Non-local robustness analysis via rewriting techniques[☆]

Ivan Gazeau, Dale Miller, Catuscia Palamidessi

INRIA & LIX/Ecole Polytechnique
Palaiseau, France

Abstract

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 that some while-loop programs always returns finite precision values close to the exact values. Our method uses techniques borrowed from rewriting theory 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, rewriting techniques

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

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

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.

Traditional compositional methods do not work well for proving robustness properties of programs. 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 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 properties. In particular, we show that it is possible to prove that a looping program is robust even though the individual unfoldings of the while loop 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 modeling 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 our previous workshop paper [4]. The current paper provides for a simplified set of conditions on looping programs in order to show their robustness. The simplification has been achieved by introducing a

rewriting framework.

Plan of the paper. In the next section we introduce the technical concepts that will allow us to prove some programs to be robust: one such concept is a generalization of the k-Lipschitz condition. Section 3 contains our main contribution: we present a schema for reasoning about robustness in programs and prove the schema's correctness. We then show the applicability of our proposal in two examples: the CORDIC algorithm for computing cosine is presented in Section 4 and Dijkstra's shortest-path algorithm is 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, various definitions of robustness have been considered. One definition, used by Chaudhuri et al [1], is based on continuity. In a subsequent paper [2], the authors considered a stronger notion of robustness, namely the k-Lipschitz property. Being k-Lipschitz for a program means that for every change of the input, the output changes proportionally. Another approach was used by Majumdar et al in [9, 10], where robustness was 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 are interested in errors not only due to inputs but also to internal computation error. Thus, we cannot simply model the effects of small variations in a program's input, we must also consider arithmetic errors (eg, round-offs) that can occur internal to the execution of a program. We will be concerned with the size of the gap between the computed (approximate) result and the exact theoretical result of a program. In a formal setting, we have to consider two semantics for any program: one where all expression return the exact mathematical result and another one where expressions are evaluated with rounding errors. So, for any program $\mathtt{P}$, we denote by $[\![\mathtt{P}]\!]_e$ its exact semantics and by $[\![\mathtt{P}]\!]_f$ its finite-precision semantics. Even if the finite-precision semantics is well defined, as is the case with, say, the IEEE standard 754, we will not based our computations on such a definition since they typically do not exhibit good mathematical property. Instead we will only work with the assumption that the distance between the exact and the finite-precision semantics for the atomic operations on reals is "small enough", namely uniformly bounded by some constant ϵ_e. More precisely: let *op* be a binary arithmetic operator of our programming language (such as + and $\star$) and let op and op' be, respectively, the exact semantics and finite-precision semantics for *op*. We shall assume that $|a\ op'\ b - a\ op\ b| \leq \epsilon_e$ for all $a, b \in \mathbb{R}$.

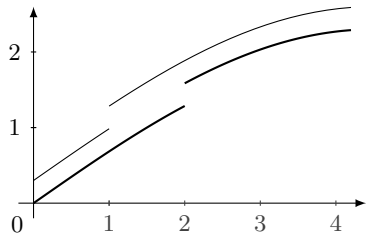

Figure 1: An example of functions that are (k, ϵ)-close

A standard way to compare two functions $f, f' : \mathbb{M} \to \mathbb{M}'$ where $(\mathbb{M}', d)$ is a metric is to use the infinity norm:

Definition 2.1 (L^∞ distance)**.** *The L^∞ distance between two functions is defined as:*

$$d_\infty(f, f') = \max_{x \in \mathbb{M}}(d(f(x), f'(x))).$$

Using this definition, we can consider that two functions are close if this distance is less than some ϵ:

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

However, this definition has the disadvantage of not being compositional. Indeed, consider two programs $\mathtt{F}$ and $\mathtt{G}$ with the following exact semantics on $\mathbb{R}$:

$$[\![\mathtt{F}]\!]_e(x) = \begin{pmatrix} 0 & x < 0 \\ 1 & x \geq 0 \end{pmatrix} \qquad [\![\mathtt{G}]\!]_e(x) = x$$

Depending on the finite-precision semantics of the programs that compute f and g, we can have $[\![\mathtt{F}]\!]_f(x) = [\![\mathtt{F}]\!]_e(x)$ while $[\![\mathtt{G}]\!]_f \neq [\![\mathtt{G}]\!]_e(x)$, and in particular we could have that $[\![\mathtt{G}]\!]_e(0) = 0$ while $[\![\mathtt{G}]\!]_f(0) = -\epsilon$. In this scenario we have $[\![\mathtt{F}]\!]_e \circ [\![\mathtt{G}]\!]_e(0) = 1$ while $[\![\mathtt{F}]\!]_f \circ [\![\mathtt{G}]\!]_f(0) = 0$.

For this reason we define a stronger property that we call (k, ϵ)-closeness property:

Definition 2.2 ((k, ϵ)-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 $k, \epsilon \in \mathbb{R}^+$. We say that f' is (k, ϵ)-close to f if the following holds:*

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

With this definition, we ensure that if there were an error on the input the output error would still be bounded. In fact, we have a weak compositionality property:

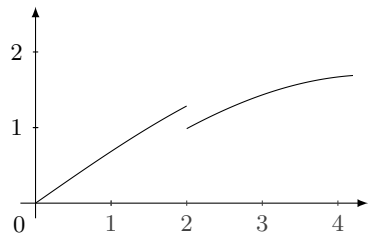

Figure 2: A $P(k, \epsilon)$ function

if f and f' are (k, ϵ)-close and g and g' are (k', ϵ')-close then $f \circ g$ and $f' \circ g'$ are $(kk', \epsilon + k\epsilon')$-close.

We should notice that not all functions are (k, ϵ)-close to themselves. We define the property $P(k, \epsilon)$ for a function as the property of being (k, ϵ)-close to itself.

Definition 2.3. *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 $k, \epsilon \in \mathbb{R}^+$. We say that f is $P(k, \epsilon)$ if*

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

If a function is not $P(k, \epsilon)$ then even if its implementation would not introduce any floating-point error, the error on the input is sufficient to generate a large error on the output. The $P(k, \epsilon)$ property will play an important role in our proof method. Intuitively, it is crucial because, if a function is not (k, ϵ)-close to itself, it has no hope to be (k, ϵ)-close to its approximation. Let us compare the $P(k, \epsilon)$ property with the notions of robustness considered in the literature.

The notions of robustness considered in [1, 2] are the continuity property and the k-Lipschitz property respectively. The continuity property, that for a function f on reals is defined as:

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

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.

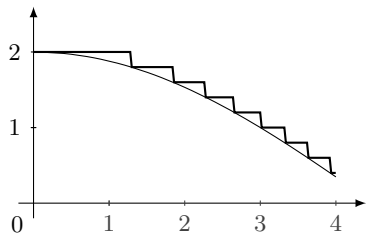

Figure 3: The function g^{-1} and its approximation

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).*

For this reason, we will use the $P(k, \epsilon)$ property that extends the k-Lipschitz property since k-Lipschitz is equivalent to $P(k, 0)$. Note that while the function f in the above example is not k-Lipschitz, but is $P(k, \mathrm{e})$.

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

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

Figure 4: The main template in imperative style

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 4.

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 $[\![\texttt{foo}]\!]_e : \mathbb{M} \to \mathbb{M}$. In practice this is always possible: if the input domain In and the output domain Out are different, then we can just define $\mathbb{M} = In \times Out$. 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.

The body of the loop is decomposed into two parts of code: `C` and `R`. The idea is that `C` select the control flow by returning `i`. As `C` can have erratic behavior, any compositional analysis is unfeasible. `R`, on the other hand, does the actual computation on m. It 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`.

To formalize the interactions between `C` and `R` we split the internal variables into three disjoint subsets: `m`, `n` and `i`. The syntax `(i,n) = C(m,n,i)` means that `C` have access to all variables of the program (`m`, `n` and `i`) but cannot update the value of m (it can only change the value of `i` and `n`). Similarly, the syntax `m = R(i,m)` represents the program fragment that can access the groups of variables `i` and `m` and just changes the value of `m`. Finally, `S(m,n,i)` is the stopping condition (of type boolean) : the program stops when $[\![\texttt{S(m,n,i)}]\!]_f = \top$, we add a condition on `S(m,n,i)` in the next section. .

In addition to these syntactic restrictions 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 $\mathcal{R}$ and $\mathcal{R}'$ the sets of these functions, namely $\mathcal{R} = \{f_i \mid i \in I\}$, and similarly $\mathcal{R}' = \{f'_i \mid i \in I\}$. Finally, the closure of $\mathcal{R}$ (resp. $\mathcal{R}'$) under function composition is denoted by $\mathcal{R}^*$ (resp. $\mathcal{R}'^*$).

