executes the computation represented by f abstractly by computing an interval and an affine form for each AST node:

```
(range: Interval, erˆr: AffineForm),
```

where `range` represents the real-valued range and $er\hat{r}$ the accumulated worst-case errors, with essentially one noise term for each roundoff error (together with artifacts from nonlinear approximations). The finite-precision range is then given by `range` + $[er\hat{r}]$, where $[er\hat{r}]$ denotes the interval represented by the affine form.

For each arithmetic operation, our procedure

(1) computes the new range with our range procedure from Section 4,
(2) propagates already accumulated roundoff errors, and
(3) computes the new roundoff error, which is then added to the propagated affine form.

Steps 2 and 3 are explained below. Since we compute the range at each intermediate node, we also check for possible overflows, ranges containing only denormals, division by zero, or negative square-root discriminants without extra effort.

Propagation of Errors with Affine Arithmetic (Step 2). For linear operations, errors are propagated with the standard rules of affine arithmetic. For multiplication, division, and square root, the error propagation depends on the range of values, so we have to adapt our computation to use the ranges computed with our Z3-backed procedure. In the following, we denote the real range of a variable x by $[x]$ and its associated error by the affine form $er\hat{r}_x$. When we write $[x] * er\hat{r}_y$, we mean that the interval $[x]$ is converted into an affine form and the multiplication is performed in affine arithmetic. Multiplication is then computed as

$$\begin{aligned} x * y &= ([x] + er\hat{r}_x)([y] + er\hat{r}_y) \\ &= [x] * [y] + [x] * er\hat{r}_y + [y] * er\hat{r}_x + er\hat{r}_x * er\hat{r}_y. \end{aligned}$$

Thus the first term contributes to the ideal range and the remaining three to the error affine form. The larger the factors $[x]$ and $[y]$ are, the larger the computed errors will

be, so a tight range estimation is important for accuracy. Division is computed as

$$\frac{x}{y} = x * \frac{1}{y} = ([x] + e\hat{r}r_x)([1/y] + er\hat{r}_{1/y})$$
$$= [x] * \left[\frac{1}{y}\right] + [x] * e\hat{r}r_{\frac{1}{y}} + \left[\frac{1}{y}\right] * e\hat{r}r_x + e\hat{r}r_x * e\hat{r}r_{\frac{1}{y}}$$

For square root, we first compute a linear approximation of the square root as in our previous work [Darulova and Kuncak 2011],

$$\sqrt{x} = \alpha * x + \zeta + \theta,$$

and then perform the affine multiplication component-wise.

Roundoff Error Computation (Step 3). Roundoff errors for floating-point arithmetic are computed at each computation step using the abstraction from Equation (1) and the `totalRange`, which is the real-valued range computed in step 1 plus the propagated roundoff errors from step 2: ϵ_m `* maxAbs(totalRange)`, where ϵ_m is the machine epsilon (and not an affine arithmetic symbolic variable) and where the `maxAbs` function computes the maximum absolute value of a range. For fixed-point arithmetic, the variable range determines how many bits are required for the integer part to avoid overflow and thus how many bits remain for the fractional part. This in turn determines the roundoff error [Anta et al. 2010]. The new roundoff error is then added to $e\hat{r}r$ as a fresh noise term.

Note that for a new data type, only this roundoff computation needs to be modified.

5.2. Separation of Errors

We could use the affine arithmetic-based procedure to track all errors, not only roundoff errors, by simply adding the initial error as a fresh noise term at the beginning. Such an approach treats all errors equally: The initial errors are propagated in the same way as roundoff errors that are committed during the computation. We found, however, that the over-approximation introduced by affine arithmetic for nonlinear computations increases substantially as the magnitude of the noise terms (i.e., the errors) becomes larger. Instead, we separate the total error as follows:

$$\begin{aligned} |f(x) - \tilde{f}(\tilde{x})| &= |f(x) - f(\tilde{x}) + f(\tilde{x}) - \tilde{f}(\tilde{x})| \\ &\leq |f(x) - f(\tilde{x})| + |f(\tilde{x}) - \tilde{f}(\tilde{x})|. \end{aligned} \tag{3}$$

The first term, $|f(x) - f(\tilde{x})|$, captures the error on the result of f caused by the initial error between x and $\tilde{x}$. The second term, $|f(\tilde{x}) - \tilde{f}(\tilde{x})|$, covers the roundoff error committed when evaluating f in finite precision but without an input error. Thus, we separate the overall error into the propagation of existing or initial errors and the newly committed roundoff errors.

We denote by $\sigma_f : \mathbb{R}^m \to \mathbb{R}$ the function that returns the maximum absolute roundoff error committed when evaluating an expression f in finite-precision arithmetic: $\sigma_f(\tilde{x}) = |f(\tilde{x}) - \tilde{f}(\tilde{x})|$. We omit the subscript $_f$, when it is clear from the context. Furthermore, $g : \mathbb{R}^m \to \mathbb{R}$ denotes a function that bounds the difference in f, given a difference in its inputs: $|f(x) - f(y)| \leq g(|x - y|)$. When $m > 1$, the absolute values are component-wise, for example, $g(|x_1 - y_1|, \ldots, |x_m - y_m|)$, but when it is clear from the context, we will write $g(|x - y|)$ for clarity. Both σ and g are understood to work over an implicit input domain, given by the precondition; we will drop the $\max_{x,\tilde{x}}$ when it is clear from the context. Thus, the overall worst-case numerical error is given by

$$|f(x) - \tilde{f}(\tilde{x})| \leq g(|x - \tilde{x}|) + \sigma(\tilde{x}). \tag{4}$$

The function σ is instantiated with the affine arithmetic-based procedure from Section 5.1. Since roundoff errors are generally small, we found affine arithmetic suitable for this purpose. In contrast, the propagation of existing errors (function g) is a continuous real-valued property that depends on the slope of the function f. In order to compute this as accurately as possible, we need to capture the end-to-end correlations between variables and thus have to look at the function as a whole. We describe this procedure next.

5.3. Propagation Errors

We instantiate Equation (4) with $g(x) = K \cdot x$, that is, $|f(x) - f(y)| \leq K|x - y|$, which bounds the deviation on the result due to a difference in the input by a linear function in the input errors. The constant K (or vector of constants K_i in the case of a multivariate function) is to be determined for each function f individually and is usually called the Lipschitz constant. We will use the in this context more descriptive name *propagation coefficient*. Note that we need to compute the propagation coefficient K for the mathematical function f and not its finite-precision counterpart $\tilde{f}$.

Error amplification or diminution depends on the derivative of the function evaluated at the value of the inputs. The steeper the function, that is, the larger the derivative, the more initial errors are magnified. For $f : \mathbb{R}^m \to \mathbb{R}$, we have

$$\max_{x,\tilde{x}} |f(x) - f(\tilde{x})| \leq \sum_{i=1}^{m} K_i \lambda_i, \qquad \text{where } K_i = \sup_{x,\tilde{x}} \left| \frac{\partial f}{\partial w_i} \right|, \tag{5}$$

where λ_i are the initial errors and w_i denote the formal parameters of f. In order words, the propagation coefficients are computed as a sound bound on the Jacobian. This computation naturally extends component-wise to multiple outputs.

