Non-local robustness analysis via rewriting techniques [*]

Ivan Gazeau, Dale Miller, and Catuscia Palamidessi

Robustness is a correctness property which intuitively means that if the inputs to a program changes less than a fixed small amount then its output changes only slightly. The study of errors caused by finite-precision semantics requires a stronger property: the results in the finite-precision semantics have to be close to the result in the exact semantics. Compositional methods often are not useful in determining which programs are robust since key constructs—like the conditional and the while-loop—are not continuous. We propose a method for proving a while-loop always returns finite precision values close to the exact values. Our method uses some rewriting theory tools to analyze the possible paths in a program's execution in order to show that while local operations in a program might not be robust, the full program might be guaranteed to be robust. This method is non-local in the sense that instead of breaking the analysis down to single lines of code, it checks certain global properties of its structure. We show the applicability of our method on two standard algorithms: the CORDIC computation of the cosine and Dijkstra's shortest path algorithm.

Keywords: Program analysis, floating-point arithmetic, robustness to errors.

1 Introduction

Floating point arithmetic is central to many critical applications. The development of methods for establishing the correctness of such programs is, of course, as important as is it difficult. One problem related to reasoning about floating point arithmetic is that it is quite different from real number arithmetic: for example, addition is neither commutative nor associative [5]. Another problem in dealing with floating point programs is the propagation of errors due to the digitization of analog quantities and the introduction of floating point errors during computation.

The programmers and certifiers of floating point programs would like to think about a program's meaning with respect to real number semantics and not with respect to the more ad hoc and complicated semantics given by some specific definition of floating point arithmetic, such as the IEEE standard 754 [8]. An important difficulty in reasoning about floating point programs is that in dealing with non-continuous operators such as the conditional and the while-loop, floating point errors can result in erratic behavior. The problem is that these constructs are *non-robust*: small variations in the data can cause large variations in the results.

When the program contains non robust operators, traditional compositional methods do not work well. Decomposing the correctness of a looping program using Hoare triples, for example, usually requires introducing abstractions (such as approximations) which can become too imprecise to be useful. Other approaches often involve very complex and intricate proof.

In this paper, we will take a different approach and move away from program analysis using Hoare's style emphasis on local and compositional analysis of a looping program. Instead, we shall simply try to model the entire space of possible "erratic" program behaviors as a kind of *rewriting relation*. In the literature dealing with rewriting systems, there are a number of techniques for inferring global properties (*e.g.*, confluence, determinacy) even when the basic relation does not satisfy the local version of these

[*]This work has been partially supported by the project ANR-09-BLAN-0345-02 CPP.

properties. In particular, we show how it is possible to show that a looping program might well be robust even if the individual unfoldings of a while loop could lead to different values. A rather direct analogy from rewriting for, say the λ-calculus, could be: while it is possible for a given λ-term to be (locally) reduced in a number of different ways to different λ-terms, it is still possible to show that globally, there is at most one normal form for λ-terms.

An excellent and simple example of this style of reasoning is illustrated by Dijkstra's minimal path algorithm [3]. This greedy algorithm moves from a source node to its neighbors, always picking the node with the least accumulated path from the source. If one makes small changes to the distances labeling edges, then the least path distance will change also by a small amount. However, the actual behavior of the loop and the marking of subsequent nodes can vary greatly with small changes to edge lengths. By modelling all possible behaviors of this looping program under the influence of fixed precision arithmetic and by imposing certain conditions on the algorithm's programming language expression, we shall be able to show that Dijkstra's algorithm is, in fact, robust.

This paper builds on and extends a our previous workshop paper [4]: the current paper provides for a completely different set of conditions on looping programs in order to show their robustness.

Plan of the paper In the next section we introduce the concept of robustness and we relate it to the notions of continuity and *k*-Lipschitz. Section 3 contains our main contribution: a schema for reasoning about robustness in programs and its correctness. We then show the applicability of our proposal in two main examples: The CORDIC algorithm for computing cosine, presented in Section 4, and the Dijkstra's shortest-path algorithm, presented in Section 5. In Section 6 we discuss some related work. Section 7 concludes and discusses some future lines of research.