3.1. The basic rewriting framework

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

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

This rewriting system can not have any termination property since for any element of $\mathbb{M}$, it is always possible to apply all the rules in $\mathcal{R}$. In fact, we will be more interested in a rewriting sub-system where some rules are removed depending on a condition on m. To make this restriction, we will assume the existence of a function $h : \mathbb{M} \to \mathbb{R}$ that will be used to compare the elements of $\mathbb{M}$.

Given such function h, we define the following relations, where $<$ is the standard ordering on $\mathbb{R}$.

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

3.2. A sufficient condition for robustness

We now list five conditions. The idea is that if these conditions hold for a particular program `foo` that fits the schema in Figure 4, then we are able to justify the closeness property holds for the functions $[\![\texttt{foo}]\!]_e$ and $[\![\texttt{foo}]\!]_f$ for some parameters.

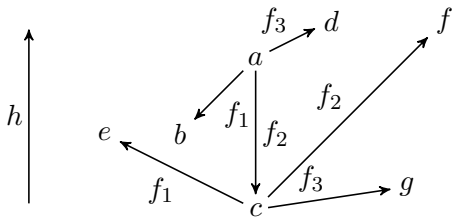

Figure 5: In this example, while the complete rewriting system contains three rules for a and c, the sub-system with the decreasing condition is such that c is a terminal state.

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. *This condition corresponds to requiring that the rewriting system satisfies the following:*

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

Next condition requires the idealized system to be noetherian.

Condition 2. *The rewriting system $\xrightarrow{>}$ is terminating: that is, there exists no infinite path $a_1 \xrightarrow{>} a_2 \xrightarrow{>} \ldots$.*

The next conditions are about the exact semantics. The first one ask the "smoothness" of the exact function, the second one require the `C` code selects only rules in the $\xrightarrow{>}$ sub-system and the third one, the stopping condition corresponds to reaching a final state.

Condition 3. *The function $[\![\texttt{foo}]\!]_e$ is $P(k_e, \epsilon_e)$*

Condition 4.

$$[\![\texttt{S(m, n, i)}]\!]_e = \top \implies m \in \bar{\mathbb{M}}'$$

In addition of these constraints on the exact 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 5. *Let $p' = f'_{i_1} \circ f'_{i_2} \circ \ldots f'_{i_n}$ be the composition of a possible execution of the program over the finite precision semantics. Let $p = f_{i_1} \circ f_{i_2} \circ \ldots f_{i_n}$. We require the closeness property between p and p':*

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

Remark 1. *Depending on the exact code of the program we can get some properties about the possible paths over the approximate semantics for instance, we might be able to bound the number of iterations. This will allow us to consider a subset of $\mathcal{R}'^*$ otherwise since $\mathcal{R}'^*$ contains arbitrary long path, there is no chance the closeness property holds for any path of $\mathcal{R}'^*$.*

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

Condition 6.

$$[\![\texttt{S(m,n,i)}]\!]_f = \top \implies \exists m' \in \bar{\mathbb{M}}, \quad d(m, m') \leq \delta$$

Finally, our main theorem is the following.

Theorem 3.1. *If the six conditions are satisfied, the function $[\![\texttt{f○○}]\!]_f$ computed in a finite-precision semantics and the function $[\![\texttt{f○○}]\!]_e$ that would be computed in the idealized semantics satisfies the closeness property:*

$$\forall x, y \in \mathbb{M}, d(f(x), f'(y)) \leq k_e(k_f d(x, y) + \epsilon_f + \delta) + \epsilon_e + \delta$$

Proof

For any $x \in \mathbb{M}$, we consider the rewriting system given by the $\xrightarrow{>}$ relation. This rewriting system is terminating according to Condition 2. This system is also locally confluent according to Condition 1. Hence, our rewriting system is globally confluent:

$$\forall a, b, c \in \mathbb{M}, b \,{}^*\!\xleftarrow{<} a \xrightarrow{>} *\, c \implies \exists d \in \bar{\mathbb{M}}, b \xrightarrow{>} *\, d \,{}^*\!\xleftarrow{<} c \tag{2}$$

Hence, every element $a \in \mathbb{M}$ has a unique normal form b (which means $a \xrightarrow{>}{}^* b$ and there is no c, $b \xrightarrow{>} c$).

Now, we will generalize the global confluence property by removing the decreasing condition from the left hand side of the implication. 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 $\exists d, x_1 \xrightarrow{>}{}^* d \,{}^*\!\xleftarrow{<} x_2$ and then x_2 has f as a normal form. By induction, f is also the normal form of x_n. Hence we have:

$$\forall a, b, c \in \mathbb{M}, b \,{}^*\!\leftarrow a \to^* c \implies \exists d \in \bar{\mathbb{M}}, b \xrightarrow{>}{}^* d \,{}^*\!\xleftarrow{<} c \tag{3}$$

Now we have this property, we can prove the closeness property between $[\![\texttt{f○○}]\!]_f$ and $[\![\texttt{f○○}]\!]_e$. Let $x, y \in \mathbb{M}$, we have to prove there exist k and ϵ such that

$$d([\![\texttt{f○○}]\!]_e(x), [\![\texttt{f○○}]\!]_f(y)) \leq k d(x, y) + \epsilon$$

.

We note $[\![\texttt{f}\circ\circ \,]\!]_e(x) = f_{i_n} \circ f_{i_{n-1}} \circ \dots f_1(x)$ the composition made by $\texttt{f}\circ\circ$ during its execution on the exact semantics on input x. From the rewriting perspective, we note $x = a_0 \xrightarrow{>} a_1 \xrightarrow{>} \dots \xrightarrow{>} a_n = [\![\texttt{f}\circ\circ \,]\!]_e(x)$ the intermediate states of the execution (the figure 6 represents all the path used in this proof). We have proved the $\xrightarrow{>}$ rewriting system to be normalizing so there exists a unique normal form x_f for x. Moreover, we can derived from the property 3 $a_n \xrightarrow{>}{}^* x_f$. Then, since from Condition 4 we have $a_n \in \bar{\mathbb{M}}$, we can conclude $a_n = x_f$.

We also note $[\![\texttt{f}\circ\circ \,]\!]_f(y) = f'_{j'_n} \circ f'_{j_{n'-1}} \circ \dots f'_{j_1}(y)$ the composition made by $\texttt{f}\circ\circ$ during its execution on the finite precision semantics on input y.

We now consider the function $p(x) = f_{j'_n} \circ f_{j_{n'-1}} \circ \dots f_{j_1}(x)$ corresponding to the finite precision control flow applied to the exact value and the exact computation. According to this definition we have $x \to^* p(x)$. Since $x \to^* p(x)$ and $x \xrightarrow{>}{}^* [\![\texttt{f}\circ\circ \,]\!]_e(x)$ we derived from the property 3: $\exists d \in \bar{\mathbb{M}}, p(x) \xrightarrow{>}{}^* d \,{}^* \xleftarrow{<} [\![\texttt{f}\circ\circ \,]\!]_e(x)$. Then, since $[\![\texttt{f}\circ\circ \,]\!]_e(x) \in \bar{\mathbb{M}}$, $d = [\![\texttt{f}\circ\circ \,]\!]_e(x)$.

Now, Condition 6 provide some $z \in \bar{\mathbb{M}}$ with $d([\![\texttt{f}\circ\circ \,]\!]_f(y), z) \leq \delta$. From this inequality and the inequality of the Condition 5, we derived with a triangular inequality:

$$d(z, p(x)) \leq k_f d(x, y) + \epsilon_f + \delta$$

Then we can apply Condition 3 on input $p(x)$ and z:

$$d([\![\texttt{f}\circ\circ \,]\!]_e(x), z) \leq k_e d(p(x), z) + \epsilon_e$$

So we can derive with a triangular inequality:

$$d([\![\texttt{f}\circ\circ \,]\!]_e(x), [\![\texttt{f}\circ\circ \,]\!]_f(y)) \leq d(x_f, z) + d(z, [\![\texttt{f}\circ\circ \,]\!]_f(y)).$$

We get:

$$d([\![\texttt{f}\circ\circ \,]\!]_e(x), [\![\texttt{f}\circ\circ \,]\!]_f(y)) \leq k_e(k_f d(x, y) + \epsilon_f + \delta) + \epsilon_e + \delta$$

Therefore $[\![\texttt{f}\circ\circ \,]\!]_e$ and $[\![\texttt{f}\circ\circ \,]\!]_f$ are $(k_e k_f, k_e(\epsilon_f + \delta) + \epsilon_e + \delta)$-close. □

4. Example: the CORDIC algorithm for computing cosine

In this section, we apply our method to a program implementing the CORDIC algorithm [13]. We assume the finite-precision semantics have the properties detailed in section 3.2.

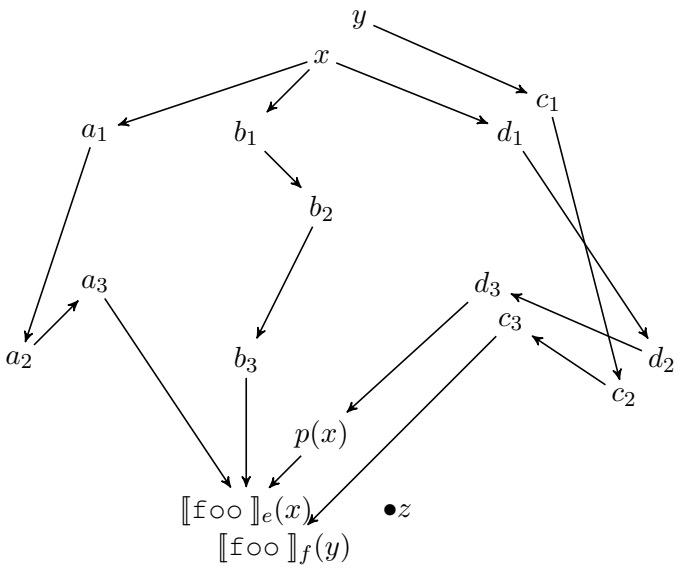

Figure 6: A picture of the proof
The a path correspond to the exact computation.
The b path is a theoretical path containing only $\xrightarrow{>}$ rules.
The c path is the actual path.
The d path mixes the control flow of the finite precision semantics with the computations of the exact semantics.

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 of section 3:

```
foo(m){
     n,i initialized;
     while(! S(m,n,i)){
```

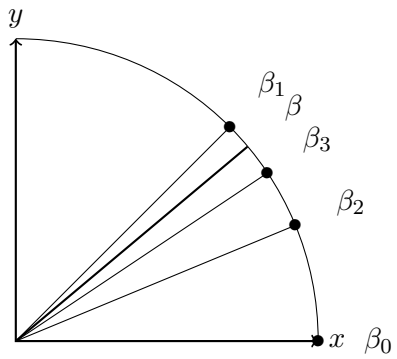

Figure 7: some steps of the CORDIC algorithm

```
        (i,n) = C(m,n,i);
        m = R(i,m);
    }
    return m; }
```

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

```
C(<x,y,beta>,<>,<sigma,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>;}
```

Here m in our pattern stands for the tuple `<x,y,beta>` and `<sigma,n>` correspond to `i` (there is no variable in `n`) .

From `R(<sigma,n>,<x,y,beta>)`, we get our set $\mathcal{R}$. The functions of $\mathcal{R}$ will be indexed by α where $\alpha =$ sigma $\pi/2^n$ where $n \in \{1, 2, \ldots, -\log_2(e) - 1\}$. Finally, we define the h function on m. We take $h($<x,y,beta>$) =$ |beta|.

Now, we need to prove that the six conditions of Section 3.2 are satisfied.

4.2. Proofs of the conditions

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

1. Condition 1 is about the generalized confluence property. The main idea for proving this property is that rotations commute. 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 $f_\alpha(m) \xrightarrow{>} {}^* f_\alpha^k(m) {}^* \xleftarrow{<} f'_\alpha(m)$ by using the rewrite rules f_α and $f_{-\alpha}$ the appropriate number of times.
2. Condition 2 is the termination of the rewriting system. 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. Condition 3 ask for the regularity of the exact semantics. If a is a final form, $|a| \leq e$. The CORDIC algorithm principle is such that if $a = 0$ 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. Condition 4 expresses the stopping condition is reached only when m is in a normal form. In our case, the stopping condition implies $h(m) \leq$ e. Moreover, any rewriting rule changes the value of e by at least $2e$. Hence there is no more rewriting rules that have the decreasing condition: m is a terminal state.
5. Condition 5 expresses that if the control flow was the same the finite precision semantics and the exact semantics would be close to each other. 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 $7\epsilon_e$. The function is also $1 + |\alpha|$-Lipschitz. Hence, R is $P(1 + |\alpha|, 7\epsilon_e)$. 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.
6. Condition 6 require the approximate semantics to stop close to the exact semantics. When the stopping condition is reached, the value of $beta$ in the exact semantics should be less than e, hence as for the proof of Condition 4, $m \in \bar{\mathbb{M}}$.

5. Example: Dijkstra's shortest path algorithm

In this section we apply our method to Dijkstra's shortest path algorithm. When given a labeled 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;}
```

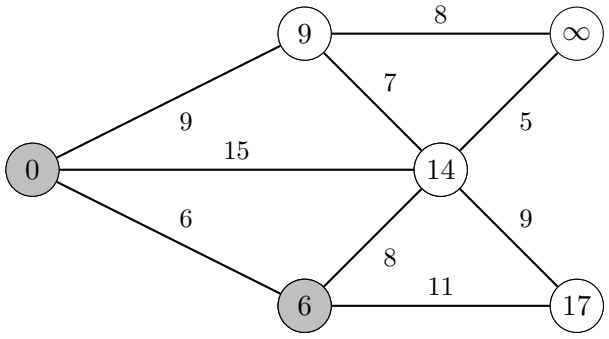

Figure 8: Dijkstra's algorithm after two steps

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

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

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>, <u, i>) {
     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];
            }
        }
}
```

Here m is the tuple `<graph, pathestimate>` which is also the Cartesian product of the actual input and the actual output. The tuple i is `<u,i>` an oriented edge: the node i to potentially update from the value of the node u and the length of the edge `graph[u][i]`. The tuple n contains the auxiliary variables `<count, mark>` used to mark the node which already have there definitive value in `pathestimate` .

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

$$c \xrightarrow[<u,v>]{\geq} e \xrightarrow[<u',v'>]{\geq} f \xleftarrow[<u,v>]{\leq} d \xleftarrow[<u',v'>]{\leq} b$$

   where $x \xrightarrow[<u,v>]{\geq} y$ means that when we apply the rules that updates v from u and the length of the edge (u, v) either $x \xrightarrow{\geq} y$ (the value `pathestimate` for y has been updated with a smaller value) or $x = y$ and (the update rules does not do anything).
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 is a 1-Lipschitz function (the function is hence $P(1, 0)$).
4. When the algorithm stops, the minimal path is computed for each node so h cannot decrease anymore.
5. The function `R` is $P(1, \epsilon_e)$. 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 $P(1, n^2\epsilon_e)$.

6. 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 $l\epsilon_e$.

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 (thus causing 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 analyzing a conditional too narrowly.

In the two papers [1, 2], a robustness analysis is done for 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. As in our theorem, this syntax needs additional conditions (mainly the commutativity for two observable equivalent commands). However, their abstract language does not allow all the programs that we consider here: in particular, CORDIC is not in the scope of the methods used in those papers. As a result, their conditions are simpler and their proofs are more directed than ours. Another difference with these other papers is that they aim to provide the exact semantics and then to treat the computational errors qualitatively, since robust programs are 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.

7. Future work and conclusion

We have presented a definition of robustness for a program and we have presented the property $P(k, \epsilon)$ that expresses that the finite-precision semantics is close to the exact one. Then we have presented several conditions that allow us to

establish this closeness property. The main theorem allows the code to be decomposed into several parts and requires conditions on these subpart. This methodology allows reasoning first on exact semantics and then second on 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-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.

The future for any method that is base on templates usually involves looking for more templates and, additionally, attempting to extend the scope of existing templates. We plan to pursue both of these research directions. Also, we are interested in finding other definitions for robustness that are more suitable for floating-point semantics, e.g., taking into account the fact that rounding is proportional to the size of a floating-point number.

Acknowledgments:. We would like to thank Eric Goubault and Jean Goubault-Larrecq for many useful discussions on the topic of this paper. The anonymous reviewers of an earlier draft of this paper have also given us valuable comments.

8. 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 Navid-Pour. 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. In Herbert Wiklicky and Mieke Massink, editors, *Proceedings of the 10th Workshop on Quantitative Aspects of Programming Languages and Systems (QAPL 2012)*, volume 85 of *Electronic Proceedings in Theoretical Computer Science*, pages 63–76, 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.