We formally derive the computation of the propagation coefficients K_i for a multivariate function $f : \mathbb{R}^m \to \mathbb{R}$ in the following. Let $h : [0, 1] \to \mathbb{R}$ such that $h(\theta) := f(y + \theta(z - y))$. Without loss of generality, assume $y < z$. Then $h(0) = f(y)$ and $h(1) = f(z)$ and $\frac{d}{d\theta} h(\theta) = \nabla f(y + \theta(z - y)) \cdot (z - y)$. By the mean value theorem: $f(z) - f(y) = h(1) - h(0) = h'(\zeta)$, where $\zeta \in [0, 1]$,

$$\begin{aligned}
|f(z) - f(y)| &= |h'(\zeta)| = |\nabla f(y + \zeta(z - y)) \cdot (z - y)| \\
&= \left| \left(\left. \frac{\partial f}{\partial w_1} \right|_s, \ldots, \left. \frac{\partial f}{\partial w_m} \right|_s \right) \cdot (z - y) \right|, \quad s = y + \zeta(z - y) \\
&= \left| \frac{\partial f}{\partial w_1} \cdot (z_1 - y_1) + \cdots + \frac{\partial f}{\partial w_m} \cdot (z_m - y_m) \right| \\
&\leq \sum_{i=1}^{m} \left| \frac{\partial f}{\partial w_i} \right| \cdot |z_i - y_i| \qquad (**),
\end{aligned}$$

where the partial derivatives are evaluated at $s = y + \zeta(z - y)$ (which we omit for readability). The value of s in (**) is constrained to be in $s \in [y, z]$, so for a sound analysis we have to determine the maximum absolute value of the partial derivative over $[y, z]$. y and z in our application range over the values of x and $\tilde{x}$, respectively, so we compute the maximum absolute value of $\frac{\partial f}{\partial x_i}$ over all possible values of x and $\tilde{x}$. With $|y_i - z_i| \leq \lambda_i$ we finally obtain Equation (5).

Bounding Partial Derivatives. Rosa first computes the partial derivatives in Equation (5) symbolically and then soundly bounds them over all possible values of x and $\tilde{x}$. Both interval and affine arithmetic suffer from possibly large over-approximations due to nonlinearity and loss of correlations. Furthermore, they cannot take additional

constraints into account, for example, from branch conditions (e.g., y < x) or user-defined constraints on the inputs. We use the range computation from Section 5 that allows us to take these into account, making the computed propagation coefficients much more accurate.

Recall that the arithmetic operations permitted are $\{+, -, *, /, \sqrt{}\}$, and thus the derivatives could be in principle discontinuous or undefined. Rosa detects these cases automatically during the range bound computation and reports them as errors.

Sensitivity to Input Errors. Beyond providing a way to compute the propagated initial errors, Equation (5) also makes the sensitivity of the function f to input errors explicit. The user can use this knowledge, for example, to determine which inputs need to be obtained more precisely, for example, by better measurements. We report the values of K back to the user.

5.4. Relationship with Affine Arithmetic

Both our presented propagation procedure and propagation using affine arithmetic perform approximations. The question then arises: When is it preferable to use one over the other? Our experience and experiments show empirically that for longer nonlinear computations, error propagation based on Lipschitz continuity gives better results, whereas for shorter and linear computations this is not the case. In this section, we present an analysis of this phenomenon based on a small example.

Suppose we want to compute $x * y - x^2$. For this discussion, we consider propagation only and disregard roundoff errors. We consider the case where x and y have an initial error of $\delta_x\epsilon_1$ and $\delta_y\epsilon_2$, respectively, where $\epsilon_i \in [-1, 1]$ are the formal noise symbols of affine arithmetic. Without loss of generality, we assume $\delta_x, \delta_y \geq 0$. We first derive the expression for the error with affine arithmetic, using the definition of multiplication from Section 5.1. We denote by $[x]$ the evaluation of the *real-valued* range of the variable x. The total range of x is then the real-valued range plus the error: $[x] + \delta_x\epsilon_1$, where $\epsilon_1 \in [-1, 1]$. Multiplying out, and removing the $[x][y] - [x]^2$ term (since it is not an error term), we obtain the expression for the error of $x * y - x^2$:

$$\begin{aligned}&\big([y]\delta_x\epsilon_1 + [x]\delta_y\epsilon_2 + \delta_x\delta_y\epsilon_3\big) - (2[x]\delta_x\epsilon_1 + \delta_x\delta_x\epsilon_4)\\ &\quad = ([y] - 2[x])\delta_x\epsilon_1 + [x]\delta_y\epsilon_2 + \delta_x\delta_y\epsilon_3 + \delta_x\delta_x\epsilon_4,\end{aligned} \tag{6}$$

ϵ_3 and ϵ_4 are fresh noise symbols introduced by the nonlinear approximation.

Now we use our second method and compute the propagation coefficients:

$$\frac{\partial f}{\partial x} = y - 2x \qquad \frac{\partial f}{\partial y} = x,$$

so the error is given by

$$|[y + \delta_y\epsilon_2 - 2(x + \delta_x\epsilon_1)]|\delta_x + |[x + \delta_x\epsilon_1]|\delta_y. \tag{7}$$

We obtain this expression by instantiating Equation (**) with the expressions representing the ranges of x and y. Note that these ranges include the errors.

Multiplying out Equation (7), we obtain the following:

$$|[y - 2x]|\delta_x + |[x]|\delta_y + 2\delta_x\delta_y + \delta_x\delta_x. \tag{8}$$

In summary, with affine arithmetic, we need to compute the ranges used for propagation at each computation step, that is, in Equation (6), we compute $[x]$ and $[y]$ separately. In contrast, with the Lipschitz-based technique, we evaluate $[y - 2x]$ and $[x]$, where we can take all correlations between the variables x and y into account. It is these correlations that improve the computed error bounds. For instance, if we choose $x \in [1, 5]$ and $y \in [-1, 2]$ and we, say, know that $x < y$, then by a stepwise

computation we obtain $[y] - 2[x] = [-1, 2] - 2[1, 5] = [-11, 0]$, whereas taking the correlations into account, we can narrow down the range of x to $[1, 2]$ and obtain $[y - 2x] = [-1, 2] - 2[1, 2] = [-5, 0]$. Hence, since we compute the maximum absolute value of these ranges for the error computation, affine arithmetic will use the factor 11, whereas our Lipschitz-based approach will use 5.

But, comparing Equation (8) with Equation (6), we also see that the term $\delta_x \delta_x$ is included twice with our Lipschitz-based approach, whereas in the affine propagation it is only included once. We conclude that a Lipschitz-based error propagation is most useful for longer computations where it can leverage correlations. In other cases, we keep the existing affine arithmetic-based technique. It does not require a two-step computation, so we want to use it for smaller expressions. We remark that for linear operations the two approaches are equivalent.