2 Robustness of floating-point programs with respect to the exact semantics

Robustness is a standard concept from control theory [12, 11]. In the case of programming languages, there are two definitions of robustness that have been considered. One definition used by Chaudhuri et al [1] considered robustness to be based on continuity. Later Chaudhuri et al [2] considered a stronger notion of robustness, namely the *k*-Lipschitz property: that is, for every change to the input to a program the output changes proportionally. Another approach was used by Majumdar et al in [9, 10] where robustness is formulated as "if the input of the program changes by an amount less than ε, where ε is a *fixed* constant, then the output changes only slightly." In our paper, we propose a more flexible and general notion of robustness that generalizes both of these concepts. We now motivate and explain our notion of robustness in more detail.

The notions of robustness considered in [1, 2] are mainly useful for *exact semantics*, namely when we do not take into account the errors introduced by the representation and/or the computation. In this case, the only deviation comes from the error of the input. The continuity property, that for a function f on reals is defined as:

$$\forall \varepsilon > 0\, \exists \delta\, \forall i, i' \in \mathbb{R}\ \ |i - i'| < \delta \Rightarrow |f(i) - f(i')| < \varepsilon$$

ensures that the correct output can be approximated when we can approximate the input closely enough. This notion of robustness, however, is too weak in many settings, because a small variation in the input can cause an unbounded change in the output. The *k*-Lipschitz property, defined as

$$\forall i, i' \in \mathbb{R}\ \ |f(i) - f(i')| \leq k|i - i'|$$

amends this problem because it bounds the variation in the output linearly by the variation in the input.

In our setting, however, the k-Lipschitz property is too strong even for the exact semantics. Indeed, there are algorithms that have a desired precision e as a parameter and are considered correct as long as the result differs by at most e from the results of the mathematical function they are meant to implement. A program of this kind may be discontinuous (and therefore not k-Lipschitz) even if it is considered to be a correct implementation of a k-Lipschitz function. The phenomenon is illustrated by the following example

Example 2.1. *The program f illustrated below is meant to compute the inverse of a strictly increasing function $g : \mathbb{R}^+ \to \mathbb{R}^+$ whose inverse is k-Lipschitz for some k.*

```
f(i){ y=0;
      while(g(y) < i){
            y = y+e; }
      return y; }
```

The program f approximates g^{-1} with precision e in the sense that

$$\forall x \in \mathbb{R}^+,\ f(x) - e \leq g^{-1}(x) \leq f(x) \tag{1}$$

Given the inequality (1)*, we would like to consider the program f as robust, even though the function it computes is discontinuous (and hence not k-Lipschitz, for any k).*

This observation leads us to define another property, to capture robustness. Intuitively, this property means that the function is "close enough" to be a k-Lipschitz function:

$$\exists g \in F_k, \forall i \in \mathbb{R}, |f(i) - g(i)| \leq \frac{\varepsilon}{2}$$

where F_k is the set of k-Lipschitz functions. By expanding the definition of k-Lipschitz, we have the following equivalent definition, which we denote as $P^1_{k,\varepsilon}$.

$$\forall i, i' \in \mathbb{R}, |f(i) - f(i')| \leq k|i - i'| + \varepsilon$$

This property extends the k-Lipschitz property, which can be expressed as $P^1_{k,0}$. The function f in the above example is not k-Lipschitz, but it is $P^1_{k,\varepsilon}$. The ε in this case is the value of `e` in the program. .

Now, we want to extend this definition to allow for several variables and for other metric spaces besides $\mathbb{R}$: e.g., probability distributions, intervals arithmetic etc. Thus, we consider, instead, two metric spaces: one for input (I, d_I) and the other for the return value (R,d_R). Hence, our robustness property $P_{k,\varepsilon}$ is defined as it follows:

Definition 2.1. *Let I and R be metric spaces with distance d_I and d_R respectively, let $f : I \to R$ be a function, and let $k, \varepsilon \in \mathbb{R}^+$. We say that f is $P_{k,\varepsilon}$ if*

$$\forall i, i' \in I, d_R(f(i), f(i')) \leq k d_I(i, i') + \varepsilon$$

Our main concern in this paper is to check if a program in the finite precision semantics returns a result close to the result of the same program in an exact semantics. If we note by f the function computed in the exact semantics and by f' the one computed in a finite-precision arithmetic, a reasonable definition of robustness would be:

$$\forall x \in \mathbb{M}, d(f(x) - f'(x)) \leq \varepsilon$$

```
foo(m){
      n initialized;
      while(! S(m)){
        (i,n) = C(m,n);
            m = R(i,m);
      }
      return m; }
```

