DOI:10.1145/2240236.2240262

Continuity and Robustness of Programs

By Swarat Chaudhuri, Sumit Gulwani, and Roberto Lublinerman

Abstract

Computer scientists have long believed that software is different from physical systems in one fundamental way: while the latter have continuous dynamics, the former do not. In this paper, we argue that notions of continuity from mathematical analysis are relevant and interesting even for software. First, we demonstrate that many everyday programs are *continuous* (i.e., arbitrarily small changes to their inputs only cause arbitrarily small changes to their outputs) or *Lipschitz continuous* (i.e., when their inputs change, their outputs change at most proportionally). Second, we give an mostly-automatic framework for verifying that a program is continuous or Lipschitz, showing that traditional, discrete approaches to proving programs correct can be extended to reason about these properties. An immediate application of our analysis is in reasoning about the *robustness* of programs that execute on uncertain inputs. In the longer run, it raises hopes for a toolkit for reasoning about programs that freely combines logical and analytical mathematics.

1. INTRODUCTION

It is accepted wisdom in computer science that the dynamics of software systems are inherently discontinuous, and that this fact makes them fundamentally different from physical systems. More than 25 years ago, Parnas[15] attributed the difficulty of engineering reliable software to the fact that "the mathematical functions that describe the behavior of [software] systems are not continuous." This meant that the traditional analytical calculus—the mathematics of choice when one is analyzing the dynamics of, say, fluids—did not fit the needs of software engineering too well. Logic, which can reason about discontinuous systems, was better suited to being the mathematics of programs.

In this paper, we argue that this wisdom is to be taken with a grain of salt. First, many everyday programs are continuous in the same sense as in analysis—that is, arbitrarily small changes to its inputs lead to arbitrarily small changes to its outputs. Some of them are even *Lipschitz continuous*—that is, perturbations to the program's inputs lead to at most proportional changes to its outputs. Second, we show that analytic properties of programs are not at odds with the classical, logical methods for program verification, giving a logic-based, mostly-automated method for formally verifying that a program is continuous or Lipschitz continuous. Among the immediate applications of this analysis is reasoning about the *robustness* of programs that execute under uncertainty. In the longer run, it raises hopes for a unified theory of program analysis that marries logical and analytical approaches.

Now we elaborate on the above remarks. Perhaps the most basic reason why software systems can violate continuity is *conditional branching*—that is, constructs of the form "**if** B **then** P_1 **else** P_2." Continuous dynamical systems arising in the physical sciences do not typically contain such constructs, but most nontrivial programs do. If a program has a branch, then even the minutest perturbation to its inputs may cause it to evaluate one branch in place of the other. Thus, we could perhaps conclude that any program containing a branch is ipso facto discontinuous.

To see that this conclusion is incorrect, consider the problem of sorting an array of numbers, one of the most basic tasks in computing. Every classic algorithm for sorting contains conditional branches. But let us examine the *specification* of a sorting algorithm: a mathematical function *Sort* that maps arrays to their sorted permutations. This specification is not only continuous but Lipschitz continuous: change any item of an input array A by $\pm\epsilon$, and each item of $Sort(A)$ changes at most by $\pm\epsilon$. For example, suppose A and A' are two input arrays as below, with A' obtained by perturbing each item of A at most by ± 1. Then $Sort(A')$ can be obtained by perturbing each item of $Sort(A)$ at most by ± 1.

$$\begin{array}{rr} A: 1,3,5,3,4 & A': 1,2,4,4,3 \\ Sort(A): 1,3,3,4,5 & Sort(A'): 1,2,3,4,4 \end{array}$$

Similar observations hold for many of the classic computations in computer science, for example, shortest path and minimum spanning tree algorithms. Our program analysis extends and automates methods from the traditional analytical calculus to prove the continuity or Lipschitz continuity of such computations. For instance, to verify that a conditional statement within a program is continuous, we generalize the sort of insight that a high-school student uses to prove the continuity of a piecewise function like

$$\text{abs}(x) = \begin{cases} -x & \text{if} \quad x < 0 \\ x & \text{if} \quad x \geq 0. \end{cases}$$

This paper is based on two previous works: "Continuity Analysis of Programs," by S. Chaudhuri, S. Gulwani, and R. Lublinerman, published in *POPL* (2010), 57–70, and "Proving Programs Robust," by S. Chaudhuri, S. Gulwani, R. Lublinerman, and S. Navidpour, published in *FSE* (2011), 102–112.

Intuitively, abs(x) is continuous because its two "pieces" x and $-x$ are continuous, and because x and $-x$ agree on values in the neighborhood of $x = 0$, the point where a small perturbation can cause abs(x) to switch from evaluating one piece to evaluating the other. Our analysis uses the same idea to prove that "**if** B **then** P_1 **else** P_2" is continuous: it inductively verifies that P_1 and P_2 are continuous, then checks, often automatically, that P_1 and P_2 become semantically equivalent at states where the value of B can flip on a small perturbation.

When operating on a program with loops, our analysis searches for an inductive proof of continuity. To prove that a continuous program is Lipschitz continuous, we inductively compute a collection of *Lipschitz matrices* that contain numerical bounds on the slopes of functions computed along different control paths of the program.

Of course, complete software systems are rarely continuous. However, verification technique like ours allows us to identify modules of a program that satisfy continuity properties. A benefit of this is that such modules are amenable to analysis by continuous methods. In the longer run, we can imagine a reasoning toolkit for programs that combines continuous analysis techniques, for example, numerical optimization or symbolic integration, and logical methods for analyzing code. Such a framework would expand the classical calculus to functions encoded as programs, a representation worthy of first-class treatment in an era where much of applied mathematics is computational.

A more immediate application of our framework is in the analysis of programs that execute on *uncertain* inputs, for example noisy sensor data or inexact scientific measurements. Unfortunately, traditional notions of functional correctness do not guarantee predictable program execution on uncertain inputs: a program may produce the correct output on each individual input, but even small amounts of noise in the input could change its output radically. Under uncertainty, traditional correctness properties must be supplemented by the property of *robustness*, which says that small perturbations to program's inputs do not have much effect on the program's output. Continuity and Lipschitz continuity can both serve as definitions of robustness, and our analysis can be used to prove that a program is robust.

The rest of the paper is organized as follows. In Section 2, we formally define continuity and Lipschitz continuity of programs and give a few examples of computations that satisfy these properties. In Section 3, we give a method for verifying a program's continuity, and then extend it to an analysis for Lipschitz continuity. Related work is presented in Section 4; Section 5 concludes the paper with some discussion.

2. CONTINUITY, LIPSCHITZ CONTINUITY, AND ROBUSTNESS

In this section, we define continuity[2] and Lipschitz continuity[3] of programs and show how they can be used to define robustness. First, however, we fix the programming language Imp whose programs we reason about.

Imp is a "core" language of imperative programs, meaning that it supports only the most central features of imperative programming—assignments, branches, and loops. The language has two discrete data types—integers and arrays of integers—and two continuous data types—reals and arrays of reals. Usual arithmetic and comparisons on these types are supported. In conformance with the model of computation under which algorithms over reals are typically designed, our reals are infinite-precision, and elementary operations on them are assumed to be given by unit-time oracles.