5.5. Refactoring

As discussed, affine arithmetic works best for computing error bounds when the expression is shorter and the errors are small, whereas propagation coefficients are preferable otherwise. We use this principle inside Rosa not only to distinguish between roundoff errors and the method's input errors but also inside the method body if it contains local variable declarations. For example, consider the following code snippet:

```
val a = f1(x)
f2(x, a)
```

where `a` is declared locally, and becomes an input variable to the computation in f_2. Similarly, the errors on `a` become input errors in f_2. Rosa will thus compute the errors on `a` by using propagation coefficients to compute the propagation and affine arithmetic to compute the roundoff errors. When computing the error on f_2, the total error on `a` is propagated together with the error on `x` with propagation coefficients. The roundoff errors from computing `a` have thus become initial errors for the computation of f_2. In practice, this strategy often leads to improved error bounds, as it keeps the expressions treated by affine arithmetic short. We currently rely on the programmer to refactor the code, but this could also be automated in the future.

5.6. Implementation

We have implemented Rosa in the Scala programming language. Internally, we use a rational data type implemented on top of Java's `BigIntegers` for all our computations. This lets us easily interface with Z3, which also uses rationals, and also to avoid having to deal with roundoff errors internally.

5.7. Comparison with the State-of-the-Art

We compare Rosa to two other tools that can automatically quantify numerical errors: Fluctuat [Goubault and Putot 2011] (version from January 2015) and FPTaylor [Solovyev et al. 2015] (version from April 2015).

Fluctuat is an abstract interpreter that uses affine arithmetic for computing both the ranges of variables and for the error bounds. In order to combat the overapproximations introduced by affine arithmetic, Fluctuat can add constraints on noise terms [Ghorbal et al. 2010]. Further, Fluctuat uses Taylor approximations locally to handle division and square root [Ghorbal et al. 2009], but the expansion is hard coded and does not consider the global expression.

Another technique employed by Fluctuat is interval subdivision, where the user can designate up to two variables in the program whose ranges will be subdivided and analyzed separately and the results then merged. This procedure works for floating-point arithmetic, as the decimal point is dynamic. For fixed-point arithmetic, however, the global ranges are needed at each point to determine the static fixed-point

formats. These can be obtained with interval subdivision, but, because the errors cannot be evaluated over the smaller ranges, the benefits will be reduced. Naturally, interval subdivision increases the runtime of the analysis, especially for multivariate functions, and the optimal subdivision strategy may not always be obvious. Interval subdivision could be also used in place of our SMT-supported range computation; the procedure would, however, not be able to take into account arbitrary correlations between variables or additional constraints. We choose here to compare our SMT-based technique against Fluctuat with and without subdivision to obtain a good comparison between the techniques. In the future, we expect a combination of different techniques to work best.

Fluctuat also has a procedure for computing discretization errors and can handle loops either by computing fixpoints or by unrolling. Finally, Fluctuat also separates errors similarly to our presented approach, although it does not treat the individual parts fundamentally differently, as we do. We want to note that our formalism has also enabled the unified treatment of loops and discontinuities.

FPTaylor [Solovyev et al. 2015] is a recent tool for computing the roundoff errors of nonlinear expressions, including transcendental functions. It relies, similarly to Rosa, on Taylor series but does the expansion with respect to errors, whereas we expand with respect to the function's parameters. Furthermore, FPTaylor uses global optimization as the backend solver, which enables the use of transcendental functions (Z3's nlsat solver only supports arithmetic). FPTaylor currently only supports error computation for straight-line computations in floating-point arithmetic.

Like Rosa, both Fluctuat and FPTaylor also compute relative errors from absolute errors, whenever the resulting range does not straddle zero.

Another framework that can be used for estimating numerical errors is the Frama-C framework [CEA-LIST 2015] with the Gappa front-end [Boldo and Marché 2011; Linderman et al. 2010]. Gappa supports automated roundoff error computation based on interval or affine arithmetic and can additionally take hints from the user to enable proving very precise properties. In this article, we focus on automated techniques, and, since those inside Gappa are subsumed by Fluctuat, we choose to compare Rosa only against the latter.

Some of the techniques used by Rosa and Fluctuat have also been employed in the area of synthesis of fixed-point programs, that is, the determination of the number of integer and fractional bits needed [Mallik et al. 2007; Jha and Seshia 2013; Lee et al. 2006; Gaffar et al. 2004; Jha and Seshia 2013; Kinsman and Nicolici 2009]. These are, however, mostly specific to fixed-point arithmetic, and we thus review them further in Section 8.

5.8. Experimental Results

We have chosen a number of benchmarks from the domains of scientific computing and embedded systems [Anta et al. 2010; Woodford and Phillips 2012] to evaluate the accuracy and performance of our technique. The tool and all benchmarks are open source and available at https://github.com/malyzajko/rosa.

We perform all tests in double floating-point precision, as this is the only precision supported by both Fluctuat and FPTaylor. In our experience, while the absolute errors naturally change with varying precisions and data types, relative differences when comparing different tools on the same precision data type remain similar. This is because the differences between analyzing absolute errors for fixed-point or floating-point arithmetic with techniques like ours are minor.

Experiments were performed on a desktop computer running Ubuntu 14.04.1 with a 3.5GHz i7 processor and 16GB of RAM and using the unstable branch (as of December 10, 2014) of Z3.

All three tools have a number of settings that affect the tradeoff between accuracy and performance, and it is impossible for us to capture them all. In our experiments, we have thus chosen to focus on accuracy first, that is, we tried to choose the setting that resulted in the tightest error bounds, unless the performance suffered disproportionately by this choice.

Table I shows our experimental results in terms of accuracy (absolute errors computed) and performance (running time of tool). Running times for Rosa and FPTaylor are rounded to two decimal digits. For Fluctuat, however, we report times rounded to seconds, because Fluctuat itself does not report the total running times, and we had to run the analysis via a GUI that made more precise timing measurements infeasible. We consider three flavors of our benchmarks: inputs with roundoff errors only, inputs with initial larger uncertainties, and inputs with an additional nonlinear constraint.

Inputs with Roundoff. In the first set of benchmarks in Table I we assume only roundoff as the initial error on inputs. We compare against Fluctuat without and with subdivisions. For the subdivisions, we uniformly chose 20 subdivisions for the two inputs where the effect was largest. While choosing more is certainly possible, we found the running time increased rapidly and disproportionately with the accuracy gains. For FPTaylor, we used default settings with the improved rounding model, approximate optimization, and the branch and bound optimization procedure, which we believe are the most accurate settings. The annotation "(ref)" marks benchmarks that are refactored. Rosa's technique in general benefits from such a refactoring (see Section 5.5), but this is also sometimes the case for Fluctuat when subdivisions are used.

FPTaylor is mostly able to compute the tightest error bounds on these benchmarks, but we observe that the differences (except for the jet example) are in many cases quite small. FPTaylor's computation is also the most time consuming in the majority of cases, although again the differences are often small.

Inputs with Uncertainties. The second set of benchmarks features inputs with uniform uncertainty of 1e-11, aiming to compare the different tools ability to estimate error propagation accurately. The tools' settings are the same as for the first set of testcases. Except for the jet example, which is difficult for Z3, Rosa computes essentially as tight error bounds as FPTaylor with a smaller running time.