Figure 1: The main template in imperative style

However, this definition is not compositional. Consider the following functions f and g on $\mathbb{R}$:

$$f(x) = \left(\begin{array}{ll} 0 & x < 0 \\ 1 & x \geq 0 \end{array} \right. \qquad\qquad g(x) = x$$

Depending on the programs that compute f and g, we can have $f = f'$ while $g \neq g$, and in particular we could have that $g(0) = 0$ while $g'(0) = -\varepsilon$. In this scenario we have $f(g(0)) = 1$ while $f'(g'(0)) = 0$. For this reason we define a stronger property that we call closeness property:

Definition 2.2 (Closeness property)**.** *Let I and R be metric spaces with distance d_I and d_R respectively. Let f and f' be two functions from I to R and let $k, \varepsilon \in \mathbb{R}^+$. We say that f' is close to f if the following holds:*

$$\forall x, y \in I, d_R(f(x), f'(y)) \leq k d_I(x,y) + \varepsilon$$

Note that the closeness property implies that the function in the exact semantics is P_{k,ε_0} for some ε_0.

The following section provides a framework to prove the finite-precision semantics of some implementation is close to the exact semantics.

3 A framework for robustness based on rewriting

In this section, we consider a robustness analysis for while loops that satisfy the schema described in Figure 1.

For any program $P(x)$, we denote by $[\![\texttt{P(x)}]\!]_e$ its exact semantics and by $[\![\texttt{P(x)}]\!]_f$ its finite-precision semantics. We define $\mathbb{M}$ to be the domain of the input in the exact semantics, and we assume the domain of the input in the finite-precision semantics to be a subset of $\mathbb{M}$. For instance $\mathbb{M}$ could be $\mathbb{R}$ or $\mathbb{R}^n$, and the corresponding subset in the finite-precision semantics would be the subset of $\mathbb{R}$ (reps. $\mathbb{R}^n$) which can be represented in the machine. Our program `foo` has to be of the type $[\![\text{foo}(x)]\!]_e : \mathbb{M} \to \mathbb{M}$. In practice this is always possible: if the input domain *In* and the output domain *Out* are different, then we can just define $\mathbb{M} = \mathit{In} \times \mathit{Out}$. We assume that $\mathbb{M}$ is a metric space, and we will denote as d its distance. This is the metric with respect to which our definition of robustness is formulated. For instance, if $\mathbb{M} = \mathbb{R}^n$, we could choose the L_1 distance on $\mathbb{M}$ defines as: $d(x,y) = \sum_{i=1}^{n} |x_i - y_i|$.

We now explain the syntax and the structure of our template.

`S(m)` is the stopping condition (of type boolean), which we assume to depend only on the value of `m`.

The body of the loop is decomposed into two parts of code: `C` and `R`. The idea is that `C` can have erratic behavior which makes unfeasible any compositional analysis. `R`, on the other hand, should have specific properties (detailed in the Section 3.2) which makes the program as a whole to behave somewhat regularly, despite the "bad behavior" of `C`.

The syntax `(i,n) = C(m,n)` represents a program fragment that can access the groups of variables `m` and `n`. We assume that the intersection between `m` and `n` is empty. This implies, in particular, that the variables in `n` are not accessible to `R`. Moreover `C` can change the value of `n` and return a value `i` which is read by `R`.

The syntax `m = R(i,m)` represents a program fragment that can access the groups of variables `i` and `m` and just changes the value of `m`.

In addition to these syntactic restriction for `C` and `R`, we add some conditions on `R`. To express these conditions, we first introduce some notation. We are going to interpret the code `R(i,m)` in the exact semantics $[\![\cdot]\!]_e$, and the finite-precision semantics $[\![\cdot]\!]_f$. In each case, the semantics of `R(i,m)` is a function of one argument (possibly a tuple) corresponding to `m`, while we regard `i` as a parameter. We use the notation f_i to represent the function $f_i(m) = [\![\texttt{R(i,m)}]\!]_e$, with $i \in I$, where I is the set of the possible values that can be assigned to `i` by the program. Similarly, we define $f'_i(m) = [\![\texttt{R(i,m)}]\!]_f$. We denote by $\mathscr{R}$ and $\mathscr{R}'$ the sets of these functions, namely $\mathscr{R} = \{f_i \mid i \in I\}$, and similarly $\mathscr{R}' = \{f'_i \mid i \in I\}$. Finally, the closure of $\mathscr{R}$ (resp. $\mathscr{R}'$) under function composition is denoted by $\mathscr{R}^*$ (resp. $\mathscr{R}'^*$).