Each data type in Imp is associated with a *metric*.[a] This metric is our notion of distance between values of a given type. For concreteness, we fix, for the rest of the paper, the following metrics for the Imp types:

- The integer and real types are associated with the Euclidean metric $d(x, y) = |x - y|$.
- The metric over arrays (of reals or integers) of the same length is the L_∞*-norm*: $d(A_1, A_2) = \max_i\{|A_1[i] - A_2[i]|\}$. Intuitively, an array changes by ϵ when its size is kept fixed, but one or more of its items change by ϵ. We define $d(A_1, A_2) = \infty$ if A_1 and A_2 have different sizes.

The syntax of arithmetic expressions E, Boolean expressions B, and programs *Prog* is as follows:

$$
\begin{array}{lll}
E & ::= & x \mid A[i] \mid c \mid E + E \mid E \cdot E \\
B & ::= & E > 0 \mid B \wedge B \mid B \vee B \mid \neg B \\
Prog & ::= & \mathbf{skip} \mid x := E \mid A[i] := E \mid Prog; Prog \\
 & & \mathbf{if}\ B\ \mathbf{then}\ Prog\ \mathbf{else}\ Prog \mid \mathbf{while}\ B\ \mathbf{do}\ Prog.
\end{array}
$$

Here x is a typed variable, c is a typed constant, A is an array variable, i an integer variable or constant, $+$ and $\cdot$ respectively represent addition and multiplication over scalars (reals or integers), and the Boolean operators are as usual. We assume an orthogonal type system that ensures that all expressions and assignments in our programs are well-typed. The set of variables appearing in P is denoted by $Var(P) = \{x_1, \ldots, x_n\}$.

As for semantics, for simplicity, let us restrict our focus to programs that terminate on all inputs. Let *Val* be a universe of typed *values*. A *state* of P is a vector $\sigma \in Val^n$. Intuitively, for all $1 \le i \le n$, $\sigma(i)$ is the value of the variable x_i at state σ. The set of all states of P is denoted by $\Sigma(P)$.

The semantics of the program P, an arithmetic expression e occurring in P, and a Boolean expression b in P are now respectively given by maps $[\![P]\!]: \Sigma(P) \to \Sigma(P)$, $[\![e]\!]: \Sigma(P) \to Val$, and $[\![b]\!]: \Sigma(P) \to \{true, false\}$. Intuitively, for each state σ of P, $[\![e]\!](\sigma)$ and $[\![b]\!](\sigma)$ are respectively the values of e and b at σ, and $[\![P]\!](\sigma)$ is the state at which P terminates after starting execution from σ. We omit the inductive definitions of these maps as they are standard.

Our definition of continuity of programs is an adaptation of the traditional ϵ-δ definition of continuous functions. As a program can have multiple inputs and outputs, we define continuity with respect to an *input variable* x_i and an *output*

[a] Recall that a *metric* over a set S is a function $d: S \times S \to \mathbb{R} \cup \{\infty\}$ such that for all x, y, z, we have (1) $d(x, y) \ge 0$, with $d(x, y) = 0$ iff $x = y$; (2) $d(x, y) = d(y, x)$; and (3) $d(x, y) + d(y, z) \ge d(x, z)$.

variable x_j. Intuitively, if P is continuous with respect to input x_i and output x_j, then an arbitrarily small change to the initial value of any x_i, while keeping the remaining variables fixed, must only cause an arbitrarily small change to the final value of x_j. Variables other than x_j are allowed to change arbitrarily.

Formally, consider states σ, σ' of P and any $\epsilon > 0$. Let x_i be a variable of type τ, and let d_τ denote the metric over type τ. We say that σ and σ' are *ϵ-close* with respect to x_i, and write $\sigma \approx_{\epsilon,i} \sigma'$, if $d_\tau(\sigma(i), \sigma'(i)) < \epsilon$. We call σ' an *ϵ-perturbation* of σ with respect to x_i, and write $\sigma \equiv_{\epsilon,i} \sigma'$, if σ and σ' are ϵ-close with respect to x_i, and further, for all $j \neq i$, we have $\sigma(j) = \sigma'(j)$. Now we define:

Definition 1 (Continuity). *A program P is* continuous *at a state σ with respect to an input variable x_i and an output variable x_j if for all $\epsilon > 0$, there exists a $\delta > 0$ such that for all $\sigma' \in \Sigma(P)$,*

$$\sigma \equiv_{\delta,i} \sigma' \Rightarrow [\![P]\!](\sigma) \approx_{\epsilon,j} [\![P]\!](\sigma')$$

An issue with continuity is that it only certifies program behavior under *arbitrarily small* perturbations to the inputs. However, we may often want a definition of continuity that establishes a *quantitative* relationship between changes to a program's inputs and outputs. Of the many properties in function analysis that accomplish this, *Lipschitz continuity* is perhaps the most well known. Let $K > 0$. A function $f : \mathbb{R} \to \mathbb{R}$ is *K-Lipschitz continuous*, or simply *K-Lipschitz*, if a $\pm\epsilon$-change to x can change $f(x)$ by at most $\pm K\epsilon$. The constant K is known as the *Lipschitz constant* of f. It is easy to see that if f is K-Lipschitz for some K, then f is continuous at every input x.

We generalize this definition in two ways while adapting it to programs. First, as for continuity, we define Lipschitz continuity with respect to an input variable x_i and an output variable x_j. Second, we allow Lipschitz constants that depend on the *size of the input*. For example, suppose x_i is an array of length N, and consider ϵ-changes to it that do not change its size. We consider P to be *N-Lipschitz* with respect to x_i if on such a change, the output x_j can change at most by $N \cdot \epsilon$. In general, a Lipschitz constant of P is a function $K : \mathbb{N} \to \mathbb{R}_{\geq 0}$ that takes the size of x_i as an input.

Formally, to each value v, let us associate a *size* $\|v\| > 0$. If v is an integer or real, then $\|v\| = 1$; if v is an array of length N, then $\|v\| = N$. We have:

Definition 2 (Lipschitz continuity). *Let $K : \mathbb{N} \to \mathbb{R}_{\geq 0}$. The program P is K-*Lipschitz *with respect to an input x_i and an output x_j if for all $\sigma, \sigma' \in \Sigma(P)$ and $\epsilon > 0$,*

$$\sigma \equiv_{\epsilon,i} \sigma' \wedge (\|\sigma(i)\| = \|\sigma'(i)\|) \Rightarrow [\![P]\!](\sigma) \approx_{m,j} [\![P]\!](\sigma')$$

where $m = K(\|\sigma(i)\|) \cdot \epsilon$.

More generally, we could define Lipschitz continuity of P within a subset Σ' of its state space. In such a definition, σ and σ' in Definition 2 are constrained to be in Σ', and no assertion is made about the effect of perturbations on states outside Σ'. Such a definition is useful because many realistic programs are Lipschitz only within certain regions of their input space, but for brevity, we do not consider it here.