Inputs with Nonlinear Constraint. For the last set of benchmarks, we have constrained the inputs with a nonlinear constraint of the form $x*x+y*y+z*z < c$, where x, y, z are input variables and c is a meaningful benchmark-specific constant. This constraint is representative of constraints that cannot be captured by a linear technique like affine arithmetic. In Rosa, this constraint can be specified naturally in the precondition. In Fluctuat, it is possible to enclose the computation in an if-condition (`if (constr) ...`) and the affine terms will be constrained with a linearized branch condition. We used the "Constraints on noise symbols" setting. FPTaylor provides syntax to specify additional constraints; however, these are only supported with Z3 as the backend, and hence without the improved rounding model. We observe that no one tool consistently provides the most accurate error estimates, but that FPTaylor's technique turns out to be quite expensive in this case.

6. LOOPS

We have identified a class of loops for which the propagation of errors idea allows us to express the numerical errors as a function of the number of iterations. Concretely, we assume a single non-nested loop without conditional branches for which the ranges of variables are bounded and fixed statically. We do not attempt to prove that ranges are preserved across loop iterations; we leave the discovery of suitable inductive invariants

Table I. Absolute Errors Computed by Rosa, Fluctuat, and FPTaylor for Double-Precision Floating-Point Arithmetic. (r) Marks Refactored Benchmarks and (e) Marks Benchmarks with Additional Input Errors

	Absolute errors				Running time in seconds			
benchmark	Rosa	Fluctuat	Fluctuat (subdiv)	FPTaylor	Rosa	Fluctuat	Fluctuat (subdiv)	FPTaylor
with roundoff errors only								
doppler	4.15e-13	3.90e-13	1.54e-13	**1.35e-13**	7.44	1	2	6.03
doppler (r)	2.42e-13	3.90e-13	1.40e-13	**1.35e-13**	6.85	1	2	6.82
jet	5.33e-09	4.08e-08	2.10e-11	**1.17e-11**	94.39	1	2	11.87
jet (r)	4.91e-09	4.08e-08	1.88e-11	**1.17e-11**	76.22	1	2	11.47
rigidBody	3.65e-11	3.65e-11	3.65e-11	**3.61e-11**	0.74	1	2	5.40
rigidBody (r)	3.65e-11	3.65e-11	3.65e-11	**3.61e-11**	0.66	1	2	4.64
sine	5.74e-16	7.97e-16	7.41e-16	**5.52e-16**	1.32	1	1	5.08
sineOrder3	9.96e-16	1.15e-15	1.09e-15	**8.90e-16**	0.24	1	1	3.73
sqroot	**2.87e-13**	3.21e-13	3.21e-13	**2.87e-13**	0.49	1	1	6.78
turbine1	5.99e-14	9.20e-14	2.21e-14	**2.11e-14**	4.93	1	2	7.54
turbine1 (r)	5.15e-14	9.26e-14	2.21e-14	**2.11e-14**	1.19	1	2	7.08
turbine2	7.68e-14	1.29e-13	2.87e-14	**2.62e-14**	1.43	1	2	5.76
turbine2 (r)	6.30e-14	1.34e-13	2.87e-14	**2.62e-14**	0.92	1	2	6.27
turbine3	4.62e-14	6.99e-14	**1.34e-14**	1.55e-14	3.19	1	2	6.51
turbine3 (r)	4.02e-14	7.03e-14	**1.32e-14**	1.55e-14	1.18	1	2	6.74
				total	201	15	27	102
				total (-jet)	31	13	23	78
with input errors								
doppler (re)	1.83e-11	5.45e-11	2.21e-11	**1.82e-11**	12.35	1	2	6.90
jet (re)	3.36e-7	4.67e-4	1.37e-7	**3.85e-8**	75.96	1	2	12.44
turbine1 (re)	**4.61e-10**	1.82e-9	6.02e-10	**4.61e-10**	1.14	1	2	7.98
turbine2 (re)	5.87e-10	2.82e-9	6.14e-10	**5.86e-10**	0.88	1	2	9.12
turbine3 (re)	3.33e-10	1.24e-9	**2.53e-10**	3.32e-10	1.12	1	2	7.47
rigidBody (re)	**1.50e-7**	**1.50e-7**	**1.50e-7**	**1.50e-7**	1.035	1	2	5.48
sine (e)	**1.00e-11**	2.09e-11	1.01e-11	**1.00e-11**	1.285	1	1	5.57
				total	94	7	13	55
				total (-jet)	18	6	11	43
with input constraint								
doppler (r)	1.76e-14	1.09e-13	4.84e-14	**1.57e-14**	6.93	1	2	4.70
doppler (re)	**4.67e-13**	1.37e-11	6.28e-12	4.77e-13	11.10	1	2	10.07
jet (r)	4.91e-9	4.08e-8	1.88e-11	**1.48e-11**	83.55	1	2	1730.16
jet (re)	3.36e-7	4.67e-4	**1.37e-7**	-	80.87	1	2	-
rigidBody (r)	1.66e-11	3.65e-11	3.34e-11	**1.52e-11**	12.53	1	2	60.65
rigidBody (re)	8.84e-8	1.50e-7	1.15e-7	**6.78e-8**	12.52	1	2	283.44
turbine1 (r)	4.26e-14	8.66e-14	**2.21e-14**	2.48e-14	2.42	1	2	5.69
turbine1 (re)	4.61e-10	1.94e-9	6.51e-10	**4.59e-10**	1.75	1	2	108.36
turbine2 (r)	5.26e-14	1.45e-13	**2.44e-14**	2.92e-14	1.53	1	2	5.11
turbine2 (re)	5.87e-10	3.02e-9	6.33e-10	**4.84e-10**	1.79	1	2	36.80
turbine3 (r)	3.55e-14	7.32e-14	**9.50e-15**	1.49e-14	4.02	1	2	10.39
turbine3 (re)	3.33e-10	1.33e-09	**2.30e-10**	2.76e-10	4.25	1	2	315.91
				total	223	12	24	2571
				total (-jet)	59	10	20	841

that imply range bounds for future work. Our approach does not include all loops; in particular, our proposed approach is only applicable to forward computations and not, for instance, iterative algorithms. For these, tracking roundoff errors across loop iterations is not meaningful. Our technique does cover a number of interesting patterns though, including simulations of initial value problems in physics. We note that the alternative for analyzing numerical errors in general nonlinear loops is unrolling, as computations of fixpoints return top, respectively, the error bound infinity, on all but some very specialized loops. Loop unrolling, however, as our experiments show, does not scale well when the computations are nonlinear.

6.1. General Error Propagation

Representing the computation of the loop body by f, recall that we want to compute the overall error after k-fold iteration f^k of f: $\max_{x,\tilde{x}} |f^k(x) - \tilde{f}^k(\tilde{x})|$. f, g, and σ are now vector valued, $f, g, \sigma : \mathbb{R}^n \to \mathbb{R}^n$, because we are nesting the potentially multivariate function f. In essence, we want to compute the effect of iterating Equation (4).

THEOREM. *Let g be such that $|f(x) - f(y)| \leq g(|x - y|)$, it satisfies $g(x + y) \leq g(x) + g(y)$, and is monotonic. Further, σ and λ satisfy $|f(\tilde{x}) - \tilde{f}(\tilde{x})| \leq \sigma(\tilde{x})$ and $|x - \tilde{x}| \leq \lambda$ as before. The absolute value is taken component-wise. Then the numerical error after k iterations is given by*

$$|f^k(x) - \tilde{f}^k(\tilde{x})| \leq g^k(|x - \tilde{x}|) + \sum_{i=0}^{k-1} g^i(\sigma(\tilde{f}^{k-i-1}(\tilde{x}))) \tag{9}$$

Thus, the overall error after k iterations can be decomposed into the initial error propagated through k iterations, and the roundoff error from the ith iteration propagated through the remaining iterations.

