A non-local method for robustness analysis of floating point programs

Ivan Gazeau, Dale Miller, Catuscia Palamidessi

A non-local method for robustness analysis of floating point programs *

Ivan Gazeau, Dale Miller, and Catuscia Palamidessi
INRIA and LIX, Ecole Polytechnique

Robustness is a standard correctness property which intuitively means that if the input to the program changes less than a fixed small amount then the output changes only slightly. This notion is useful in the analysis of rounding error for floating point programs because it helps to establish bounds on output errors introduced by both measurement errors and by floating point computation. Compositional methods often do not work since key constructs—like the conditional and the while-loop—are not robust. We propose a method for proving the robustness of a while-loop. 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

Programs using floating point arithmetic are often used for critical applications and it is therefore fundamental to develop methods to establish the correctness of such programs. A central 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. As is well known, floating point arithmetic on these representations is quite different from real number arithmetic: for example, addition is neither commutative nor associative [5].

The developers of floating point programs would like to think in terms of real number semantics instead of the more ad hoc and complicated semantics given by some specific definition of floating point arithmetic, such as the IEEE standard 754 [8]. A central problem in trying to reason 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 what appears to be erratic behavior. The problem is that these constructs are in general *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 either introducing abstractions (e.g., approximations) which can then make conclusions too imprecise, or to undergo a very complex and intricate proof.

In this paper, we will take a different approach: we shall describe some programs where such erratic behavior is recognized and find a way to reason and bound all of that behavior. By moving away from the reasoning using Hoare's style emphasis on local and compositional analysis of a looping program, we are able to avoid reasoning about individual erratic behaviors: instead, we will treat such behaviors as an aggregate and try to bound the behavior of that aggregate.

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

M. Massink and H. Wiklicky (Eds.):Workshop on Quantitative Aspects of Programming Languages and Systems 2012 (QAPL 2012)
EPTCS 85, 2012, pp. 63–76, doi:10.4204/EPTCS.85.5

To illustrate such a possibility in reasoning, consider 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: that is, this algorithm is continuous. However, the actual behavior of the loop and the marking of subsequent nodes can vary greatly with small changes to edge lengths. Our approach to reasoning will allow us to view all of these apparently erratic choices of intermediate paths as an aggregate on which we are able to establish the robustness of the entire algorithm.

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 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

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, changes to the input to a program lead to only proportionally bounded changes to the output. 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. This is due to the following reasons:

1. If we consider a *finite precision semantics*, like floating point implementations, the constant factor k can become much bigger than the one optimal for the exact semantics. For instance, assume that the available representations are the numbers in the set $\{k2^{-32} | k \in \mathbb{Z}\}$ and rounding is done by taking the lower value, and observe that a function like $f : x \mapsto 2^{-4}x$, which is 2^{-4}-Lipschitz

in the exact semantics, is only 1-Lipschitz in this approximate semantics. Indeed, there exist two values that differ by just 2^{-32} and return a result that differ by 2^{-32}. For example, take 1 and $1-2^{-32}$: we have that $f(1) = 2^{-4}$ and $f(1-2^{-32}) = 2^{-4} - 2^{-36}$, but the second result will be rounded down to $2^{-4} - 2^{-32}$.

2. 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 program f which 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

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

Given the above inequality, 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).

These two observations lead us to define another property, $P^1_{k,\varepsilon}$, to capture robustness:

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

This property amends the two previous problems by setting ε to 2^{-32} in the first example and to e in the second example. It also extends the usual definition of the k-Lipschitz property, which can be expressed as $P^1_{k,0}$.

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 $P^2_{k,\varepsilon}$ becomes

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

Finally, since we are studying small deviation, it is not useful to get this property for any i and i' in I but rather when they are close: i.e., $d_I(i,i') \leq \delta$, for suitable values $\delta \in \mathbb{R}^+$. In convex spaces, this property can be easily extended to pairs of inputs having distance more than δ by using intermediate values. So, finally, in this paper we propose the property $P_{k,\varepsilon,\delta}$, described in the following definition.

Definition 2.1. *Let I and R be metric spaces with distance d_I and d_R respectively, $f : I \to R$ a function, $k, \varepsilon \in \mathbb{R}^+$, and let $\delta \in \mathbb{R}^+ \cup \{+\infty\}$. We define the property $P_{k,\varepsilon,\delta}$ for the function f as follows:*

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

3 A schema and its correctness

The main characteristic of our schema is to subdivide the code into several parts instead of analyzing it line by line. Our template, which we show in a moment, divides the data structures in an algorithm into two parts, called A and B. Here, A is the witness to the progress of the algorithm: in particular, the stopping condition will only depend on A (and the input). The structure B is used to accumulate results that provide the answer when the stopping condition is satisfied.