Now we note that many of the classic computations in computer science are in fact continuous or Lipschitz.

Example 1 (Sorting). *Let Sort be a sorting program that takes in an array A of reals and returns a sorted permutation of A. As discussed in Section 1, Sort is 1-Lipschitz in input A and output A, because any ϵ-change to the initial value of A (defined using our metric on arrays) can produce at most an ϵ-change to its final value. Note that this means that Sort is continuous in input A and output A at every program state.*

What if A was an array of integers? Continuity still holds in this case, but for more trivial reasons. Since A is of a discrete type, the only arbitrarily small *perturbation that can be made to A is no perturbation at all. Obviously, the program output does not change in this case. However, reasoning about continuity turns out to be important even under these terms. This is apparent when we try to prove that Sort is 1-Lipschitz when A is an array of integers. The easiest way to do this is to "cast" the input type of the program into an array of reals and prove that the program is 1-Lipschitz even after this modification, and this demands a proof of continuity.*

Example 2 (Shortest paths). *Let SP be a correct implementation of a shortest path algorithm. We view the graph G on which SP operates as an array of reals such that G[i] is the weight of the i-th edge. An ϵ-change to G thus amounts to a maximum change of $\pm\epsilon$ to any edge weight of G, while keeping the node and edge structure intact.*

The output of SP is the array d of shortest path distances in G—that is, d[i] is the length of the shortest path from the source node to the i-th node u_i of G. We note that SP is N-Lipschitz in input G and output d (N is the number of edges in G). This is because if each edge weight in G changes by an amount ϵ, a shortest path weight can change at most by $N \cdot \epsilon$.

On the other hand, suppose SP had a second output: an array π whose i-th element is a sequence of nodes forming a minimal-weight path between src and u_i. An ϵ-change to G may add or subtract elements from π—that is, perturb π by the amount ∞. Therefore, SP is not K-Lipschitz with respect to the output π for any K.

Example 3 (Minimum spanning trees). *Consider any algorithm MST for computing a minimum-cost spanning tree in a graph G with real edge weights. Suppose MST has a variable c that holds, on termination, the cost of the minimum spanning tree. MST is N-Lipschitz (hence continuous everywhere) in the input G and the output c.*

Example 4 (Knapsack). *Consider the Knapsack problem from combinatorial optimization. We have a set of items $\{1,\ldots, N\}$, each item i associated with a real* cost *$c[i]$ and a real* value *$v[i]$. We also have a nonnegative, real-valued budget. The goal is to identify a subset Used $\subseteq \{1,\ldots, N\}$ such that the constraint $\Sigma_{j \in Used}\, c[i] \leq$ Budget is satisfied, and the value of $tot_v = \Sigma_{j \in Used}\, v[i]$ is maximized.*

Let our output variable be tot_v. As a perturbation can turn a previously feasible solution infeasible, a program Knap solving *this problem is discontinuous with respect to the input c and also with respect to the input Budget. At the same time, Knap is N-Lipschitz with respect to the input v: if the value of each item changes by* $\pm\epsilon$, *the value of* tot_v *can change by* $\pm N\epsilon$.

Continuity and Lipschitz continuity can be used as definitions of *robustness*, a property that ensures that a program behaves predictably on uncertain inputs. Uncertainty, in this context, can be modeled by either nondeterminism (the value of x has a margin of error $\pm\epsilon$) or a probability distribution. For example, a robust embedded controller does not change its control decisions abruptly because of uncertainty in its sensor-derived inputs. The statistics computed by a robust scientific program are not much affected by measurement errors in the input dataset.

Robustness has benefits even in contexts where a program's relationship with uncertainty is not adversarial. The input space of a robust program does not have isolated "peaks"—that is, points where the program output is very different from outputs on close-by inputs. Therefore, we are more likely to cover a program's behaviors using random tests if the program is robust. Also, robust computations are more amenable to randomized and approximate program transformations[20] that explore trade-offs between a program's quality of results and resource consumption. Transformations of this sort can be seen to deliberately introduce uncertainty into a program's operational semantics. If the program is robust, then this extra uncertainty does not significantly affect its observable behavior, hence such a transformation is "safer" to perform. More details are available in our conference paper on robustness analysis.[3]

Now, if a program is Lipschitz, then we can give quantitative upper bounds on the change to its behavior due to uncertainty in its inputs, and further, this bound is small if the inputs are only slightly uncertain. Consequently, Lipschitz continuity is a rather strong robustness property. Continuity is a weaker definition of robustness—a program computing e^x is continuous, even though it hugely amplifies errors in its inputs. Nonetheless, it captures freedom from a common class of robustness violations: those where uncertainty in the inputs alters a program's control flow, and this change leads to a significant change in the program's output.

3. PROGRAM VERIFICATION

Now we present our automated framework for proving a program continuous[2] or Lipschitz.[3] Our analysis is *sound*—that is, a program proven continuous or Lipschitz by our analysis is indeed continuous. However, as the analysis targets a Turing-complete language, it is *incomplete*—for example, a program may be continuous even if the analysis does not deem it so.

3.1. Verifying continuity

First we show how to verify the continuity of an Imp program. We use as running examples three of the most well-known algorithms of computing: Bubble Sort, the Bellman–Ford shortest path algorithm, and Dijkstra's shortest path algorithm (Figure 2). As mentioned in Example 1, a program is always continuous in input variables of discrete types. Therefore, to make the problem more interesting, we assume that the input to Bubble Sort is an array of reals. As before, we model graphs by arrays of reals where each item represents the weight of an edge.

Given a program P, our task is to derive a syntactic *continuity judgment* for P, defined as a term $b \vdash Cont(P, In, Out)$, where b is a Boolean formula over Var, and In and Out are sets of variables of P. Such a judgment is read as "For each $x_i \in In$ and $x_j \in Out$ and each state σ where b is true, P is continuous in input x_i and output x_j at σ." We break down this task into the task of deriving judgments $b \vdash Cont(P', In, Out)$ for programs P' that are syntactic substructures of P. For example, if P is of the form "**if** b **then** P_1 **else** P_2," then we recursively derive continuity judgments for P_1 and P_2.

Continuity judgments are derived using a set of syntactic proof rules—the rules can be converted in a standard way into an automated program analysis that iteratively assigns continuity judgments to subprograms. Figure 1 shows the most important of our rules; for the full set, see the original reference.[2] To understand the syntax of the rules, consider the rule Base. This rule derives a *conclusion* $b \vdash Cont(P, In, Out)$, where b, In, and Out are arbitrary, from the *premise* that P is either "**skip**" or an assignment.

The rule Sequence addresses sequential composition of programs, generalizing the fact that the composition of two continuous functions is continuous. One of the premises of this rule is a *Hoare triple* of the form $\{b_1\}P\{b_2\}$. This is to be read as "For any state σ that satisfies b_1, $[\![P]\!](\sigma)$ satisfies b_2. (A standard program verifier can be used to verify this premise.) The rule In-Out allows us to restrict or generalize the set of input and output variables with respect to which a continuity judgment is made.