PROOF. We show this by induction. The base case $k = 1$ is covered by our treatment of straight-line computations (Section 5.2). By adding and subtracting $f(\tilde{f}^{k-1}(\tilde{x}))$, we get

$$\begin{pmatrix} |f^k(x)_1 - \tilde{f}^k(\tilde{x})_1| \\ \vdots \\ |f^k(x)_n - \tilde{f}^k(\tilde{x})_n| \end{pmatrix} \leq \begin{pmatrix} |f^k(x)_1 - f(\tilde{f}^{k-1}(\tilde{x}))_1| \\ \vdots \\ |f^k(x)_n - f(\tilde{f}^{k-1}(\tilde{x}))_n| \end{pmatrix} + \begin{pmatrix} |f(\tilde{f}^{k-1}(\tilde{x}))_1 - \tilde{f}^k(\tilde{x})_1| \\ \vdots \\ |f(\tilde{f}^{k-1}(\tilde{x}))_n - \tilde{f}^k(\tilde{x})_n| \end{pmatrix}.$$

Applying the definitions of g and σ

$$\leq g \begin{pmatrix} |f^{k-1}(x)_1 - \tilde{f}^{k-1}(\tilde{x})_1| \\ \vdots \\ |f^{k-1}(x)_n - \tilde{f}^{k-1}(\tilde{x})_n| \end{pmatrix} + \sigma(\tilde{f}^{k-1}(\tilde{x})),$$

then using the induction hypothesis and monotonicity of g,

$$\leq g\left(g^{k-1}(\vec{\lambda}) + \sum_{i=0}^{k-2} g^i(\sigma(\tilde{f}^{k-i-1}(\tilde{x})))\right) + \sigma(\tilde{f}^{k-1}(\tilde{x})),$$

and then using $g(x+y) \leq g(x) + g(y)$, we finally have

$$\leq g^k(\vec{\lambda}) + \sum_{i=1}^{k-1} g^i(\sigma(\tilde{f}^{k-i-1}(\tilde{x}))) + \sigma(\tilde{f}^{k-1}(\tilde{x}))$$
$$= g^k(\vec{\lambda}) + \sum_{i=0}^{k-1} g^i(\sigma(\tilde{f}^{k-i-1}(\tilde{x}))). \quad \square$$

6.2. Closed-Form Expression

We instantiate the propagation function g as before using propagation coefficients. These can, however, differ for each loop iteration if the ranges of variables change. Thus, evaluating Equation (9) as given, with a fresh set of propagation coefficients for each iteration i, amounts to loop unrolling but with a loss of correlation between each loop iteration. We observe that when the ranges are bounded (as by our assumption), then we can compute K as a matrix of propagation coefficients and similarly obtain $\sigma(\tilde{f}^i) = \sigma$ as a vector of constants, both *valid for all iterations*. From this, we obtain a closed form for the expression of the error:

$$|f^k(x) - \tilde{f}^k(\tilde{x})| \leq K^k\lambda + \sum_{i=1}^{k-1} K^i\sigma + \sigma = K^k\lambda + \sum_{i=0}^{k-1} K^i\sigma,$$

where λ is the vector of initial errors. Denoting by I the identity matrix, if $(I-K)^{-1}$ exists,

$$|f^k(x) - \tilde{f}^k(\tilde{x})| \leq K^k\lambda + ((I-K)^{-1}(I-K^k))\sigma.$$

We obtain K^k with power-by-squaring and compute the inverse with the Gauss-Jordan method with rational coefficients to obtain sound results (though a closed form is not strictly necessary for our purpose, because we do know the number of iterations k).

Computing K and σ. When the ranges of the variables of the loop are inductive, that is, both the real-valued and the finite-precision values remain within the initial ranges, then these are clearly the ranges for the computation of K and roundoffs σ. For loops, we are faced with a chicken-and-egg problem: For the computation of error bounds, we require the finite-precision ranges, but in order to compute the finite-precision ranges from the real-valued ones, we need to know the error bounds. (For straight-line computations, where we only do a forward computation, we always know the error bounds.) We solve this issue by requiring the programmer to specify the finite-precision ranges with the following syntax: `a <= ~ x && ~ x <= b`, as in Figure 3. We believe that it is reasonable to assume that a user writing these applications has the domain knowledge to be able to provide these specifications.

6.3. Handling Additional Sources of Errors

What if roundoff errors are not the only errors present? If the real-valued computation given by the specification is also the *ideal* computation, then we can simply add the errors in the same way as roundoff errors. If the real-valued computation is, however, already an approximation of some other *unknown* ideal function, say, f_*, then it is not directly clear how our error computation applies.

This may be the case, for example, for truncation errors due to a numerical algorithm. To model such errors, let us suppose that we can compute (or at least overestimate) these by a function $\tau : \mathbb{R}^n \to \mathbb{R}^n$, that is, $\tau_{f_*}(x) = |f_*(x) - f(x)|$.

In the following, we consider the one-dimensional case $n = 1$ for simplicity of exposition, but it generalizes as before to the n-dimensional case. We can apply a similar

Table II. Absolute Errors and Running Times (in Seconds) for Different Benchmarks and Different Number of Loop Iterations

	Absolute errors		Running time	
benchmark	Rosa	Fluctuat	Rosa	Fluctuat
pendulum 50	**2.21e-14**	2.43e-13	8.00	47
pendulum 100	**8.82e-14**	∞	8.00	—
pendulum 250	**2.67e-12**	∞	8.00	—
pendulum 500	**6.54e-10**	∞	8.00	—
pendulum 1000	**3.89e-5**	∞	8.00	—
mean 100	3.21e-7	**9.92e-9**	4.40	1
mean 500	1.62e-6	**1.01e-8**	6.01	5
mean 1000	3.30e-6	**1.01e-8**	6.877	27
mean 2000	4.51e-6	**1.03e-8**	3.82	158
mean 3000	4.96e-6	**1.05e-8**	3.79	392
mean 4000	5.12e-6	**1.06e-8**	4.18	734
nbody 50	**1.30e-11**	∞	793.26	—
nbody 100	**1.35e-8**	∞	775.87	—

separation of errors as before:

$$\begin{aligned}|f_*(x) - \tilde{f}(\tilde{x})| &\leq |f_*(x) - f(x)| + |f(x) - f(\tilde{x})| + |f(\tilde{x}) - \tilde{f}(\tilde{x})| \\ &= \tau(x) + g(|x - \tilde{x}|) + \sigma(\tilde{x}),\end{aligned}$$

which lets us decompose the overall error into the truncation, the propagated initial, and the roundoff error. If we now iterate, then we find by a similar argument as before:

$$\begin{aligned}&|f_*^m(x) - \tilde{f}(\tilde{x})| \\ &\leq g^m(|x - \tilde{x}|) + \sum_{j=0}^{m-1} g^j\big(\tau(f_*^{m-j-1}(x))\big) + g^j\big(\sigma(\tilde{f}^{m-j-1}(\tilde{x}))\big) \\ &= g^m(|x - \tilde{x}|) + \sum_{j=0}^{m-1} g^j\big(\tau(f_*^{m-j-1}(x)) + \sigma(\tilde{f}^{m-j-1}(\tilde{x}))\big).\end{aligned}$$

The result essentially means that our previously defined method can also be applied to the case when truncation (or similar) errors are present. We do not pursue this direction further, however, and leave a proper automated treatment of truncation errors to future work.

6.4. Experimental Results

We evaluate our technique on three benchmarks in Table II: pendulum, mean, and nbody. We already presented the pendulum benchmark in Figure 3. The mean benchmark computes a running average of values in a range of [-1200, 1200]. The nbody benchmark is a two-body simulation of Jupiter orbiting around the Sun. For each benchmark, we consider different number of iterations of the loop and report the error for one of the loop's variables. Fluctuat computes a trivial fixpoint for these benchmarks, as the errors grow with each iteration, so we manually set the number of times the loop is unrolled. For the pendulum 50 benchmark, Rosa is able to compute a tighter error bound with a faster runtime. For larger numbers of iterations, Fluctuat reports an absolute error of ∞. This is also the result for the nbody benchmark. For the mean benchmark, where the computation is less complex, Fluctuat can compute tighter error bounds, at the expense of much longer analysis times. This illustrates that our technique outperforms unrolling in Fluctuat for benchmarks that are highly nonlinear,

whereas Fluctuat's strategy may be used for cases where the nonlinearity is limited as is the number of iterations. Note that Rosa's runtime is largely independent of the loop's number of iterations, as the bulk of the computation is performed once for the loop body only.

7. DISCONTINUITIES

Recall the piecewise jet engine approximation from Figure 4. Due to the initial errors on `x` and `y`, the real-valued computation may take a different branch than the finite-precision one, and thus produce a different result. We call this difference the *discontinuity error*.

In the following, we will assume that individual branch conditions are of the form `e1` $\circ$ `e2`, where $\circ \in \{<, \leq, >, \geq\}$ and `e1`, `e2`, are arithmetic expressions. More complex conditions can be expressed by nesting conditionals. We do not assume the function represented by the conditional to be neither smooth nor continuous. We perform our analysis pairwise for each pair of paths in the program. While this gives, in the worst case, an exponential number of cases to consider, we found that many of these paths are infeasible due to inconsistent branch conditions; such infeasible paths are eliminated early using straightforward calls to the SMT solver.

7.1. Applying Separation of Errors

Using our previous notation, let us consider a function with a single branch statement:

```
if (c) f1
else f2
```

and let f_1 and f_2 be the real-valued functions corresponding to the `if` and the `else` branch, respectively. Then, the discontinuity error is given by $|f_1(x) - \tilde{f}_2(\tilde{x})|$, that is, the real-valued computation takes branch f_1 and the finite-precision one f_2. The opposite case is analogous. We again apply the idea of separation of errors:

$$|f_1(x) - \tilde{f}_2(\tilde{x})| \leq |f_1(x) - f_1(\tilde{x})| + |f_1(\tilde{x}) - f_2(\tilde{x})| + |f_2(\tilde{x}) - \tilde{f}_2(\tilde{x})|. \tag{10}$$

The individual components are

(1) $|f_1(x) - f_1(\tilde{x})|$: the difference in f_1 due to initial errors. We can compute this difference with our propagation coefficients: $|f_1(x) - f_1(\tilde{x})| \leq K|x - \tilde{x}|$.
(2) $|f_1(\tilde{x}) - f_2(\tilde{x})|$: the real-valued difference between f_1 and f_2. We can bound this value by the Z3-aided range computation from Section 4.
(3) $|f_2(\tilde{x}) - \tilde{f}_2(\tilde{x})|$: the roundoff error when evaluating f_2 in finite-precision arithmetic. We use the procedure from Section 5.1 as before.

x and $\tilde{x}$ in $|f_1(x) - \tilde{f}_2(\tilde{x})|$ are correlated by the initial error $|x - \tilde{x}| \leq \lambda$. Constructing only one constraint capturing the discontinuity error becomes very difficult for the underlying SMT solver and has caused our previous method for computing discontinuity errors to scale only to unary functions [Darulova and Kuncak 2014]. With the separation above, the individual parts are easier to handle for the solver, since we reduce the number of variables and correlations in each of the three parts. On the other hand, it clearly introduces an additional over-approximation, but we observed in our experiments that this is in general small. In contrast, Fluctuat's approach relies on constraints on the affine forms to capture the different branch conditions [Goubault and Putot 2013].

7.2. Determining Ranges for x and $\tilde{x}$

As in the previous sections, it is crucial to determine the ranges of $x, \tilde{x} \in \mathbb{R}$ over which to evaluate the individual parts of Equation (10). A sound approach would be to simply use

Table III. Absolute Discontinuity Errors Computed and Running Times (in Seconds) of Rosa and Fluctuat

	Absolute errors		Running time	
benchmark	Rosa	Fluctuat	Rosa	Fluctuat
cubicSpline	**1.25e-15**	12.00	8.41	1
jetApprox	**0.0232**	18.40	45.53	1
jetApprox (err)	**0.0242**	19.06	43.06	1
jetApproxBadFit	**0.8825**	9.305	15.18	1
jetApproxBadFit (err)	**0.8852**	10.09	10.17	1
jetApproxGoodFit	**0.0428**	5.191	4.89	1
jetApproxGoodFit (err)	**0.0450**	5.193	4.23	1
linearFit	**0.6374**	1.721	2.53	1
quadraticFit	**0.2548**	10.60	20.56	1
quadraticFit (err)	**0.2551**	10.96	19.48	1
quadraticFit2	**3.14e-9**	0.6321	4.04	1
quadraticFit2 (err)	**0.0009**	0.7188	3.77	1
simpleInterpolator	3.40e-5	**1.0e-5**	0.61	1
sortOfStyblinski	**1.0878**	27.07	4.78	1
sortOfStyblinski (err)	**1.0982**	28.82	4.22	1
squareRoot	**0.0238**	0.0394	2.20	1
squareRoot3	**2.76e-9**	0.4289	5.62	1
squareRoot3Invalid	**3.93e-9**	0.4288	5.47	1
styblinski	**4.81e-8**	121.16	29.16	1
styblinski (err)	**0.0132**	124.10	24.56	1

the input ranges, but this would lead to unnecessary over-approximations. In general, not all inputs can exhibit a divergence between the real-valued and the finite-precision computation; those that can can be determined by the branch conditions and the errors on evaluating these. Consider the branch condition `if (e1 < e2)` and, as before, the case where the real-valued path takes the if-branch, that is, variable x satisfies $e_1 < e_2$ and $\tilde{x}$ satisfies $\tilde{e_1} \geq \tilde{e_2}$. The constraint for the finite-precision variables $\tilde{x}$ is then

$$e_1 + \delta_1 < e_2 + \delta_2 \wedge e_1 \geq e_2 \wedge |x - \tilde{x}| \leq \lambda,$$

where δ_1, δ_2 are error intervals on evaluating $e1$ and $e2$ in finite-precision, respectively. This constraint expresses that we want to bound Equation (10) over those values that satisfy the condition $e1 \geq e2$ but are "close enough" such that their corresponding ideal real value could take the other path. We create such a constraint both for the variables representing finite-precision values ($\tilde{x}$), as well as the real-valued ones x and use them as additional constraints when computing the individual parts of Equation (10). The procedure for other branch conditions is analogous.

7.3. Experimental Results

We evaluate our technique on a number of benchmarks with discontinuities, which we have either constructed by piecewise approximating a more complex function or chosen from Goubault and Putot [2013]. All the benchmarks' source code is available online. We compare our results in terms of accuracy and performance against Fluctuat. Fluctuat does not check for discontinuity errors by default; we enable this analysis with the "Unstable test analysis"' option (this is the only way). Subdivisions, however, do not appear to work with this setting. Table III summarizes our results. While Fluctuat is faster than Rosa, Rosa is able to compute significantly tighter error bounds and, we believe, achieves a good compromise between accuracy and performance.

8. RELATED WORK

To the best of our knowledge, Fluctuat [Goubault and Putot 2011, 2013], FPTaylor [Solovyev et al. 2015], and Gappa [Boldo and Marché 2011; Linderman et al. 2010] are most related to our work (see our comparison in Section 5.7). We are not aware of other tools or techniques that can *soundly* and *automatically* quantify numerical errors in the presence of nonlinearity, branches, and loops.

In the context of abstract interpretation, domains exist that are sound with respect to floating-points and that can be used to prove the absence of undesirable runtime issues such as division by zero [Blanchet et al. 2003; Miné 2004; Feret 2004; Chen et al. 2008; Ghorbal et al. 2009]. Feret [2005] presents an abstract domain that associates the ranges with the iteration count, similar to our proposed technique for loops. Martel [2002] considers the stability of loops by proving whether loops can asymptotically diverge. The problem that we are solving differs, however, as we want to quantify the *difference* between the real-valued and the finite-precision computation.

Floating-point arithmetic has been formalized in the SMT-LIB format [Brain et al. 2015], and approaches exist that deal with the prohibiting complexity of bit-precise techniques via approximations [Brillout et al. 2009; Haller et al. 2012]. For encoding roundoff errors, a combination of the theory of real and floating-point arithmetic is needed; we are not aware of such an approach that is able to quantify the deviation of finite-precision computations with respect to reals. Floating-point precision assertions can also be proven using an interactive theorem prover [Boldo and Marché 2011; Linderman et al. 2010; Ayad and Marché 2010; Harrison 2006]. These tools can reason about ranges and errors of finite-precision implementations but target specialized and precise properties, which, in general, require an expert user and interactively guiding the proof. Very tight error bounds have been shown by manual proof for certain special computations, such as powers [Graillat et al. 2014]. Our work chooses a different tradeoff among accuracy, automation, and generality.

Floating-point accuracy has naturally also been a concern in numerical analysis. The work in this area has focused mostly on designing accurate algorithms, and example collections can be found in Higham [2002]. Demmel et al. [2008] provide a survey of results on the existence of algorithms for evaluating polynomials that have a relative error less than 1. Implementations of polynomials have also been explored in the context of hardware synthesis for different error targets [Drane 2013]. These goals differ considerably from Rosa's, which determines an error bound for an *arbitrary* arithmetic expression but does not try to derive a new *algorithm* for meeting an error target. In contrast to our technique, which is fully automated and whose error computation is applicable to *any* arithmetic expression, these analyses are specific to each algorithm and performed manually.

Synthesis of fixed-point arithmetic programs has also been an area of active research, with different utilized techniques: simulation or testing [Mallik et al. 2007; Jha and Seshia 2013], interval or affine arithmetic [Lee et al. 2006], or automatic differentiation [Gaffar et al. 2004]. Some approaches try to optimize the bit-width, whereas in our case we keep it fixed but provide a sound and accurate analysis, which could be used in combination with an optimization technique, like, for example, Jha and Seshia [2013]. A similar approach to our range estimation has been developed independently by Kinsman and Nicolici [2009] in the context of fixed-point arithmetic. We also identify the potential of additional constraints and develop optimizations to make the use of an SMT solver efficient enough. Further, our techniques aim to be generally applicable to various finite-precision arithmetics.

Several approaches also exist to test the stability of numerical programs, for example, by perturbation of low-order bits and rewriting [Tang et al. 2010] or by perturbing the

rounding modes [Scott et al. 2007]. Another common theme is to run a higher-precision program alongside the original one. Benz et al. [2012] does so by instrumentation, Paganelli and Ahrendt [2013] generates constraints that are then discharged with a floating-point arithmetic solver, and Chiang et al. [2014] developed a guided search to find inputs that maximize errors. Lam et al. [2013a] uses instrumentation to detect cancellation and thus loss of precision. Ivancic et al. [2010] combines abstract interpretation with model checking to check the stability of programs, tracking one input at a time. Majumdar et al. [2010] uses concolic execution to find two sets of inputs that maximize the difference in the outputs. These approach are based on testing, however, and cannot prove sound bounds. Testing has also been used as a verification method for optimizing mixed-precision computations [Rubio-González et al. 2013; Lam et al. 2013b].

It is natural to use the Jacobian for sensitivity analysis. Related to our work is a proof framework using this idea for showing programs robust in the sense of k-Lipschitz continuity [Chaudhuri et al. 2011]. Note, however, that our approach does not require programs to be continuous. Gazeau et al. [2012] relaxes the strict definition of robustness to programs with specified uncertainties and presents a framework for proving while-loops with a particular structure robust. Our work follows the philosophy of these approaches in leveraging Jacobians of program paths, yet we explicitly incorporate the handling of roundoff errors in a fully automated system.

9. CONCLUSION

We believe that numerical errors, such as roundoff errors, should not be an afterthought and that programming language support is needed and can be provided to help scientists write numerical code that does what it is expected to do. To this end, we presented, on one hand, a real-valued specification language with explicit error annotations from which our tool Rosa synthesizes finite-precision code that fulfills the given specification. On the other hand, we presented a set of techniques based on unified principles that provides automated, efficient, static error analysis that is crucial towards making such a compiler practical. We have extensively evaluated these techniques against state-of-the-art tools and we believe they represent an interesting compromise between accuracy and efficiency.

REFERENCES

Adolfo Anta, Rupak Majumdar, Indranil Saha, and Paulo Tabuada. 2010. Automatic verification of control system implementations. In *EMSOFT*.

Ali Ayad and Claude Marché. 2010. Multi-prover verification of floating-point programs. In *IJCAR*.

David H. Bailey, Yozo Hida, Xiaoye S. Li, and Brandon Thompson. 2013. C++/Fortran-90 double-double and quad-double package. Retrieved from http://crd-legacy.lbl.gov/~dhbailey/mpdist/.

Florian Benz, Andreas Hildebrandt, and Sebastian Hack. 2012. A dynamic program analysis to find floating-point accuracy problems. In *PLDI*.

Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2003. A static analyzer for large safety-critical software. In *PLDI*.

Sylvie Boldo and Claude Marché. 2011. Formal verification of numerical programs: From C annotated programs to mechanical proofs. *Mathematics in Computer Science* 5, 4 (2011), 377–393.

Martin Brain, Cesare Tinelli, Philipp Rümmer, and Thomas Wahl. 2015. An automatable formal semantics for IEEE-754 floating-point arithmetic. In *ARITH*.

Angelo Brillout, Daniel Kroening, and Thomas Wahl. 2009. Mixed abstractions for floating-point arithmetic. In *FMCAD*.

CEA-LIST. 2015. Frama-C Software Analyzers. (2015). Retrieved from http://frama-c.com/index.html.

Swarat Chaudhuri, Sumit Gulwani, Roberto Lublinerman, and Sara Navidpour. 2011. Proving programs robust. In *ESEC/FSE*.

Liqian Chen, Antoine Miné, and Patrick Cousot. 2008. A sound floating-point polyhedra abstract domain. In *APLAS*.

Wei-Fan Chiang, Ganesh Gopalakrishnan, Zvonimir Rakamaric, and Alexey Solovyev. 2014. Efficient search for inputs causing high floating-point errors. In *PPoPP*.

Eva Darulova and Viktor Kuncak. 2011. Trustworthy numerical computation in scala. In *OOPSLA*.

Eva Darulova and Viktor Kuncak. 2014. Sound compilation of reals. In *POPL*.

Luiz H. de Figueiredo and Jorge Stolfi. 2004. Affine arithmetic: Concepts and applications. *Numer. Algor.* 37, 1–4 (2004).

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *TACAS*.

James Demmel, Ioana Dumitriu, Olga Holtz, and Plamen Koev. 2008. Accurate and efficient expression evaluation and linear algebra. *Acta Numer.* 17 (2008).

Theo Drane. 2013. *Lossy Polynomial Datapath Synthesis*. Ph.D. Dissertation. Imperial College London.

Jérôme Feret. 2004. Static analysis of digital filters. In *ESOP*.

Jérôme Feret. 2005. The arithmetic-geometric progression abstract domain. In *VMCAI*.

Altaf Abdul Gaffar, Oskar Mencer, Wayne Luk, and Peter Y. K. Cheung. 2004. Unifying bit-width optimisation for fixed-point and floating-point designs. In *FCCM*.

Ivan Gazeau, Dale Miller, and Catuscia Palamidessi. 2012. A non-local method for robustness analysis of floating point programs. In *QAPL*.

Khalil Ghorbal, Eric Goubault, and Sylvie Putot. 2009. The zonotope abstract domain taylor1+. In *CAV*.

Khalil Ghorbal, Eric Goubault, and Sylvie Putot. 2010. A logical product approach to zonotope intersection. In *CAV*.

David Goldberg. 1991. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.* 23, 1 (1991).

Eric Goubault and Sylvie Putot. 2011. Static analysis of finite precision computations. In *VMCAI*.

Eric Goubault and Sylvie Putot. 2013. Robustness analysis of finite precision implementations. In *APLAS*.

Stef Graillat, Vincent Lefèvre, and Jean-Michel Muller. 2014. *On the Maximum Relative Error When Computing x^n in Floating-point Arithmetic*. Technical Report <ensl-00945033v2>. Laboratoire d'Informatique de Paris 6, Inria Grenoble Rhône-Alpes.

Leopold Haller, Alberto Griggio, Martin Brain, and Daniel Kroening. 2012. Deciding floating-point logic with systematic abstraction. In *FMCAD*.

John Harrison. 2006. Floating-point verification using theorem proving. In *Formal Methods for Hardware Verification: 6th International School on Formal Methods for the Design of Computer, Communication, and Software Systems*.

Nicholas J. Higham. 2002. *Accuracy and Stability of Numerical Algorithms*. Siam.

Computer Society IEEE. 2008. IEEE standard for floating-point arithmetic. IEEE Std 754-2008 (2008).

ISO/IEC. 2008. *Programming Languages—C—Extensions to Support Embedded Processors*. Technical Report ISO/IEC TR 18037.

Franjo Ivancic, Malay K. Ganai, Sriram Sankaranarayanan, and Aarti Gupta. 2010. Numerical stability analysis of floating-point computations using software model checking. In *MEMOCODE*.

Susmit Jha and Sanjit A. Seshia. 2013. Synthesis of optimal fixed-point implementations of numerical software routines. In *Proceedings of the 6th International Workshop on Numerical Software Verification (NSV)*.

Dejan Jovanović and Leonardo de Moura. 2012. Solving non-linear arithmetic. In *IJCAR*.

Adam B. Kinsman and Nicola Nicolici. 2009. Finite precision bit-width allocation using sat-modulo theory. In *DATE*.

Michael O. Lam, Jeffrey K. Hollingsworth, Bronis R. de Supinski, and Matthew P. Legendre. 2013b. Automatically adapting programs for mixed-precision floating-point computation. In *ICS*.

Michael O. Lam, Jeffrey K. Hollingsworth, and G. W. Stewart. 2013a. Dynamic floating-point cancellation detection. *Parallel Comput.* 39, 3 (2013).

Dong-U. Lee, Altaf Abdul Gaffar, Ray C. C. Cheung, Oskar Mencer, Wayne Luk, and George A. Constantinides. 2006. Accuracy-guaranteed bit-width optimization. *IEEE Trans. CAD Integr. Circ. Syst.* 25, 10 (2006).

Michael D. Linderman, Matthew Ho, David L. Dill, Teresa H. Meng, and Garry P. Nolan. 2010. Towards program optimization through automated analysis of numerical precision. In *CGO*.

Rupak Majumdar, Indranil Saha, and Zilong Wang. 2010. Systematic testing for control applications. In *MEMOCODE*.

Arindam Mallik, Debjit Sinha, Prithviraj Banerjee, and Hai Zhou. 2007. Low-power optimization by smart bit-width allocation in a systemc-based asic design environment. *IEEE Trans. CAD Integr. Circ. Syst.* 26, 3 (2007).

Matthieu Martel. 2002. Static analysis of the numerical stability of loops. In *SAS*.

Antoine Miné. 2004. Relational abstract domains for the detection of floating-point run-time errors. In *ESOP*.

Ramon E. Moore. 1966. *Interval Analysis*. Prentice-Hall.

Gabriele Paganelli and Wolfgang Ahrendt. 2013. Verifying (in-)stability in floating-point programs by increasing precision, using SMT solving. In *SYNASC*.

Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H. Bailey, Costin Iancu, and David Hough. 2013. Precimonious: Tuning assistant for floating-point precision. In *SC*.

N. Stan Scott, Fabienne Jézéquel, Christophe Denis, and Jean Marie Chesneaux. 2007. Numerical 'health check' for scientific codes: The CADNA approach. *Computer Physics Communications* (2007).

Alexey Solovyev, Charles Jacobsen, Zvonimir Rakamaric, and Ganesh Gopalakrishnan. 2015. Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. In *FM*.

Enyi Tang, Earl Barr, Xuandong Li, and Zhendong Su. 2010. Perturbing numerical calculations for statistical analysis of floating-point program (in)stability. In *ISSTA*.

Chris Woodford and Chris Phillips. 2012. *Numerical Methods with Worked Examples*. Vol. 2nd. Springer.

Randy Yates. 2013. *Fixed-Point Arithmetic: An Introduction*. Technical Report. Digital Signal Labs. Retrieved from http://www.digitalsignallabs.com/fp.pdf.

Received August 2015; revised April 2016; accepted November 2016