3.1 The schema structure definition

Instead of presenting a formal definition of program schema and matching of code, we illustrate these with the schema in Figure 1.

```
foo(i){
      a = a0;
      b = b0;
      while(S(i,a)){
            c = O(a,b,c,i);
            a = M(a,c);
            b = N(i,b,c);
      }
      return b; }
```

Figure 1: The main template

Here, the schema variables `a`, `b`, `c`, etc, denote tuples of program variables such that no program variable occurs twice among these schema variables. Program expressions such as

```
c = O(a,b,c,i);
```

denotes a program phrase that computes new values for the variables denoted by `c` from values of variables in the tuples `a`, `b`, `c`, and `i`. The actual computation here will be denoted by `O`. This looping program initializes the variables in `a` and `b` with the values in the tuples `a0` and `b0`, respectively. The stopping condition for the loop is given by the boolean valued expression `S(i,a)` and the result of the program is the tuple of values denoted by the variables in `b`.

We shall assume that all program variables are typed in the usual way: variables may range over the values in their associated type. Our analysis of the metric properties of a looping program will, however, consider that tuples of variables, for example, `a` and `b` in Figure 1, range over some *metric space* on the Cartesian product of the variables in the tuple.

3.2 A sufficient condition for robustness

We shall now prove that a program having the generic structure of *foo* given in Figure 1 has, under certain conditions, the property $P_{k,\varepsilon,\delta}$ for some k, ε, δ.

The aim of our method is to postpone the analysis of the exact semantics of commands as far as possible. In order to begin the analysis without specific knowledge of this semantics, we need to manipulate other programs made from the functions O, M, and N that have been identified. For example, the program *listFoo* in Figure 2 will be used to extract the list of values of c obtained for a particular execution of *foo* with input i. The new lines added to *listFoo* will assume the usual semantics for natural numbers.

We now define two new programs. The first is the foo_b program given below: it has the same shape as *foo* but instead

```
ListFoo(i){
      a = a0;
      b = b0;
      j = 0;
      while(! S(i,a)){
            c = O(a,b,c,i);
            j = j+1;
            l[j] = c;
            a = M(a,c);
            b = N(i,b,c); }
      return l; }
```

Figure 2: Collecting `c` values in a list

of setting `c` by the computation of `O(a,b,c,i)`, it sets `c` with
the values of a list given in input. Naturally, the stop condition
for the loop is now that all elements of the list have been accessed. Note that since a was just used in the computation of O, the commands affecting a are now useless and can be removed.

```
foo_b(l,i){
//    a = a0;
      b = b0;
      for(int j = 0; j < l.length; j++ ){
            c = l[j];
//          a = M(a,c);
            b = N(i,b,c); }
      return b; }
```

We have used Java-style instructions such as $l.length$ for the length of the list l and $l[j]$ for the j^{th} element of the list l. (The // syntax is used to form a comment.) We define the new function $foo_B(i,i') = foo_b(listFoo(i),i')$. Notice that $foo_B(i,i) = foo(i)$.

The second program $foo_a(l)$ is the same program as foo_b except that a is returned instead of b. In this program, the lines where b is set are now useless.

```
foo_a(l){
      a = a0;
//    b = b0;
      for(int j = 0; j < l.length; j++ ){
            c = l[j];
            a = M(a,c);
//          b = N(i,b,c);
      }
      return a; }
```

Finally, we define $foo_A(i) = foo_a(listFoo(i))$. The two function foo_A and foo_B and relations between them will be used to indirectly analyze the program *foo*.

In what follows, we use the following conventions: the domain of the variables a, b, c, and i are A, B, C and I, respectively, and $a0$ and $b0$ are some determined constants of type A and B respectively. For every type X, the expression X^* denote the type of lists of type X.

We now introduce four conditions that need to hold to prove that the *foo* program satisfies $P_{k,\varepsilon,\delta}$ for appropriate values of k, ε, and δ. These conditions apply to *eight* parameters: namely, $\delta, k_{N^*}, \varepsilon_{N^*}, K_A, \varepsilon_2$, $K_s, \varepsilon_s, \varepsilon_t$. Condition $C1$ expresses the property $P_{k_{N^*},\varepsilon_{N^*},\delta}$ for the transformed program foo_B, condition $C2$ expresses the fact that there is a relationship between the values stored in A and the values stored in B, and condition $C3$ and $C4$ address the stability of the stop condition $S(i,a)$.

Condition 3.1 (C1). $\forall l \in C^*.P_{k_{N^*},\varepsilon_{N^*},\delta}(\lambda z.foo_b(l,z))$.

The next condition states that whenever two inputs i and i' are within a δ of each other then it is the case that if their images in A (under foo_a) are close, then their images in B (under foo_b) are close.

Condition 3.2 (C2).

$$\forall i_1, i \in I, d_I(i,i_1) \leq \delta \implies d_B(foo_B(i,i), foo_B(i_1,i)) \leq k_A d_A(foo_A(i_1), foo_A(i)) + \varepsilon_2$$

The stopping condition S should satisfy the following two conditions. The first expresses that the boundary of the region $\{a \mid S(i,a)\}$ cannot vary too much.

Condition 3.3 (C3)**.**

$$\forall a \in A, \forall i, i' \in I, d_I(i,i') \leq \delta \wedge S(i',a) \implies \exists a' \in A, d_A(a,a') \leq k_s d_I(i',i) + \varepsilon_s \wedge S(i,a')$$

The following condition on S states that the diameter of the region $\{a \mid S(i,a)\}$ is as small as the desired precision.

Condition 3.4 (C4)**.**

$$\forall a, a' \in A, \forall i \in I, S(i,a) \wedge S(i,a') \implies d_A(a,a') \leq \varepsilon_t$$

Finally, our main theorem is the following.

Theorem 3.1. *If the program foo terminates and the conditions C1, C2, C3, and C4 hold, then* $P_{k_0,\varepsilon_0,\delta}$ *holds for the function computed by foo with* $k_0 = k_{N^*} + k_A k_s$ *and* $\varepsilon_0 = \varepsilon_{N^*} + k_A(\varepsilon_s + \varepsilon_t) + \varepsilon_2$.

Proof In the proof, we will use these two observations:

1. Since $listFoo(i)$ is obtained from the computation of $foo(i)$, and since $foo_B(i,i')$ replaces the result of O by this list, if we compute $foo_B(i,i)$ we are replacing each value for c by itself. Therefore we have that $foo(i) = foo_B(i,i)$.
2. In the execution of $foo(i)$, the final value of a that satisfies the stopping condition $S(i,a)$ is $foo_A(i)$.

By the observation 1, proving the theorem is equivalent to proving

$$\forall i, i0 \in I, d_I(i,i0) \leq \delta \implies d_B(foo_B(i,i), foo_B(i0,i0)) \leq k_0 d_I(i,i0) + \varepsilon_0.$$

By condition C1, choosing $l = listFoo(i0)$, we have

$$\forall i, i0 \in I, d_I(i,i_0) \leq \delta \implies d_B(foo_b(listFoo(i0),i0), foo_b(listFoo(i0),i)) \leq k_{N^*} d_I(i,i0) + \varepsilon_{N^*}.$$

By definition of foo_B, we have

$$\forall i, i0 \in I, d_I(i,i_0) \leq \delta \implies d_B(foo_B(i0,i0), foo_B(i0,i)) \leq k_{N^*} d_I(i,i0) + \varepsilon_{N^*}. \tag{1}$$

From observation 2, $S(i0, foo_A(i0))$ holds. By condition C3 (instantiating i' with $i0$) we derive that:

$$\forall i, i0 \in I, d_I(i,i_0) \leq \delta \implies \exists a' \in A, d_A(foo_A(i0),a') \leq k_s d_I(i,i0) + \varepsilon_s \wedge S(i,a'). \tag{2}$$

Hence, by observations 2 and 1, $S(i, foo_A(i))$ also holds. From inequality (2) and condition C4, we derive

$$d_A(a', foo_A(i)) \leq \varepsilon_t. \tag{3}$$

From the last inequality and from inequality (2), we derive, using the triangle inequality

$$d_A(foo_A(i0), foo_A(i)) \leq k_s d_I(i,i0) + \varepsilon_s + \varepsilon_t. \tag{4}$$

From condition C2 and inequality (4), we have

$$\forall i, i0 \in I, d_I(i,i0) \leq \delta \implies d_B(foo_B(i0,i), foo_B(i,i)) \leq k_A(k_s d_I(i,i0) + \varepsilon_s + \varepsilon_t) + \varepsilon_2. \tag{5}$$