The next rule—Ite—handles conditional statements, and is perhaps the most interesting of our rules. In a conditional branch, a small perturbation to the input variables can cause control to flow along a different branch, leading to a syntactically divergent behavior. For instance, this happens in Lines 3–4 in the Bubble Sort algorithm in Figure 2—perturbations to items in A can lead to either behaviors

Figure 1. Key rules in continuity analysis.

$$\text{(Base)}\ \frac{P \text{ is } \mathbf{skip} \text{ or } x := e}{b \vdash Cont(P, In, Out)}$$

$$\text{(Sequence)}\ \frac{\begin{array}{ll} b_1 \vdash Cont(P_1, In_1, Out_1) & In_2 \subseteq Out_1 \\ b_2 \vdash Cont(P_2, In_2, Out_2) & \{b_1\}P_1\{b_2\} \end{array}}{b_1 \vdash Cont(P_1; P_2, In_1, Out_2)}$$

$$\text{(In-Out)}\ \frac{\begin{array}{ll} b \vdash Cont(P, In, Out) & b' \Rightarrow b \\ Out' \subseteq Out \quad In' \subseteq In & Var(P) \cap V = \emptyset \end{array}}{b' \vdash Cont(P, In' \cup V, Out' \cup V)}$$

$$\text{(Ite)}\ \frac{\begin{array}{c} c \vdash Cont(P_1, In, Out) \quad c \vdash Cont(P_2, In, Out) \\ c \wedge \mathcal{B}(b) \vdash (P_1 \equiv_{Out} P_2) \end{array}}{c \vdash Cont(\mathbf{if}\ b\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2, In, Out)}$$

$$\text{(Loop)}\ \frac{\begin{array}{lll} P \equiv \mathbf{while}\ b\ \mathbf{do}\ R & \{c \wedge b\}R\{c\} & c \vdash Cont(R, X, X) \\ c \vdash Sep(P, X) & & \mathcal{B}(b) \wedge c \vdash (R \equiv_X \mathbf{skip}) \end{array}}{c \vdash Cont(P, X, X)}$$

Figure 2. Bubble sort, the Bellman-Ford algorithm, and Dijkstra's algorithm.

```
BubbleSort (A : array of reals)
1  for j := 1 to (|A| − 1);
2      do for i := 1 to (|A| − 1);
3          do if (A[i] > A[i + 1])
4              then t := A[i]; A[i] := A[i + 1]; A[i + 1] := t;

BellmanFord(G : array of reals, src : int)
1  ...
2  for i := 1 to (|G| − 1)
3      do for each edge (v, w) of G
4          do if d[v] + G(v, w) < d[w]
5              then d[w] := d[v] + G(v, w)

Dijkstra(G : array of reals, src : int)
1  ...
2  while W ≠ ∅
3      do choose edge (v, w) ∈ W such that d[w] is minimal;
4          remove (v, w) from W;
5          if d[w] + G[w, v] < d[v]
6              then d[v] := d[w] + G[w, v]
```

of either "swapping $A[i]$ and $A[i + 1]$" or "leaving A unchanged."

The core idea behind the rule Ite is to show that such a divergence does not really matter, because at the program states where arbitrarily small perturbations to the program variables can "flip" the value of the guard b of the conditional statement (let us call the set of such states the *boundary* of b), the branches of the conditional are arbitrarily close in behavior.

Precisely, let us construct from b the following formula:

$$\mathcal{B}(b) = \begin{cases} \mathit{false} & \text{if } b \text{ only uses discrete variables} \\ e = 0 & \text{if } b \text{ equals } (e > 0) \text{ and } e \text{ uses at least one variable of a continuous data type} \\ \mathcal{B}(b_1) \vee \mathcal{B}(b_2) & \text{if } b \text{ equals } (b_1 \vee b_2) \text{ or } (b_1 \wedge b_2) \\ \mathcal{B}(b') & \text{if } b \text{ equals } \neg b' \end{cases}$$

Note that $\mathcal{B}(b)$ represents an overapproximation of the boundary of b. Also, for a set of output variables *Out* and a Boolean formula c, let us call programs P_1 and P_2 *Out-equivalent under c*, and write $c \vdash (P_1 \equiv_{Out} P_2)$, if for each state σ that satisfies c, the states $[\![P_1]\!](\sigma)$ and $[\![P_2]\!](\sigma)$ agree on the values of all variables in *Out*. We assume an oracle that can determine if $c \vdash (P_1 \equiv_{Out} P_2)$ for given c, P_1, P_2, and *Out*. In practice, such equivalence questions can often be solved fully automatically using modern automatic theorem provers.[19] Now, to derive a continuity judgment for a program "**if** b **then** P_1 **else** P_2" with respect to the outputs *Out*, Ite shows that P_1 and P_2 become *Out*-equivalent under the condition $\mathcal{B}(b)$.

The rule Loop derives continuity judgments for **while**-loops. The idea here is to prove the body R of the loop continuous, then inductively argue that the entire loop is continuous too. In more detail, the rule derives a continuity judgment $c \vdash \mathit{Cont}(R, X, X)$, where c is a *loop invariant*—a property that is always true at the loop header—and X is a set of variables. Now consider any state σ satisfying c. An arbitrarily small perturbation to this state leads to an arbitrarily small change in the value of each variable in X at the end of the first iteration, which only leads to an arbitrarily small change in the value of each variable in X at the end of the second iteration, and so on. Continuity follows.

Some subtleties need to be considered, however. An execution from a slightly perturbed state may terminate earlier or later than it would in the original execution. Even if the loop body is continuous, the extra iterations in either the modified or the original execution may cause the states at the loop exit to be very different. We rule out such scenarios by asserting a premise called *synchronized termination*. A loop "**while** b **do** R" fulfills this property with respect to a loop invariant c and a set of variables X, if $\mathcal{B}(b) \wedge c \vdash R \equiv_X$ **skip**). Under this property, even if the loop reaches a state where a small perturbation can cause the loop to terminate earlier (similarly, later), the extra iterations in the original execution have no effect on the program state. We can ignore these iterations in our proof.

Second, even if the loop body is continuous in input x_i and output x_j for every $x_i, x_j \in X$, an iteration may drastically change the value of program variables not in X. If there is a data flow from these variables to some variable in X, continuity will not hold. We rule out this scenario through an extra condition. Consider executions of P whose initial states satisfy a condition c. We call a set of variables X of P *separable* under c if the value of each $z \in X$ on termination of any such execution is independent of the initial values of variables not in X. We denote the fact that X is separable in this way by $c \vdash \mathit{Sep}(P, X)$.