3.1 The basic rewriting framework

We now define our rewriting framework. The domain $\mathbb{M}$ and the set of functions $\mathscr{R}$ are those given in the previous section. For $a,b \in \mathbb{M}$, define

- $a \rightarrow b$ if there exists $g \in \mathscr{R}$ such that $b = g(a)$
- $a \rightarrow^* b$ f if there exists $g \in \mathscr{R}^*$ such that $b = g(a)$.

In the analysis that follows, we will assume the existence of a function $h : \mathbb{M} \rightarrow \mathbb{R}$ that will be used to "measure" the elements of $\mathbb{M}$. Given such function h, we define the following relations, where $<$ is the standard ordering on $\mathbb{R}$.

- $a \overset{>}{\rightarrow} b$ if there exists $g \in \mathscr{R}$ and $b = g(a)$ such that $h(b) < h(a)$.
- $a \overset{>}{\rightarrow}{}^* b$ is the transitive closure of $\overset{>}{\rightarrow}$, which can be defined by induction as follows: $a = b$ or there exists $c \in \mathbb{M}$ such that $a \overset{>}{\rightarrow}{}^* c$ and $c \overset{>}{\rightarrow} b$.
- $\bar{\mathbb{M}}$ is the set of normal forms in $\mathbb{M}$ with respect to the relation $\overset{>}{\rightarrow}$, i.e. $\bar{\mathbb{M}} = \{m \in \mathbb{M} \mid \nexists m' \in \mathbb{M}.\ m \overset{>}{\rightarrow} m'\}$.

3.2 A sufficient condition for robustness

We now list five conditions. The idea is that if these conditions hold for a particular function that fits the schema in Figure 1, then we are able to justify the function as being robust.

The first two conditions are concerned with the rewriting relations. Condition 1 strengthens the notion of local confluence: not only we require that two values that derive from the same element converge again to the same element, but also that they converge while decreasing the measure h.

Condition 1 (C1)**.** *This condition corresponds to requiring that the rewriting system satisfies the following:*

$$\forall a,b,c \in \mathbb{M}, b \leftarrow a \rightarrow c \implies \exists d \in \mathbb{M}.\ b \overset{>}{\rightarrow}{}^* d \ {}^*\overset{<}{\leftarrow} c \wedge a \overset{>}{\rightarrow}{}^* d$$

Next condition requires the idealized system to be noetherian.

Condition 2 (C2)**.** *The rewriting system $\overset{>}{\to}$ is terminating: that is, there exists no infinite path $a_1 \overset{>}{\to} a_2 \overset{>}{\to} \dots$.*

Remark 1. *If Conditions 1 and 2 hold, then for every $a \in \mathbb{M}$ there is a unique value $b \in \bar{\mathbb{M}}$ such that $a \overset{>}{\to}{}^* b$. This function, which maps a to b, is called* normalizing function *given by $\overset{>}{\to}{}^*$.*

Next, we require the regularity of the normalizing function.

Condition 3 (C3)**.** *The normalizing function $\overset{>}{\to}{}^*$ is P_{k_e,ε_e}*

In addition of these constraints on the real number semantics, there are conditions to grant that the finite-precision semantics is not too far away from the exact semantics. The first one is about the closeness between the idealized semantics and the finite-precision semantics.

Condition 4 (C4)**.** *We consider the subset $\mathscr{R}_r'^*$ of $\mathscr{R}'^*$ of all the possible paths that the algorithm can take. For all $p' \in \mathscr{R}_r'^*$, $p' = f'_{p'_1} o f'_{p'_2} o \dots$ (*i.e. *some finite composition) so we can define a corresponding function $p \in \mathscr{R}^*$: $p = f_{p'_1} o f_{p'_2} o \dots$ (for the same corresponding finite composition). We need the closeness property (for some chosen parameters k_f and ε_f) between p and p':*

$$\forall x, y \in \mathbb{M}, d(p'(x), p(y)) \le k_f d(x,y) + \varepsilon_f.$$

Note we do not expect to really compute the set $\mathscr{R}_r'^*$ but we will over approximate it. For instance, if we know the algorithm does at most n iterations, we may prove the property for $\mathscr{R}'^n$.

The last condition expresses the need for the program to stop when the variable `m` is close to a normal form.

Condition 5 (C5)**.** *Let $F \subseteq \mathbb{M}$ be the set of the final states reachable by our program. We need:*

$$\forall a \in F, \exists a' \in \bar{\mathbb{M}}, d(a,a') \le \delta$$

Finally, our main theorem is the following.

Theorem 3.1. *If conditions C1 through C5 are satisfied, the function f' computed in a finite-precision semantics and the function f that would be computed in the idealized semantics satisfies the following property:*

$$\forall x, y \in \mathbb{M}, d(f(x), f'(y)) \le k_e(k_f d(x,y) + \varepsilon_f + \delta) + \varepsilon_e + \delta$$

Proof

For any $x \in \mathbb{M}$, we consider the rewriting system given by the $\overset{>}{\to}$ relation. This rewriting system is terminating according to Condition 2. It contains no loops because the value of h strictly decrease at each step. This system is also locally confluent according to Condition 1. Hence, from Condition 2 our rewriting system is globally confluent:

$$\forall a, b, c \in \mathbb{M}, b \;{}^*\!\overset{\le}{\leftarrow} a \overset{>}{\to} *c \implies \exists d \in \bar{\mathbb{M}}, b \overset{>}{\to} *d \;{}^*\!\overset{\le}{\leftarrow} c$$

We can generalize this property where the decreasing condition is removed. Indeed, let $x_1 \to x_2 \to \dots x_n$ a reduction. Let f the normal form of x_1, by applying Condition 1 on x_1 and x_2 leads to $\exists d, x_1 \overset{>}{\to}{}^* d \;{}^*\!\overset{\le}{\leftarrow} x_2$ and then x_2 has f as a normal form. By induction, f is also the normal form of x_n. Hence we have:

$$\forall a, b, c \in \mathbb{M}, b \;{}^*\!\leftarrow a \to^* c \implies \exists d \in \bar{\mathbb{M}}, b \overset{>}{\to}{}^* d \;{}^*\!\overset{\le}{\leftarrow} c \tag{2}$$

Now we have this property on the functions $\mathscr{R}$ we will prove that our program is $P_{k,\varepsilon}$. Let $x \in \mathbb{M}$ as $\mathscr{R}$ is normalizing, there exists a unique $x_f \in \bar{\mathbb{M}}$ such that $x \overset{>}{\to} x_f$. Now we consider $y \in \mathbb{M}$, let $p' \in \mathscr{R}'^*$

the function computed with input y and $p \in \mathscr{R}^*$ the corresponding function. According to Condition 4 $d(p'(y), p(x)) \leq k_f d(x,y) + \varepsilon_f$. Since, according to Condition 5, there exists z in a normal form, $d(z, p'(y)) \leq \delta$. We derive from the two last inequalities: $d(z, p(x)) \leq k_f d(x,y) + \varepsilon_f + \delta$ Then we can apply Condition 3 on input $p(x)$ and z:

$$d(x_f, z) \leq k_e d(p(x), z) + \varepsilon_e$$

So we can derive with a triangular inequality ($d(x_f, p'(y)) \leq d(x_f, z) + d(z, p'(y))$):

$$d(x_f, p'(y)) \leq k_e(k_f d(x,y) + \varepsilon_f + \delta) + \varepsilon_e + \delta$$

□

4 Example: the CORDIC algorithm for computing cosine

In this section, we apply our method to a program implementing the CORDIC algorithm [13]. We consider any finite-precision semantics for which every arithmetic operator has an imprecision of at most ε_e. More precisely: Let `@` be a binary arithmetic operator of our programming language (such as + and $\star$) and let @ and @$'$ be, respectively, the exact semantics and finite-precision semantics for `@`. We shall assume that $d(a@'b, a@b) \leq \varepsilon_e$ for all $a, b \in \mathbb{M}$.

CORDIC (COordinate Rotation DIgital Computer) is a class of simple and efficient algorithms to compute hyperbolic and trigonometric functions using only basic arithmetic (addition, subtraction and shifts), plus table look-up. The notions behind this computing machinery were motivated by the need to calculate the trigonometric functions and their inverses in real time navigation systems. Still nowadays, since the CORDIC algorithms require only simple integer math, CORDIC is the preferred implementation of math functions on small hand calculators.

CORDIC computes using successive approximations: a sequence of successively smaller rotations based on binary decisions hone in on the value we want to find. The CORDIC version illustrated in the program below computes the cosine of any angle in $[0, \pi/2]$.

```
double cos(double beta)
{
    double x = 1, y = 0, x_new, sigma, e = 1E-10;
    int n = 1;
    while(|beta| > e) {
      if(beta > 0)
        sigma=-1;
      else
        sigma=1;
      n = n+1;
      beta += sigma*(PI/(2^n));
      fact = cos(PI/(2^n));      // Values stored
      ts = sigma*tan(PI/(2^n)); // Values stored
      x_new = x + y*ts;
      y = fact * (y - x*ts);
      x = fact * x_new; }
   return x; }
```

Note that this program makes call to trigonometric functions like cosine itself. But in the actual implementation, as it is explained in the comments, these calls (that are done on values divided by successive powers of two) are stored in a database so that no computation of these functions is actually done.

4.1 Scheme instantiation

To apply our method, we have to decompose our program such that it fits the general pattern :

```
foo(m){
     while(! S(m)){
          (i,n) = C(m,n);
          m = R(i,m);
     }
     return m; }
```

Here the function C and R can be instantiated by the following blocks.

```
C(<x,y,beta>,<n>) {
  if(beta > 0)
    sigma=-1;
  else
    sigma=1;
  n = n+1;
  return <sigma,n>; }


R(<sigma,n>,<x,y,beta>) {
   beta += sigma*PI/(2^n);
   fact = cos(PI/(2^n));
   ts = sigma*tan(PI/(2^n));
   x_new = x + y*ts;
   y = fact * (y - x*ts);
   x = fact * x_new;
   return <x,y,beta>;}
```

We also have :

```
S(<x,y,beta>) {
   return |beta| <= e ; }
```

Here m in our pattern stands for the tuple `<x,y,beta>` and `sigma` correspond to `i`.

From `R(<sigma,n>,<x,y,beta>)`, we get our set $\mathscr{R}$. The functions of $\mathscr{R}$ will be indexed by α where $\alpha = \text{sigma}\ \pi/2^n$. Finally, we define the h function on m. We takes $h(\text{<x,y,beta>}) = |\text{beta}|$.

Now, we need to prove that the conditions C1 through C5 of Section 3.2 are satisfied.

4.2 Proofs of the conditions

We sketch the proofs of each of these five conditions below.

1. We have $f_\alpha(f_{\alpha'}(m)) = f_{\alpha+\alpha'}(m)$ due to trigonometric properties. We consider some value $m =$ `<x,y,beta>` and two functions f_α and $f_{\alpha'}$ (for symmetry reason we can consider $\alpha \leq \alpha'$). As $\alpha =$ sigma $\pi/2^n$ and $\alpha' =$ sigma $\pi/2^{n'}$, there exists $q \in \mathbb{N}$ such that $\alpha' = 2^q\alpha$. Then we can prove there exists a unique $k \in \mathbb{Z}$ such that $h(f_\alpha^k(m)) \leq e$ and $f_\alpha(m) \overset{>}{\to}{}^* f_\alpha^k(m) \; {}^*\overset{\leq}{\leftarrow} f'_\alpha(m)$ by using the rewrite rules f_α and $f_{-\alpha}$ the appropriate number of times.
2. The absolute value of `beta` either decreases by at least $|\alpha|$ when the value of beta is greater than `e` or it is in a normal form. As $|\alpha|$ have a minimal value, the system is terminating.
3. If a is a final form, $|a| \leq e$. We know that is $a = 0$ at the end of the program, then the function computes the exacts coordinates which is a $\sqrt{2}$-Lipschitz function. Hence our program is $P_{\sqrt{2},\sqrt{2}e}$.
4. The `R` function contains seven operators. Since `fact` is less than 1, there is no emphasis of the elementary errors inside `R`, so the deviation of `R` is at most $7\varepsilon_e$. The function is also $1+|\alpha|$-Lipschitz. Hence, `R` is $P_{1+|\alpha|,7\varepsilon_e}$. As, `angle` is divided by two at each iteration, the k_f factor for the whole program will be bounded whatever the number of iterations. So we can compute the values for k_f and ε_f.
5. When the stopping condition is reached, the value of *beta* in the exact semantics should be less than e, hence the value in the finite-precision semantics is close to some value smaller than e.

5 Example: Dijkstra's shortest path algorithm

In this section we apply our method to Dijkstra's shortest path algorithm. When given a labelled graph, this algorithm computes the shortest path between a source and any vertex of the graph. In the following implementation of Dijkstra's algorithm, we use the following conventions: the number of vertices is fixed to `n`, all vertices are connected, and the maximum value for a path 999 (some stand-in for infinity).

```
int[] dijkstra( int graph[n][n]){
   int pathestimate[n],mark[n];
   int source,i,j,u,predecessor[n],count=0;
   int minimum(int a[],int m[],int k);
   for(j=1;j<=n;j++){
      mark[j]=0;
      pathestimate[j]=999;
      predecessor[j]=0;}
   source=0;
   pathestimate[source]=0;
   while(count<n){
     if(i==n){
       u=minimum(pathestimate,mark,n);
       mark[u]=1;
       count=count+1;
       i=0;}
     else {
          i=i+1;}
     if(pathestimate[i]>pathestimate[u]+graph[u][i]){
          pathestimate[i]=pathestimate[u]+graph[u][i];
          predecessor[i]=u;}}
     return pathestimate;}

int minimum(int a[],int m[],int k){
     int mi=999;
     int i,t;
     for(i=1;i<=k;i++){
          if(m[i]!=1){
               if(mi>=a[i]){
                    mi=a[i];
                    t=i;}}}
     return t;}
```

We will prove, by instantiating our scheme, that the following program implementing the Dijkstra's algorithm has a finite-precision semantics close to the exact semantics according to the L_1 distance. More precisely, if we assume, as for CORDIC, that the difference between floating point arithmetic and exact arithmetic is at most ε_e for elementary operators then we can prove

$$\forall x, y \in \mathbb{M}, d(f(x) - f'(y)) \leq kd(x,y) + \varepsilon$$

for values of k and ε that we shall show how to compute later and which depend on the desired precision.

5.1 Scheme instantiation

The instantiation of the general pattern is done as follows.

```
C (<graph, pathestimate>, <count, mark>) {
     if(i==n){
       u=minimum(pathestimate,mark,n);
       mark[u]=1;
       count=count+1;
       i=0;
     }
     else {
          i=i+1;
     }
     return <u, i>;
}

R (<u, i>, <graph, pathestimate>) {
        if(pathestimate[i]>pathestimate[u]+graph[u][i]){
             pathestimate[i]=pathestimate[u]+graph[u][i];
        }
  }
}
```

Next, we identify the stopping condition:

```
S(graph,<count,mark>) {
  return count >= n;}
```

Here m is the tuple `<graph, pathestimate>` which is also the Cartesian product of the actual input and the actual output. Finally, we define the function h as the sum of the value of all nodes of `pathestimate`.

5.2 Proofs of the conditions

We now have to prove that the conditions hold for the given instantiations. Once again, we sketch the proofs of these conditions.

1. Let $a \in \mathbb{M}$ and $c \xleftarrow[<u',v'>]{} a \xrightarrow[<u,v>]{} b$ two pairs of nodes where we apply the rules. We can compute that $c \xrightarrow[<u',v'>]{} d \xleftarrow[<u,v>]{} b$ with $h(c) > h(d)$ or $c = d$ and $h(b) > h(d)$ or $c = d$.
2. The system is terminating because each node cannot have a value less than the minimal distance and this value is reached.
3. The function that Dijkstra's algorithm computes a 1-Lipschitz function.
4. The function `R` is P_{1,ε_e}. The number of iteration is bounded by the square of the number of nodes as each iteration mark a node and a node can be marked only one. So any executed path is $P_{1,n^2\varepsilon_e}$.
5. We define the length l of the computation for `pathestimate`as 0 for the source and 1 plus the maximal length of its neighbors with a smaller value than it. We can prove by induction that the error due to finite-precision arithmetic for any node is less than $l\varepsilon_e$.

6 Related Work

Static analysis via abstract interpretation can be an effective method for deriving precise bounds on deviations [6, 7]. Since such static analysis is generally limited to analyzing code line-by-line, significant over approximations might be necessary. For example, when encountering an "if" instruction (or a looping construct), a static analyzer with have to assume that either the control flow is not perturbed by the finite-precision errors (often unrealistic) or the results from the two branches of the conditional must be merged (often causing significant over-approximation). In our examples here, control flow can be perturbed a great deal by precision errors and merging both branches is not a solution as the program is not locally continuous. Our method is useful for solving this problem since it avoids narrowly analyzing the semantics of the conditional.

In the two papers [2, 1], a robustness analysis is done for the Dijkstra's algorithm. The authors split their analysis into two parts: first they prove the continuity of the algorithm and second they prove it is piecewise robust. The problem of discontinuity that can occur at some point of the execution is solved through an abstract language syntax for loops. Like in our theorem, this syntax need additional conditions (mainly the commutativity for two observable equivalent commands). However, their abstract language is more specific than our theorem: CORDIC is not in the scope of the methods used in these these papers. As a result, their conditions are simpler and their proofs are more directed than ours. The other distinction is in the property proved. Their paper aims at furnishing the whole semantics which is an exact one and computational errors are treated qualitatively with the argument that a robust program is not sensitive to small variations. In our analysis, we first prove the program is correct and computes some specified function in the exact semantics and then we prove the finite-precision semantics is close to this function. The last difference is our design for analyzing non-local-robustness: we prefer to consider that a program can hide some structure between its parts instead of rewriting the program into a syntax that hide non-local behavior.

7 Future work and conclusion

We have presented a definition of robustness for a program and we have presented a properties $P_{k,\varepsilon}$ that expresses that the finite-precision semantics is close to the exact one. Then we have presented a theorem that proves this closeness property. The originality of this theorem is that it decomposes the code in several parts and requires condition on these subpart in a non compositional way. It also allow reasoning first on the exact semantics and then about simple properties about the closeness of sub-parts of the code.

We have presented a theorem that allow us to prove the robustness of some floating points programs. This theorem is abstract enough to be applicable in a number of rather different programs: here, we illustrate its use with programs to compute cosine using the CORDIC method and to compute the shortest path in a graph.

For future work, we would like to find other template for other kind of programs that are difficult to prove through compositional analysis. If we get enough theorem based on template like this one, we might be able to find a general method where the proof of robustness of a program is done by identifying a structure inside the code instead of doing it by compositionally.

Also, we are interested in finding other definitions for robustness that are more suitable for the floating point semantics. For instance, to take into account the fact the rounding is proportional to the value in the floating point semantics.

Acknowledgments: We would like to thank Eric Goubault and Jean Goubault-Larrecq for many useful discussions on the topic of this paper

References

[1] Swarat Chaudhuri, Sumit Gulwani, and Roberto Lublinerman. Continuity analysis of programs. In Manuel V. Hermenegildo and Jens Palsberg, editors, *POPL*, pages 57–70. ACM, 2010.

[2] Swarat Chaudhuri, Sumit Gulwani, Roberto Lublinerman, and Sara NavidPour. Proving programs robust. In Tibor Gyimóthy and Andreas Zeller, editors, *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13rd European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, pages 102–112. ACM, 2011.

[3] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numer. Math.*, 1:269–271, 1959.

[4] Ivan Gazeau, Dale Miller, and Catuscia Palamidessi. A non-local method for robustness analysis of floating point programs. Technical report, INRIA, June 2012.

[5] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–47, March 1991.

[6] Eric Goubault. Static analyses of the precision of floating-point operations. In Patrick Cousot, editor, *Static Analysis, 8th International Symposium*, volume 2126 of *Lecture Notes in Computer Science*, pages 234–259. Springer Verlag, 2001.

[7] Eric Goubault and Sylvie Putot. Static analysis of finite precision computations. In Ranjit Jhala and David A. Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, volume 6538 of *Lecture Notes in Computer Science*, pages 232–247. Springer, 2011.

[8] IEEE Task P754. *IEEE 754-2008, Standard for Floating-Point Arithmetic*. IEEE, pub-IEEE-STD:adr, August 2008.

[9] Rupak Majumdar and Indranil Saha. Symbolic robustness analysis. In Theodore P. Baker, editor, *IEEE Real-Time Systems Symposium*, pages 355–363. IEEE Computer Society, 2009.

[10] Rupak Majumdar, Indranil Saha, and Zilong Wang. Systematic testing for control applications. In *MEMOCODE*, pages 1–10, 2010.

[11] The parsec benchmark suite.

[12] Stefan Pettersson and Bengt Lennartson. Stability and robustness for hybrid systems. In *Proceedings of the 35th edition of Decision and Control*, pages 1202–1207, 1996.

[13] Jack E. Volder. The CORDIC trigonometric computing technique. *IRE Transactions on Electronic Computers*, EC-8:330–334, 1959.