From inequalities (1) and (5), using the triangle inequality, we derive

$$\begin{gathered} \forall i, i0 \in I, d_I(i,i0) \leq \delta \\ \implies \\ d_B(foo_B(i,i), foo_B(i0,i0)) \leq k_{N^*} d_I(i,i0) + \varepsilon_{N^*} + k_A(k_s d_I(i,i0) + \varepsilon_s + \varepsilon_t) + \varepsilon_2. \end{gathered}$$

Finally, we define $\varepsilon_0 = \varepsilon_{N^*} + k_A(\varepsilon_s + \varepsilon_t) + \varepsilon_2$ and $k_0 = k_{N^*} + k_A k_s$. □

4 Example: the CORDIC algorithm for computing cosine

In this section we apply our method to a program implementing the CORDIC algorithm [13], and we prove that it is $P_{k,\varepsilon,\infty}$.

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 lookup. 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 now-a-days, since the CORDIC algorithms require only simple integer math, CORDIC is the preferred implementation of math functions on small hand calculators.

CORDIC is a successive approximation algorithm: A sequence of successively smaller rotations based on binary decisions drives the algorithm towards 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, theta = 0, sigma, e = 1E-10;
    int Pow2=1;
    while(|theta - beta| > e) {
         Pow2 *= 2;
         if(beta > theta)
           sigma=1;
         else
           sigma=-1;
         sigma=sigma/Pow2;
         theta += atan(sigma); // Value stored
         fact= cos(atan(sigma)); // Value stored
         x_new = x + y*sigma;
         y = fact * (y - x*sigma);
         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 first of all to instantiate the schema variables A, B, C (cf. Section 3.1) with a suitable partition of the variables of the program. The variables I are determined: they must be instantiated with the variables which represent the input.

In this example the partition for the variables will be the following.

```
A := double theta;
B := double x,y;
C := double sigma;
I := double beta;
```

We now must define a suitable metric on the types of the variables in A and B. We choose the following:

- d_A is the usual distance on $\mathbb{R}$.
- d_B is the L_2 norm on $\mathbb{R}^2$.

Now we need to identify the stopping condition $S(i,a)$. This is given by:

```
S(beta,theta) := | theta - beta | <= e
```

Then, we need to instantiate the functions $M(a,c)$, $N(i,b,c)$, $O(a,b,c,i)$ of the schema with suitable regions of code. We choose these as follows:

```
O(theta,<x,y>,sigma,beta) {
    Pow2 *= 2;
    if(beta > theta)
      sigma=1;
    else
      sigma=-1;
    sigma=sigma/Pow2;
    return sigma; }

M(theta,sigma) {
    theta += atan(sigma);
    return theta; }

N(beta,<x,y>,sigma) {
    fact = cos(atan(sigma));
    x_new = x + y*sigma;
    y = fact * (y - x*Pow2);
    x = fact * x_new;
    return <x,y>; }
```

Finally, we need to prove that the conditions C1, C2, C3, and C4 (cf. Section 3.2) are satisfied.

4.2 Proofs of the conditions

C1: $\forall l \in C^*.P_{k_{N^*},\varepsilon_{N^*},\delta}(\lambda z.foo_b(l,z))$ This condition can be proved for the following program by such standard techniques as abstract interpretation or Hoare triples.

```
double cos(double beta, int[] listFoo)
{
    double x = 1, y = 0, x_new, theta = 0, sigma = 0,e = 1E-10;
    int Pow2=1;
    for(int j=O;j<listFoo.length;j++) {
         sigma=listFoo[j];
         fact = cos(sigma);
         x_new = x + y*sigma;
         y = fact * (y - x*sigma);
         x = fact * x_new;
```

```
    }
    return x*K;
}
```

C2: $\forall i_1, i \in I, d_I(i,i_1) \leq \delta \implies d_B(foo_B(i,i), foo_B(i_1,i)) \leq k_A d_A(foo_A(i_1), foo_A(i)) + \varepsilon_2$ This part of the proof is rather technical. The interested reader can find it in the appendix of [4]. The proof of C2 is the most difficult part of this example. We have proved it "by hand", and we do not claim that there is an easy way to automate it. However, this proof points out that we can prove the intended property without considering the whole semantics of the program, but just the relevant properties.

C3: $\forall a \in A, \forall i, i' \in I, d_I(i,i') \leq \delta \wedge S(i',a) \implies \exists a' \in A, d_A(a,a') \leq k_s d_I(i',i) + \varepsilon_s \wedge S(i,a')$ The instantiation of $S(i,a)$ corresponds to $|i-a| \leq e$, so C3 is given by the condition:

$$\forall a \in A, \forall i, i' \in I, |i-a| \leq e, \exists a' \in I, |a-a'| \leq k_s |i-i'| + \varepsilon_s \wedge |i'-a'| \leq e$$

We can satisfy this property by setting $a' = a + i' - i$, $k_s = 1$, and $\varepsilon_s = 0$.

C4: $\forall a, a' \in A, \forall i \in I, S(i,a) \wedge S(i,a') \implies d_A(a,a') \leq \varepsilon_t$ C4 can be rewritten, once we instantiate $S(i,a)$ to

$$\exists \varepsilon_t, \forall a, a' \in A, \forall i \in I, |i-a| \leq e \wedge |i-a'| \leq e \implies |a-a'| \leq \varepsilon_t$$

Which is true for $\varepsilon_t = 2e$.

5 Example: Dijkstra's shortest path algorithm

In this section we apply our method to Dijkstra's shortest path algorithm. This is an algorithm that, given a graph, computes the shortest path between a source and any vertex of the graph. We will prove, by instantiating our schema, that the following program implementing the Dijkstra's algorithm can be proved $P_{1,0,0}$ in the semantic of real numbers using our theorem.

In the following program we use some conventions: the number of vertices is fixed to w, all vertices are connected, and the maximum value for a path is 999 (some stand-in of infinity).

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

```
      for(i=1;i<=w;i++){
         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;}
```

5.1 Scheme instantiation

To apply our theorem, we have to instantiate the scheme variables A, B, C with some variables of the program. The variables of I are instantiated with the variables that represent the input. We choose the following instantiation: A contains the variables *count* and *mark*, B the array of double *pathestimate* and C the variable u which identify the current vertex to propagate.

```
A := int count;int mark[w];
B := pathestimate[w];
C := int u;
I := graph[w][w];
```

We now have to choose a suitable metric on the types of the variables, and we choose the following: d_I is the L_1 norm on an array of real numbers, d_B is the L_∞ norm on array of real numbers and d_A is the identity metric: that is, the distance between two elements of A is 0 if they are the same elements and it is ∞ otherwise.

Next, we identify the stopping condition:

```
S(graph,<count,mark>) := count >= w
```

Finally, we identify the functions $M(a,c)$, $N(i,b,c)$, $O(a,b,c,i)$ with the following regions of code:

```
O (count, mark, pathestimate, u, graph) {
   u=minimum(pathestimate,mark,w);
   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;
   }
   return u;
}

M (<mark, count>, u) {
   mark[u]=1;
   count=count+1;
   return <mark,count>;
}

N (graph, pathestimate, u) {
   for(i=1;i<=w;i++){
        if(pathestimate[i]>pathestimate[u]+graph[u][i]){
             pathestimate[i]=pathestimate[u]+graph[u][i];
        }
   }
   return pathestimate;
}
```

We now have to prove that the conditions C1, C2, C3 and C4 hold for the given instantiations.

5.2 Proof of the conditions

C1: $\forall l \in C^*.P_{k_{N^*},\varepsilon_{N^*},\delta}(\lambda z.foo_b(l,z))$ For all $i0 \in I$, $foo_a(i0,i)$ is k-Lipschitz and k does not depend on $i0$. The proof of this condition can be done by using standard technical (such as Hoare triples or abstract interpretation) on the following program.

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

```
                }
            }
        }
        return pathestimate;
}
```

In an exact semantics (with real numbers), this program is 1-Lipschitz as any element of *pathestimate* is the sum of some elements of *graph*. If the analysis is done with an exact semantics (with real numbers), we are able to prove that this program is 1-Lipschitz.

C2: $\forall i_1, i \in I, d_I(i,i_1) \leq \delta \implies d_B(foo_B(i,i), foo_B(i_1,i)) \leq k_A d_A(foo_A(i_1), foo_A(i)) + \varepsilon_2$ The proof for C2 is rather technical. The basic idea is however quite simple. Indeed, the A structure is a set in a discrete space on which elements are added. So we prove that whatever the order of the element is B is constant. This is done by showing that local transpositions do not change the result. So the principle should apply in other algorithms with the same A structure. The complete proof can be found in the appendix of [4].

C3: $\forall a \in A, \forall i, i' \in I, d_I(i,i') \leq \delta \wedge S(i',a) \implies \exists a' \in A, d_A(a,a') \leq k_s d_I(i',i) + \varepsilon_s \wedge S(i,a')$ Since the instantiation of $S(i',a)$ is `count >= w`, the stopping condition does not depend on i (when the number of nodes w is fixed). Hence, the formula is satisfied for $a' = a$ with the constant $k_s = 0$ and $\varepsilon_s = 0$.

C4: $\forall a, a' \in A, \forall i \in I, S(i,a) \wedge S(i,a') \implies d_A(a,a') \leq \varepsilon_t$ Since $\{a | S(i,a)\}$ is a singleton for every i (it corresponds to the state where all the nodes are marked), the property holds for $\varepsilon_t = 0$.

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 will 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], 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 these papers which also means their conditions are simpler and their proofs are more directed than ours. The other distinction is in the semantics of the language. 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. With our analysis, we give a quantitative definition of what small enough

means. The last difference is our design for analyzing non-local-robustness. We prefer to consider non-local behaviors as happening and solving them by a program transformation using pattern than to rewrite the program in a syntax that hide the non-local behavior.

7 Future work and conclusion

We have presented a theorem that allows us to prove the robustness of some floating point 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 address a key possible weakness of our method: it is currently tied to a particular template. Although that template is presented abstractly, there should certainly be ways to improve the generality beyond the matching of a template. Also, since the property $P_{k,\varepsilon,\delta}$ (Definition 2.1) is more general than both k-Lipschitz and the other definitions of robustness [9, 10], we would like to explore applications of this property to cases where neither of the other definitions work.

Condition C2 is, at least in the examples considered in this paper, the most difficult condition to verify. This suggests that we might consider more restrictive conditions that would entail C2.

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

References

[1] Swarat Chaudhuri, Sumit Gulwani & Roberto Lublinerman (2010): *Continuity analysis of programs*. In Manuel V. Hermenegildo & Jens Palsberg, editors: *POPL*, ACM, pp. 57–70, doi:10.1145/1706299.1706308.

[2] Swarat Chaudhuri, Sumit Gulwani, Roberto Lublinerman & Sara NavidPour (2011): *Proving programs robust*. In Tibor Gyimóthy & 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*, ACM, pp. 102–112, doi:10.1145/2025113.2025131.

[3] E. W. Dijkstra (1959): *A Note on Two Problems in Connexion with Graphs*. *Numer. Math.* 1, pp. 269–271.

[4] Ivan Gazeau, Dale Miller & Catuscia Palamidessi (2012): *A non-local method for robustness analysis of floating point programs*. Technical Report, INRIA. Available at `http://hal.inria.fr/hal-00665995`.

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

[6] Eric Goubault (2001): *Static Analyses of the Precision of Floating-Point Operations*. In Patrick Cousot, editor: *Static Analysis, 8th International Symposium*, *Lecture Notes in Computer Science* 2126, Springer Verlag, pp. 234–259.

[7] Eric Goubault & Sylvie Putot (2011): *Static Analysis of Finite Precision Computations*. In Ranjit Jhala & David A. Schmidt, editors: *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, *Lecture Notes in Computer Science* 6538, Springer, pp. 232–247, doi:10.1007/978-3-642-18275-4.

[8] IEEE Task P754 (2008): *IEEE 754-2008, Standard for Floating-Point Arithmetic*. IEEE, pub-IEEE-STD:adr, doi:10.1109/IEEESTD.2008.4610935.

[9] Rupak Majumdar & Indranil Saha (2009): *Symbolic Robustness Analysis*. In Theodore P. Baker, editor: *IEEE Real-Time Systems Symposium*, IEEE Computer Society, pp. 355–363, doi:10.1109/RTSS.2009.17.

[10] Rupak Majumdar, Indranil Saha & Zilong Wang (2010): *Systematic testing for control applications*. In: *MEMOCODE*, pp. 1–10, doi:10.1109/MEMCOD.2010.5558629.

[11] *The Parsec benchmark suite*. Available at `http://parsec.cs.princeton.edu/`.

[12] Stefan Pettersson & Bengt Lennartson (1996): *Stability And Robustness For Hybrid Systems*. In: *Proceedings of the 35th edition of Decision and Control*, pp. 1202–1207.

[13] Jack E. Volder (1959): *The CORDIC Trigonometric Computing Technique*. *IRE Transactions on Electronic Computers* EC-8, pp. 330–334.