To verify that P is continuous in input x_i and output x_j at state σ, we derive a judgment $b \vdash \mathit{Cont}(P, \{x_i\}, \{x_j\})$, where b is true at σ. The correctness of the method follows from the following *soundness theorem*:

Theorem 1. *If the rules in Figure 1 can derive the judgment* $b \vdash \mathit{Cont}(P, \mathit{In}, \mathit{Out})$, *then for all* $x_i \in \mathit{In}$, $x_j \in \mathit{Out}$, *and* σ *such that* $[\![b]\!] = \mathit{true}$, *P is continuous in input* x_i *and output* x_j *at* σ.

Example 5 (Warmup). *Consider the program* "**if** $(x > 2)$ **then** $x := x/2$ **else** $x := -5x + 11$." $\mathcal{B}(x > 2)$ *equals* $(x = 2)$ *and* $(x = 2) \vdash (x := x/2) \equiv_{\{x\}} (x := -5x + 11)$. *By* Ite, *the program is continuous in input x and output x.*

Let us now use our rules on the algorithms in Figure 2.

Example 6 (Bubble Sort). *Consider the implementation of Bubble Sort in Figure 2. (We assume it to be rewritten as a* **while**-*program in the obvious way.) Our goal is to derive the judgment* $\mathit{true} \vdash \mathit{Cont}(\mathit{BubbleSort}, \{A\}, \{A\})$.

Let $X = \{A, i, j\}$, *and let us write* $R_{\langle p,q \rangle}$ *to denote the code fragment from line p to line q (both inclusive). Also, let us write* $c \vdash \mathit{Term}(\textbf{while}\ b\ \textbf{do}\ R, X)$ *as an abbreviation for* $\mathcal{B}(b) \wedge c \vdash (R \equiv_X \textbf{skip})$.

It is easy to show that true $\vdash$ *Sep*(*BubbleSort*, *X*) *and true* $\vdash$ *Sep*($R_{\langle 2,4\rangle}$, *X*). *Each loop guard only involves discrete variables, hence we derive true* $\vdash$ *Term*(*BubbleSort, X*) *and true* $\vdash$ *Term*($R_{\langle 2,4\rangle}$, *X*).

Now consider $R_{\langle 3,4\rangle}$. *As* $\mathcal{B}(A[i] > A[i+1])$ *equals* $(A[i] = A[i+1])$ *and* $(A[i] = A[i+1]) \vdash (\textbf{skip} \equiv_X R_{\langle 4,4\rangle})$, *we have true* $\vdash$ *Cont*($R_{\langle 3,4\rangle}$, *X*, *X*), *then true* $\vdash$ *Cont*($R_{\langle 2,4\rangle}$, *X*, *X*), *and then true* $\vdash$ *Cont*(*BubbleSort*, *X*, *X*). *Now the* In-Out *rule derives the judgment we are after.*

Example 7 (Bellman–Ford). *Take the Bellman–Ford algorithm. On termination, d*[*u*] *contains the shortest path distance from the source node src to the node u. We want to prove that true* $\vdash$ *Cont*(*BellmanFord*, $\{G\}$, $\{d\}$).

We use the symbols $R_{\langle p,q\rangle}$ *and Term as before. Clearly, we have true* $\vdash$ *Sep*($R_{\langle 3,5\rangle}$, *X*) *and true* $\vdash$ *Term*($R_{\langle 3,5\rangle}$, *X*), *where* $X = \{G, v, w\}$. *The two branches of the conditional in Line 4 are X-equivalent at* $\mathcal{B}(d[v] + G(v, w) < d[w])$, *hence we have true* $= \vdash$ *Cont*($R_{\langle 4,5\rangle}$, *X*, *X*), *and from this judgment, true* $\vdash$ *Cont*($R_{\langle 3,5\rangle}$, *X*, *X*). *Similar arguments can be made about the outer loop. Now we can derive the fact true* $\vdash$ *Cont*(*BellmanFord, X, X*); *weakening, we get the judgment we seek.*

Unfortunately, the rule Loop is not always enough for continuity proofs. Consider states σ and σ' of a continuous program *P*, where σ' is obtained by slightly perturbing σ. For Loop to apply, executions from σ and σ' must converge to close-by states at the end of each loop iteration. However, this need not be so. For example, think of Dijkstra's algorithm. As a shortest path computation, this program is continuous in the input graph *G* and the output *d*—the array of shortest path distances. But let us look at its main loop in Figure 2.

Note that in any iteration, there may be several items *w* for which *d*[*w*] is minimal. But then, a slightly perturbed initial value of *d* may cause a loop iteration to choose a different *w*, leading to a drastic change in the value of *d* at the end of the iteration. Thus, individual iterations of this loop are not continuous, and we cannot apply Loop.

In prior work,[2] we gave a more powerful rule, called *epoch induction*, for proving the continuity of programs like the one above. The key insight here is that if we group some loop iterations together, then continuity becomes an inductive property of the groupings. For example, in Dijkstra's algorithm, a "grouping" is a maximal set *S* of successive loop iterations that are tied on the initial value of *d*[*w*]. Let σ_0 be the program state before the first iteration in *S* is executed. Owing to arbitrarily small perturbations to σ_0, we may execute iterations in *S* in a very different order. However, an iteration that ran after the iterations in *S* in the original execution will still run after the iterations in *S*. Moreover, for a fixed σ_0, the program state, once all iterations in *S* have been executed, is the same, no matter what order these iterations were executed in. Thus, a small perturbation cannot significantly change the state at the end of *S*, and the set of iterations *S* forms a continuous computation.

We have implemented the rules in Figure 1, as well as the epoch induction rule, in the form of a mostly-automatic program analysis. Given a program, the analysis iterates through its control points, assigning continuity judgments to subprograms until convergence is reached. Auxiliary tasks such as checking the equivalence of two straight-line program fragments (needed by rule Ite) are performed automatically using the Z3[8] SMT-solver. Human intervention is expected in two forms. First, in applications of the epoch induction rule, we sometimes expect the programmer to write annotations that define appropriate "groupings" of iterations. Second, in case a complex loop invariant is needed for the proof (e.g., when one of the programs in a program equivalence query is a nested loop), the programmer is expected to supply it. There are heuristics and auxiliary tools that can be used to automate these steps, but our current system does not employ them.

Given the incompleteness of our proof rules, a natural empirical question for us was whether our system can verify the continuity of the continuous computing tasks described in Section 2. To answer this question, we chose several 13 continuous algorithms (including algorithms) over real and real array data types. Our system was able to verify the continuity of 11 of these algorithms, including the shortest path algorithms of Bellman-Ford, Dijkstra, and Floyd-Warshall; Merge Sort and Selection Sort in addition to Bubble Sort; and the minimum spanning tree algorithms of Prim and Kruskal. Among the algorithms we could not verify were Quick Sort. Please see Chaudhuri et al.[2] for more details.

3.2. Verifying Lipschitz continuity

Now we extend the above verification framework to one for Lipschitz continuity. Let us fix variables x_i and x_j of the program *P* respectively as the input and the output variable. To start with, we assume that x_i and x_j are of continuous data types—reals or arrays of reals.

Let us define a *control flow path* of a program *P* as the sequence of assignment or **skip**-statements that *P* executes on some input (we omit a more formal definition). We note that since our arithmetic expressions are built from additions and multiplications, each control flow path of *P* encodes a continuous—in fact differentiable—function of the inputs. Now suppose we can show that each control flow path of *P* is a *K*-Lipschitz computation, for some *K*, in input x_i and output x_j. This does not mean that *P* is *K*-Lipschitz in this input and output: a perturbation to the initial value of x_i can cause *P* to execute along a different control flow path, leading to a drastically different final state. However, if *P* is continuous *and* the above condition holds, then *P* is *K*-Lipschitz in input x_i and output x_j.

Our analysis exploits the above observation. To prove that *P* is *K*-Lipschitz in input x_i and output x_j, we establish that (1) *P* is continuous at all states in input x_i and output x_j and (2) each control flow path of *P* is *K*-Lipschitz in input x_i and output x_j. Of these, the first task is accomplished using the analysis from Section 3.1. To accomplish the second task, we compute a data structure—a set of *Lipschitz matrices*—that contains upper bounds on the slopes of any computation that can be carried out in a control flow path of *P*.

More precisely, let P have n variables named $x_1,\ldots, x_n$, as before. A *Lipschitz matrix* J of P is an $n \times n$ matrix, each of whose elements is a function $K : \mathbb{N} \to \mathbb{R}_{\geq 0}$. Elements of J are represented either as numeric constants or as symbolic expressions (for example, $N + 5$), and the element in the i-th row and j-th column of J is denoted by $J(i, j)$. Our analysis associates P with sets $\mathcal{J}$ of such matrices via judgments P: $\mathcal{J}$. Such a judgment is read as follows: "For each control flow path C in P and each x_i, x_j, there is a $J \in \mathcal{J}$ such that C is $J(j, i)$-Lipschitz in input x_i and output x_j."

The Lipschitz matrix data structure can be seen as a generalization of the *Jacobian* from vector calculus. Recall that the Jacobian of a function $f : \mathbb{R}^n \to \mathbb{R}^n$ with inputs $x_1,\ldots, x_n \in \mathbb{R}$ and outputs $x'_1,\ldots, x'_n \in \mathbb{R}$ is the matrix whose (i, j)-th entry is $\frac{\partial x'_i}{\partial x_j}$. If f is differentiable, then for each x'_i and x_j, f is K-Lipschitz with respect to input x_j and output x'_i, where K is any upper bound on $\left|\frac{\partial x'_i}{\partial x_j}\right|$. In our setting, each control flow path represents a differentiable function, and we can verify the Lipschitz continuity of this function by propagating a Jacobian along the path. On the other hand, owing to branches, the program P may not represent a differentiable, or even continuous, function.

However, note that it is correct to associate a conditional statement "**if** b **then** P_1 **else** P_2" with the set of matrices $(\mathcal{J}_1 \cup \mathcal{J}_2)$, where the judgments $P_1 : \mathcal{J}_1$ and $P_2 : \mathcal{J}_2$ have been made inductively. Of course, this increases the number of matrices that we have to track for a subprogram. But the proliferation of such matrices can be curtailed using an approximation that *merges* two or more of them.

This merge operation $\sqcup$ is defined as $(J_1 \sqcup J_2)(i,j) = \max(J_1(i,j), J_2(i,j))$ for all J_1, J_2, i, j. Suppose we can correctly derive the judgment $P : \mathcal{J}$. Then for any $J_1, J_2 \in \mathcal{J}$, it is also correct to derive the judgment $P : (\mathcal{J} \setminus \{J_1, J_2\} \cup \{J_1 \sqcup J_2\})$. Note that this overapproximation may *overestimate* the Lipschitz constants for some of the control flow paths in P, but this is acceptable as we are not seeking the most precise Lipschitz constant for P anyway.

Figure 3 shows our rules for associating a set $\mathcal{J}$ of Lipschitz matrices with a program P. In the first rule Skip, $\mathbf{I}$ is the identity matrix. The rule is correct because **skip** is 1-Lipschitz in input x_i and output x_i for all i, and 0-Lipschitz in input x_i and output x_j, where $i \neq j$.

To obtain Lipschitz constants for assignments to variables (rule Assign), we must quantify the way the value of an arithmetic expression e changes when its inputs are changed. This is done by computing a vector ∇_e whose i-th element is an upper bound on $\left|\frac{\partial [\![e]\!]}{\partial x_i}\right|$. In more detail, we have

$$\nabla_e(i) = \begin{cases} 0 & \text{if } e \text{ is a constant} \\ 1 & \text{if } e \text{ is } x_i \text{ or } x_i[k], \text{ for some } k \\ 0 & \text{if } e \text{ is } x_j \text{ or } x_j[k], \text{ for } j \neq i \text{ and some } k \\ \nabla_{e_1}(i) + \nabla_{e_2}(i) & \text{if } e \text{ is } (e_1 + e_2) \\ \nabla_{e_1}(i) \cdot |e_2| + \nabla_{e_2}(i) \cdot |e_1| & \text{if } e \text{ is } (e_1 \cdot e_2) \text{ and one of } e_1 \text{ and } e_2 \text{ is a constant} \\ \infty & \textit{otherwise} \end{cases}$$

Assignments $x_i[m] := e$ to array locations are slightly trickier. The location $x_i[m]$ is affected by changes to variables appearing in e; if the variable x_i does not appear in e, then perturbations to the initial value of x_i have no effect on $x_i[m]$. However, the *remaining locations* in x_i are affected by, and only by, changes to the initial value of x_i. Thus, we can view x_i as being split into two "regions"—one consisting of $x_i[m]$ and the other of every other location—with possibly different Lipschitz constants. We track these constants using two different Lipschitz matrices J and J'. Here J is as in the rule Assign, while J' is identical to the Lipschitz matrix for a hypothetical assignment $x_i := x_i$.

Figure 3. Rules for deriving Lipschitz matrices.

$$(\text{Skip})\frac{}{\textbf{skip} : \{\mathbf{I}\}}$$

$$(\text{Assign})\frac{}{(x_i := e) : \{J\}}$$

$$\text{Where } J(j,k) = \begin{cases} \nabla_e(k) & \text{if } j = i \\ 1 & \text{if } j = k \neq i \\ 0 & \text{otherwise} \end{cases}$$

$$(\text{Array-Assign})\frac{}{(x_i[m] := e) : \{J, \mathbf{I}\}} \; J \text{ is as in Assign}$$

$$(\text{Weaken})\frac{P : \mathcal{J} \qquad J_1, J_2 \in \mathcal{J}}{P : \mathcal{J} \setminus \{J_1, J_2\} \cup \{J_1 \sqcup J_2\}}$$

$$(\text{Sequence})\frac{P_1 : \mathcal{J}_1 \qquad P_2 : \mathcal{J}_2}{(P_1 ; P_2) : \{(J_2 J_1) : J_1 \in \mathcal{J}_1, J_2 \in \mathcal{J}_2\}}$$

$$(\text{Ite})\frac{P_1 : \mathcal{J}_1 \qquad P_2 : \mathcal{J}_2}{\textbf{if } b \textbf{ then } P_1 \textbf{ else } P_2 : \mathcal{J}_1 \cup \mathcal{J}_2}$$

$$(\text{While})\frac{\begin{array}{c} P = \textbf{while } b \textbf{ do } R \quad R : \mathcal{J} \quad \textit{Bound}^+(P, M) \\ \forall J \in \mathcal{J}, \forall i, j . (J(i,j) \geq 1 \vee J(i,j) = 0) \end{array}}{P : \mathcal{J}^M}$$

Sequential composition is handled by matrix multiplication (rule Sequence)—the insight here is essentially the chain rule of differentiation. As mentioned earlier, the rule for conditional statements merges the Lipschitz matrices computed along the two branches. The Weaken rule allows us to overestimate a Lipschitz constant at any point.

The rule While derives Lipschitz matrices for while-loops. Here $\textit{Bound}^+(P, M)$ is a premise that states that the symbolic or numeric constant M is an upper bound on the number of iterations of P—it is assumed to be inferred via an auxiliary checker.[11] Finally, $\mathcal{J}^M$ is shorthand for the singleton set of matrix products $\{J_1 \ldots J_M : J_i \in \mathcal{J}\}$. In cases where M is a symbolic bound, we will not be able to compute this product explicitly. However, in many practical cases, one can reduce it to a simpler manageable form using algebraic identities.

The While rule combines the rules for **if**-statements and sequential composition. Consider a loop P whose body R has Lipschitz matrix J. If the loop terminates in exactly M iterations, J^M is a correct Lipschitz matrix for it. However, if the loop may terminate after $M' < M$ iterations, we require an extra property for J^M to be a correct Lipschitz matrix: $J^i \leq J^{i+1}$ for all $i < M$. This property is ensured by the condition

$\forall i, j$: $J(i, j) = 0 \vee J(i, j) \geq 1$. Note that in the course of a proof, we can weaken *any* Lipschitz matrix for the loop body to a matrix J of this form.

We can prove the following soundness theorem:

THEOREM 2. *Let P be continuous in input x_i and output x_j. If the rules in Figure 3 derive the judgment $P : \{J\}$, then P is $J(j, i)$-Lipschitz in input x_i and output x_j.*

EXAMPLE 8 (WARMUP). *Recall the program "**if** $(x > 2)$ **then** $x := x/2$ **else** $x := -5x + 11$" from Example 5 (x is a real). Our rules can associate the left branch with a single Lipschitz matrix containing a single entry $\frac{1}{2}$, and the right branch with a single matrix containing a single entry 5. Given the continuity of the program, we conclude that the program is 5-Lipschitz in input x and output x.*

EXAMPLE 9 (BUBBLE SORT). *Consider the Bubble Sort algorithm (Figure 2) once again, and as before, let $R_{\langle p,q\rangle}$ denote the code fragment from line p to line q. Let us set x_0 to be A and x_1 to be t.*

Now, let $J = \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix}$. From the rules in Figure 3, we can derive $(t := A[i]) : \{J\}$, $(A[i] := A[i+1]) : \{\mathbf{I}\}$, and $(A[i+1] := t) : \{J, \mathbf{I}\}$.

Carrying out the requisite matrix multiplications, we get $R_{\langle 4,4\rangle} : \{J\}$. Using the rule ITE, *we have $R_{\langle 3,4\rangle} : \{\mathbf{I}, J\}$. Now, it is easy to show that $R_{\langle 3,4\rangle}$ gets executed N times, where N is the size of A. From this we have $R_{\langle 2,4\rangle} : \{\mathbf{I}, J\}^N$. Given that $J^2 = \mathbf{I}J = J\mathbf{I} = J$, this is equivalent to the judgment $R_{\langle 2,4\rangle} : \{\mathbf{I}, J\}$. From this, we derive BubbleSort $: \{J, \mathbf{I}\}$. Given the proof of continuity carried out in Example 1, Bubble Sort is 1-Lipschitz in input A and output A.*

Intuitively, the reason why Bubble Sort is so robust is that here, (1) there is no data flow from program points where arithmetic operations are carried out to points where values are assigned to the output variable and (2) continuity holds everywhere. In fact, one can formally prove that any program that meets the above two criteria is 1-Lipschitz. However, we do not develop this argument here.

EXAMPLE 10 (BELLMAN–FORD; DIJKSTRA). *Let us consider the Bellman–Ford algorithm (Figure 2) once again, and let x_0 be G and x_1 be d. Consider line 5 (i.e., the program $R_{\langle 5,5\rangle}$); our rules can assign to this program the Lipschitz matrix J, where $J = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$. With a few more derivations, we obtain $R_{\langle 4,5\rangle} : \{J\}$. Using the rule for loops, we have $R_{\langle 3,5\rangle} : \{J^N\}$, where N is the number of edges in G, and then BellmanFord $: \{J^{N^2}\}$. But note that*

$$J^{N^2} = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}^{N^2} = \begin{pmatrix} 1 & 0 \\ N^2 & 1 \end{pmatrix}.$$

Combing the above with the continuity proof in Example 7, we decide that the Bellman–Ford algorithm is N^2-Lipschitz in input G and output d.

Note that the Lipschitz constant obtained in the above proof is not the optimal one—that would be N. This is an instance of the gap between truth and provability that is the norm in program analysis. Interestingly, our rules can derive the optimal Lipschitz constant for Dijkstra's algorithm. Using the same reasoning as above, we assign to the main loop of the algorithm the single Lipschitz matrix J. Applying the LOOP *rule, we derive*

$$\textit{Dijkstra} : \left\{ \begin{pmatrix} 1 & 0 \\ N & 1 \end{pmatrix} \right\}.$$

Given that the algorithm is continuous in input G and output d, it is N-Lipschitz in input G and output d.

Let us now briefly consider the case when the input and output variables in our program are of discrete type. As a program is trivially continuous in every discrete input, continuity is not a meaningful notion in such a setting. Therefore, we focus on the problem of verifying Lipschitz continuity—for example, showing that the Bubble Sort algorithm is 1-Lipschitz even when the input array A is an array of integers.

An easy solution to this problem is to *cast* the array A into an array A_* of reals, and then to prove 1-Lipschitz continuity of the resultant program in input A_* and output A_*. As any integer is also a real, the above implies that the original algorithm is 1-Lipschitz in input A and output A. Thus, reals are used here as an *abstraction* of integers, just as (unbounded) integers are often used in program verification as abstractions of bounded-length machine numbers.

Unsurprisingly, this strategy does not always work. Consider the program "**if** $(x > 0)$ **then** $x := x + 1$ **else skip**," where x is an integer. This program is 2-Lipschitz. Its "slope" is the highest around initial states where $x = 0$: if the initial value of x changes from 0 to 1, the final value of x changes from 0 to 2. At the same time, if we cast x into a real, the resultant program is discontinuous and thus not K-Lipschitz for any K.

It is possible to give an analysis of Lipschitz continuity that does not suffer from the above issue. This analysis casts the integers into reals as mentioned above, then calculates a Lipschitz matrix of the resultant program; however, it checks a property that is slightly weaker than continuity. For lack of space, we do not go into the details of the analysis here.

We have extended our implementation of continuity analysis with the verification method for Lipschitz continuity presented above, and applied the resulting system to the suite of 13 algorithms mentioned at the end of Section 3.1. All these algorithms were either 1-Lipschitz or N-Lipschitz. Our system was able to compute the optimal Lipschitz constant for 9 of the 11 algorithms where continuity could be verified. In one case (Bellman-Ford), it certified an N-Lipschitz computation as N^2-Lipschitz. The one example on which it fared poorly was the Floyd-Warshall shortest path algorithm, where the best Lipschitz constant that it could compute was exponential in N^3.

4. RELATED WORK

So far as we know, we were the first[2] to propose a framework for continuity analysis of programs. Before us, Hamlet[12] advocated notions of continuity of software; however, he concluded that "it is not possible in practice to mechanically test for continuity" in the presence of loops. Soon after our first paper on this topic (and before our subsequent work on Lipschitz continuity of programs), Reed and Pierce[18] gave a type system that can verify the Lipschitz continuity of functional programs. This system can seamlessly handle functional data structures such as lists and maps; however, unlike our method, it cannot reason about discontinuous control flow, and would consider any program with a conditional branch to have a Lipschitz constant of ∞.

More recently, Jha and Raskhodnikova have taken a *property testing* approach to estimating the Lipschitz constant of a program. Given a program, this method determines, with a level of probabilistic certainty, whether it is either 1-Lipschitz or ϵ-far (defined in a suitable way) from being 1-Lipschitz. While the class of programs allowed by the method is significantly more restricted than what is investigated here or by Reed and Pierce [13], the appeal of the method lies in its crisp completeness guarantees, and also in that it only requires blackbox access to the program.

Robustness is a standard correctness property in control theory,[16, 17] and there is an entire subfield of control studying the design and analysis of robust controllers. However, the systems studied by this literature are abstractly defined using differential equations and hybrid automata rather than programs. The systematic modeling and analysis of robustness of *programs* was first proposed by us in the context of general software, and by Majumdar and Saha[14] in the context of control software.

In addition, there are many efforts in the abstract interpretation literature that, while not verifying continuity or robustness explicitly, reason about the uncertainty in a program's behavior due to floating-point rounding and sensor errors.[6, 7, 10] Other related literature includes work on automatic differentiation (AD),[1] where the goal is to transform a program P into a program that returns the derivative of P where it exists. Unlike the work described here, AD does not attempt verification—no attempt is made to certify a program as differentiable or Lipschitz.

5. CONCLUSION

In this paper, we have argued for the adoption of analytical properties like continuity and Lipschitz continuity as correctness properties of programs. These properties are relevant as they can serve as useful definitions of *robustness* of programs to uncertainty. Also, they raise some fascinating technical issues. Perhaps counterintuitively, some of the classic algorithms of computer science satisfy continuity or Lipschitz continuity, and the problem of systematic reasoning about these properties demands a nontrivial combination of analytical and logical insights.

We believe that the work described here is a first step toward an extension of the classical calculus to a symbolic mathematics where programs form a first-class representation of functions and dynamical systems. From a practical perspective, this is important as physical systems are increasingly controlled by software, and as even applied mathematicians increasingly reason about functions that are not written in the mathematical notation of textbooks, but as *code*. Speaking more philosophically, the classical calculus focuses on the computational aspects of real analysis, and the notation of calculus texts has evolved primarily to facilitate symbolic computation by hand. However, in our era, most mathematical computations are carried out by computers, and a calculus for our age should not ignore the notation that computers can process most easily: programs. This statement has serious implications—it opens the door not only to the study of continuity or derivatives but also to, say, Fourier transforms, differential equations, and mathematical optimization of code. Some efforts in these directions[4, 5, 9] are already under way; others will no doubt appear in the years to come.

Acknowledgments

This research was supported by NSF CAREER Award #1156059 ("Robustness Analysis of Uncertain Programs: Theory, Algorithms, and Tools").

References

1. Bucker, M., Corliss, G., Hovland, P., Naumann, U., Norris, B. Automatic Differentiation: Applications, Theory and Implementations, Birkhauser, 2006.
2. Chaudhuri, S., Gulwani, S., Lublinerman, R. Continuity analysis of programs. In *POPL* (2010), 57–70.
3. Chaudhuri, S., Gulwani, S., Lublinerman, R., Navidpour, S. Proving programs robust. In *FSE* (2011), 102–112.
4. Chaudhuri, S., Solar-Lezama, A. Smooth interpretation. In *PLDI* (2010), 279–291.
5. Chaudhuri, S., Solar-Lezama, A. Smoothing a program soundly and robustly. In *CAV* (2011), 277–292.
6. Chen, L., Miné, A., Wang, J., Cousot, P. Interval polyhedra: An abstract domain to infer interval linear relationships. In *SAS* (2009), 309–325.
7. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X. The ASTREÉ analyzer. In *ESOP* (2005), 21–30.
8. de Moura, L. M. Bjørner, N. Z3: An effcient smt solver. In *TACAS* (2008), 337–340.
9. Girard, A., Pappas, G. Approximate bisimulation: A bridge between computer science and control theory. *Eur. J. Contr. 17,* 5 (2011), 568.
10. Goubault, E. Static analyses of the precision of floating-point operations. In *SAS* (2001).
11. Gulwani, S., Zuleger, F. The reachability-bound problem. In *PLDI* (2010), 292–304.
12. Hamlet, D. Continuity in software systems. In *ISSTA* (2002).
13. Jha, M., Raskhodnikova, S. Testing and reconstruction of lipschitz functions with applications to data privacy. In *FOCS* (2011), 433–442.
14. Majumdar, R., Saha, I. Symbolic robustness analysis. In *RTSS* (2009), 355–363.
15. Parnas, D. Software aspects of strategic defense systems. *Commun. ACM 28,* 12 (1985), 1326–1335.
16. Pettersson, S., Lennartson, B. Stability and robustness for hybrid systems. In *Decision and Control* (Dec 1996), 1202–1207.
17. Podelski, A., Wagner, S. Model checking of hybrid systems: From reachability towards stability. In *HSCC* (2006), 507–521.
18. Reed, J., Pierce, B. Distance makes the types grow stronger: A calculus for differential privacy. In *ICFP* (2010).
19. Strichman, O. Regression verification: Proving the equivalence of similar programs. In *CAV* (2009).
20. Zhu, Z., Misailovic, S., Kelner, J., Rinard, M. Randomized accuracy-aware program transformations for efficient approximate computations. In *POPL* (2012).

Swarat Chaudhuri (swarat@rice.edu), Department of Computer Science, Rice University, Houston, TX.

Sumit Gulwani (sumitg@microsoft.com), Microsoft Research, Redmond, WA.

Roberto Lublinerman (rluble@psu.edu), Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA.