Continuity Analysis of Programs

Swarat Chaudhuri
Pennsylvania State University
swarat@cse.psu.edu

Sumit Gulwani
Microsoft Research
sumitg@microsoft.com

Roberto Lublinerman
Pennsylvania State University
rluble@psu.edu

Abstract

We present an analysis to automatically determine if a program represents a continuous function, or equivalently, if infinitesimal changes to its inputs can only cause infinitesimal changes to its outputs. The analysis can be used to verify the *robustness* of programs whose inputs can have small amounts of error and uncertainty—e.g., embedded controllers processing slightly unreliable sensor data, or handheld devices using slightly stale satellite data.

Continuity is a fundamental notion in mathematics. However, it is difficult to apply continuity proofs from real analysis to functions that are coded as imperative programs, especially when they use diverse data types and features such as assignments, branches, and loops. We associate data types with metric spaces as opposed to just sets of values, and continuity of typed programs is phrased in terms of these spaces. Our analysis reduces questions about continuity to verification conditions that do not refer to infinitesimal changes and can be discharged using off-the-shelf SMT solvers. Challenges arise in proving continuity of programs with branches and loops, as a small perturbation in the value of a variable often leads to divergent control-flow that can lead to large changes in values of variables. Our proof rules identify appropriate "synchronization points" between executions and their perturbed counterparts, and establish that values of certain variables converge back to the original results in spite of temporary divergence.

We prove our analysis sound with respect to the traditional ϵ-δ definition of continuity. We demonstrate the precision of our analysis by applying it to a range of classic algorithms, including algorithms for array sorting, shortest paths in graphs, minimum spanning trees, and combinatorial optimization. A prototype implementation based on the Z3 SMT-solver is also presented.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of programming languages—Program analysis.; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Mechanical verification; G.1.0 [*Numerical Analysis*]: General—Error analysis, Stability

General Terms Theory, Verification

Keywords Continuity, Program Analysis, Uncertainty, Robustness, Perturbations, Proof Rules.

POPL'10, January 17–23, 2010, Madrid, Spain.
Copyright © 2010 ACM 978-1-60558-479-9/10/01...$10.00

```
DIJK($G$ : graph, $src$ : node)
for each node $v$ in $G$
    do $d[v] := \perp$;   $prev[v] :=$ UNDEF ;
$d[src] := 0$;   $WL :=$ set of all nodes in $G$;
while $WL \neq \emptyset$
    do choose node $w \in WL$ such that $d[w]$ is minimal;
       remove $w$ from $WL$;
       for each neighbor $v$ of $w$
           do $z := d[w] + G[w, v]$;
              if $z < d[v]$
                 then $d[v] := z$;   $prev[v] := w$
```

Figure 1. Dijkstra's shortest-path algorithm

1. Introduction

Uncertainty in computation has long been a question of interest in computing [8]. An important reason for the uncertain behavior of programs is erroneous data [21]: the traffic data that a GPS device uses to plan a path may be slightly stale at the time of computation [15], and the sensor data that an aircraft controller processes may be slightly wrong [11]. In a world where computation is increasingly intertwined with sensor-derived perceptions of the physical world [12], such uncertain inputs are ubiquitous, and the assurance that programs respond *robustly* to them often vital. Does the output of a GPS device change only slightly in response to fluctuations in its inputs? If so, can we prove this fact automatically?

Robustness of programs to small amounts of error and uncertainty in their inputs can be defined via the mathematical notion of *continuity*. Recall that a function $f(x) : \mathbb{R} \to \mathbb{R}$ is continuous at the point $x = c$ if, for all infinitesimal deviations of x from c, the value of $f(x)$ deviates at most infinitesimally from $f(c)$. This provides a concrete definition of robustness: if a program implements a function that is continuous at c, then its output is not affected by small fluctuations of its input variable around the value c.

To see this definition of continuity of programs and its application in specifying robustness, consider an algorithm routinely used by path-planning GPS devices: Dijkstra's shortest-path algorithm. A program *Dijk* implementing the algorithm is shown in Figure 1—here, G is a graph with real edge-weights, src is the source node, and $G[u, v]$ is the weight of the edge (u, v). It is a property of *Dijk* that the set of paths that it computes can change completely in response to small perturbations to G (by this, let us mean that the weight of some edge of G changes slightly). However, what if our robustness requirement asserts that it is the *weight* of the shortest path that must be robust to small changes to G? In other words, assuming the array d of shortest-path distances is the program's output, is the program continuous? We note that it is—d changes at most infinitesimally if G changes infinitesimally.

Questions of continuity and robustness appear routinely in the literature on dynamical and hybrid systems [16, 17]. However, these approaches apply to systems defined by differential equations, hybrid automata [1], or graph models [18]. In the *program* verification literature, robustness has previously been considered in the restricted settings of functional synchronous programs [2], finite-state systems [10], and floating-point roundoff errors [6, 7, 13, 14]. Also, for purely numerical programs, robustness can be analyzed by abstract interpretation using existing domains [4, 5].

In contrast, this paper assumes a framing of robustness in terms of continuity, and presents a general proof framework for continuity that applies to programs—such as $Dijk$—that use data-structures such as graphs and arrays, as well as features like imperative assignments, branches, and loops. The search for such a proof framework, however, is fraught with challenges. Even a program whose inputs and outputs are over continuous domains may use temporary variables of discrete types, and manipulate data using imperative assignments, branches, and loops. It can have multiple inputs and outputs, and an output can be continuous in one input but not in another. Indeed, prior work [9] has argued for a notion of continuity for software, but failed to offer new program analysis techniques, concluding that "it is not possible in practice to mechanically test for continuity" in the presence of loops.

Recall the seemingly simple continuity property of the program $Dijk$: if d is its output, then it is continuous. However, it is highly challenging to establish this property from the text of $Dijk$. One way to prove it would be to first prove that $Dijk$ computes shortest paths, and then to establish that the costs of these paths are continuous in the weights of the edges of G. Such a proof, however, would be highly specialized and impossible to automate. What we want, therefore, is a proof methodology that reasons about continuity without aiming to prove full functional correctness, is applicable to a wide range of algorithms, and can be automated. Here we present such a method. We highlight below some of the challenges that arise and our proof rules for addressing them.

Presence of Control-flow. One challenge in proving continuity of programs is control-flow: a small perturbation can cause control to flow along a different branch leading to a syntactically divergent behavior. For example, consider the branch in Lines 9-10 in $Dijk$, which allow semantically different behaviors of either "setting $d[v]$ to z" or "leaving $d[v]$ unchanged". We present a rule for proving continuity of such if-then-else code-fragments. The key idea is to show that the two (otherwise semantically different) branches become semantically equivalent in situations (known as *discontinuities*) where the conditional can flip its value. Using this rule (ITE-1), we can show that the l-value $d[v]$ is continuous after the code-fragment "if $d[v] < z$ then $d[v] := z$." This is because the conditional $d[v] < z$ can flip values on small perturbations only when $d[v]$ was already close to z; however, under such a condition the expressions $d[v]$ and z evaluate to approximately the same value.

Non-inductiveness of continuity. The next challenge comes in extending the continuity proofs to loops. A natural approach is to set up an inductive framework for establishing continuity during each loop iteration (rule SIMPLE-LOOP). However, it turns out that continuity is not an inductive property for several loops (unlike invariants), meaning that the program variables that are continuous at the end of the loop are not necessarily continuous in each loop iteration. For example, while the array d is a continuous function of G on termination of $Dijk$, it is not continuous across each loop iteration. This is because the array d is updated in each loop iteration based on the choice of w from the workset W such that $d[w]$ is minimal. Now small fluctuations in the input weights can cause small fluctuations in the elements of d, causing it to choose a very different node w and potentially alter d completely.

Key to solving this challenge is the observation that if we group some loop iterations together, then continuity becomes an inductive property of the groupings. These groupings are referred to as *epochs*, and they have the property that the constituent iterations can be executed in any order without violating the semantics of the program. The LOOP proof-rule discharges this obligation by establishing *commutativity* of the loop body. Returning to Dijkstra's algorithm, this grouping is based on the set of elements w that have similar weight $d[w]$. The property of this grouping is that $P(w_1); P(w_2)$ is semantically equivalent to $P(w_2); P(w_1)$ where w_1 and w_2 are two elements such that $d[w_1] = d[w_2]$, where $P(w)$ represents the code-fragment in Lines 7-10.

Perturbations in Number of Loop Iterations. Another challenge in continuity proofs for loops is that the number of loop iterations may differ as a result of small perturbations to the inputs. We note that whenever such a behavior happens in continuous loops, then the effect of the extra iterations either in the original or the perturbed execution is almost equal to that of a **skip**-statement. This property—called *synchronized termination condition*—is asserted in our rules for loops. (Dijkstra's algorithm does not exemplify this challenge though, as the loop body is executed once for each graph node regardless of small changes to the edge weights.)

1.1 Contributions and Organization of the Paper

This paper makes the following contributions.

- We formalize the notion of continuity of programs by associating data-types with metric spaces and operators with continuity specifications (Sec. 2).
- We present structural rules to prove the continuity of programs in presence of control-flow (Sec. 4) and loops (Sec. 5), after establishing a formalism to reason about continuity of expressions in Sec. 3. These proof rules require establishing standard properties of code-fragments, in particular, establishing equivalence or commutativity, which can be discharged using off-the-shelf SMT solvers or assertion checkers.
- We prove our proof rules sound with respect to the standard ϵ-δ definition of continuity. This is quite challenging because the proof rules do not refer to ϵ or δ.
- We demonstrate the precision of our proof rules by showing that our framework can be used to prove continuity of several continuous classical algorithms. Sec. 6 discusses our implementation of a prototype of our framework that discharges proof rules using the SMT-solver Z3. Our current implementation requires the user to provide some annotations to identify the requisite components of the LOOP proof rule, though there are heuristics that can be used to automate this step.

2. Problem formulation

In this section, we fix a notion of continuity for imperative programs and formulate the problem of continuity analysis. First, we define a language whose semantics allows for a notion of *distances* between states—in fact, states are now elements of a *metric space*. Second, we define continuity for programs as standard mathematical continuity applied to their semantic functions. As programs have multiple inputs and observable outputs, we allow for statements such as "Program P is continuous in input x but not in input y," meaning that a small change to x must cause small changes to the *observable* program outputs, but that a small change to y may change the observable outputs arbitrarily.

Programs and expressions. We begin by fixing a simple imperative language (henceforth called IMP). The language has a single non-standard feature: a *distance metric* for each data type. Types here represent metric spaces rather than sets of values, and the semantics of expressions and programs are given by functions between metric spaces. This lets us define continuity of programs using standard mathematical machinery.

Let a *distance* be either a non-negative real, or a special value ∞ satisfying $x < \infty$ for all $x \in \mathbb{R}_{\geq 0}$. We define metric spaces over these distances in the usual way. Also, we assume:

- A set of data types. Each type τ in this set is associated with *distance metric* $dist_\tau$, and represents a metric space Val_τ whose distance measure is $dist_\tau$. The space Val_τ is known as the space of *values* of τ. For our convenience, we assume that each Val_τ contains a special value $\perp$ representing "undefined".

  In particular, we allow the types `bool` and `real` of booleans and reals. The type `real` is associated with the standard Euclidean metric, defined as $dist_{\texttt{real}}(x, y) = |x - y|$ if $x, y \neq \perp$, and $dist_{\texttt{real}}(x, y) = \infty$ otherwise. The metric on `bool` is the *discrete metric*, defined as $dist_{\texttt{bool}}(x, y) = 0$ if $x = y$, and ∞ otherwise.
- A universe Var of typed variables.
- A set $\mathcal{O}$ of *(primitive) operators*. Each operator op comes with a unique *signature* $op : \tau(p_1 : \tau_1, \ldots, p_n : \tau_n)$, where for all $i, p_i \notin Var$. Intuitively, p_i is a formal parameter of type τ_i, and τ is the type of the output value. For example, the `real` type comes with operators for addition, multiplication, division, etc.

The syntax of expressions e is now given by $e \;::=\; x \mid op(e_1, \ldots, e_n)$, where $x \in Var$ and $op \in \mathcal{O}$. Here $op(e_1, \ldots, e_n)$ is an application of the operator op on the operands $e_1, \ldots, e_n$. The set of variables appearing in the text of e is denoted by $Var(e)$. For easier reading, we often write our expressions in infix. Expressions are typed by a natural set of typing rules— as our analysis is orthogonal to this type system, we assume all our expressions to be well-typed.

As for programs P, they have the syntax:

$$\begin{array}{lll} P & ::= & \textbf{skip} \mid x := e \mid \textbf{if } b \textbf{ then } P_1 \textbf{ else } P_2 \\ & & \mid \textbf{while } b \textbf{ do } P_1 \mid P_1; P_2 \end{array}$$

where e is an expression and b a boolean expression. We denote the set of variables appearing in the text of P by $Var(P)$. For convenience, we sometimes annotate statements within a program with *labels* l. The interpretation is standard: l represents the control point immediate preceding the statement it labels.

As for semantics, let us first define a *state*:

Definition 1 (State)**.** A *state* is a map σ assigning a value in Val_τ to each $x \in Var$ of type τ.

The set of all states is denoted by Σ.

The semantics of an expression e of type τ is now defined as a function $[\![e]\!]$ of the form $\Sigma \to Val_\tau$. As expressions are built using operators, we presuppose a *semantic map* $[\![op]\!]$ for each operator $op \in \mathcal{O}$. Let op have the signature $op : \tau(p_1 : \tau_1, \ldots, p_n : \tau_n)$, and let Σ_{op} be the set of maps assigning suitably typed values to the p_i's. Then $[\![op]\!]$ is a map of type $\Sigma_{op} \to Val_\tau$.

The semantic function $[\![e]\!]$ for an expression e is now defined as:

$$[\![e]\!](\sigma) = \begin{cases} [\![op]\!]([\![e_1]\!](\sigma), \ldots, [\![e_n]\!](\sigma)) & \text{if } e = op(e_1, \ldots, e_n) \\ \sigma(x) & \text{if } e = x \in Var. \end{cases}$$

As for programs, we use a standard *functional (denotational) semantics* [23] for them. For simplicity, let us only consider programs that *terminate on all inputs*. The *semantic function* for a program P is then a map $[\![P]\!]$ of type $\Sigma \to \Sigma$ such that for all states σ_{in}, $[\![P]\!](\sigma_{in})$ is the state at which P terminates after starting execution from the state σ_{in}. The inductive definition of $[\![P]\!]$, being standard, is omitted. Note that both $[\![e]\!]$ and $[\![P]\!]$ are functions between metric spaces.

Continuity. Now that we have defined the semantics of programs as maps between metric spaces, we can use the standard ϵ-δ definition [20] to define their continuity. As programs usually have multiple inputs and outputs, we consider a notion of continuity that is parameterized by a set of *input variables* In and a set of *observable* variables Obs.

For a set of variables V, let us call two states V*-close* if they differ at most slightly in the values of the variables in V, and V*-equivalent* if they agree on values of all variables in V. Let a state σ' be a *small perturbation* of a state σ if the value of each variable in In is approximately equal, and the value of each variable not in In exactly the same, in σ and σ'. We define:

Definition 2 (Perturbations, V-closeness, V-equivalence)**.** For $\epsilon \in \mathbb{R}^+$, a state σ, and a set $In \subseteq Var$ of *input variables*, a state σ' is an *ϵ-perturbation* of a state σ (written as $Pert_{\epsilon,In}(\sigma, \sigma')$) if for all variables $x \in In$ of type τ, we have $dist_\tau(\sigma(x), \sigma'(x)) < \epsilon$, and for all variables $y \notin In$ of type τ, we have $\sigma(y) = \sigma'(y)$.

For $V \subseteq Var$ and $\epsilon \in \mathbb{R}$, the states σ and σ' are *ϵ-close in* V (written as $\sigma \approx_{\epsilon,V} \sigma'$) if for all $x \in V$ of type τ, we have $dist_\tau(\sigma(x), \sigma'(x)) < \epsilon$. The states are V*-equivalent* (written as $\sigma \equiv_V \sigma'$) if for all $x \in V$, we have $\sigma(x) = \sigma'(x)$.

Continuity of programs and expressions can now be defined by applying the traditional ϵ-δ definition:

Definition 3 (Continuity of expressions and programs)**.** Let $In \subseteq Var$ be a set of *input variables*. An expression e of type τ is *continuous* at a state σ in In if for all $\epsilon \in \mathbb{R}^+$, there exists a $\delta \in R^+$ such that for all σ' satisfying $Pert_{\delta,In}(\sigma, \sigma')$, we have $dist_\tau([\![e]\!](\sigma), [\![e]\!](\sigma')) < \epsilon$.

A program P is continuous at a state σ in a set In of input variables and a set $Obs \subseteq Var$ of *observable variables* if for all $\epsilon \in \mathbb{R}^+$, there exists a $\delta \in R^+$ such that for all σ' satisfying $Pert_{\delta,In}(\sigma, \sigma')$, we have $[\![P]\!](\sigma) \approx_{\epsilon,Obs} [\![P]\!](\sigma')$.

Intuitively, if e is continuous in In, then small changes to variables in In can change its value at most slightly, and if P is continuous in In and Obs, then small changes to variables in In can only cause small changes to variables in Obs (variables outside Obs can be affected arbitrarily).

While the ϵ-δ definition can be directly used in continuity proofs [20], such proofs are highly *semantic*. More *syntactic* proofs would reason inductively, using axioms, inference rules, and invariants. The appeal of a framework for such proofs would be twofold. First, instead of closed-form mathematical expressions, it would target *programs* that may often not correspond to cleanly defined or easily identifiable mathematical functions. Second, it would allow mechanization, even automation. Therefore, we formulate the problem of *continuity analysis*:

Problem (Continuity analysis)**.** Develop a set of syntactic proof rules that can soundly and completely determine if a program P is continuous in a set In of input variables and a set Obs of observable variables, at each state σ satisfying a property c.

In the next few sections, we present our solution to this problem. Our rules are sound. While we do not claim completeness, we offer an empirical substitute: nearly-automatic continuity proofs for 11 classic algorithms, most of them picked from a standard undergraduate textbook [3].

3. Continuity judgments and specifications

In this section, we define the basic building blocks of our reasoning framework. These include the judgments it outputs, as well as the user-provided specifications that parameterize it.

3.1 Continuity judgments

Suppose our goal is to judge the continuity of an expression e or a program P in a set In of input variables and, in the latter case, a set Obs of observable variables. Instead of obtaining judgments that hold a specific state σ, we judge continuity at a *set of states* symbolically represented by a formula b. Therefore, we define:

Definition 4 (Continuity judgment)**.** A *continuity judgment* for an expression e is a term $b \vdash \mathit{Cont}(e, \mathit{In})$, where b is a formula with free variables in Var, and $\mathit{In} \subseteq \mathit{Var}$.

A judgment for a program P is a term $b \vdash \mathit{Cont}(P, \mathit{In}, \mathit{Obs})$, where b is a formula over Var, and $\mathit{In}, \mathit{Obs} \subseteq \mathit{Var}$.

The judgment $b \vdash \mathit{Cont}(e, \mathit{In})$ is read as: "e is continuous in In at each state σ satisfying the property b." The judgment $b \vdash \mathit{Cont}(P, \mathit{In}, \mathit{Obs})$ says that the program P is continuous in the set In of input variables and the set Obs of observable variables at all states satisfying b. The judgments are *sound* if these statements are true according to the definition of continuity in Definition 3.

Note that for a judgment $b \vdash \mathit{Cont}(P, \mathit{In}, \mathit{Obs})$ (similarly, $b \vdash \mathit{Cont}(e, \mathit{In})$) to be sound, it suffices for In to be an *under-approximation* of the set of input variables, and Obs to be an *over-approximation* of the set of observable variables, in which P (similarly, e) is continuous.

Example 1. The expression $(x+y)$, where $+$ denotes real addition, is always continuous in $\{x, y\}$. On the other hand, the expression $\frac{x}{y}$, which evaluates to $\bot$ for $y = 0$, is not always continuous. Two sound judgments involving it are $\mathit{true} \vdash \mathit{Cont}(\frac{x}{y}, \{x\})$ and $(y \neq 0) \vdash \mathit{Cont}(\frac{x}{y}, \{x, y\})$, which say that: (1) the result of division is always continuous in the dividend, and (2) continuous in all non-zero divisors.

Now consider the type $(\tau \to \texttt{real})$ of *real-valued arrays*: partial functions from the index type τ to the type $\texttt{real}$. For any such array A, we define $\mathit{Dom}(A)$ to denote the *domain* of A—i.e., the set of all x such that $A[x]$ is defined.

Let us consider the following *supremum metric*:

$$\mathit{dist}_{\tau \to \texttt{real}}(A, B) = \begin{cases} \max_{i \in \mathit{Val}_\tau}\{\mathit{dist}_{\texttt{real}}(A[i], B[i])\} \\ \quad \text{if } \mathit{Dom}(A) = \mathit{Dom}(B) \\ \infty \qquad \text{otherwise.} \end{cases}$$

Intuitively, the distance between A and B is the maximum distance between elements in the same position in the two arrays.

Now consider the array-update operator Upd, commonly used to model writes to arrays. The operator has the parameters A (of type $\tau \to \texttt{real}$), i (an integer), and p (a real), and returns the array A' such that $A'[i] = p$, and $A'[j] = A[j]$ for all $j \neq i$. To exclude erroneous writes, let Upd evaluate to $\bot$ if $p = \bot$ or if A contains an undefined value. In that case, the following judgment is sound:

$$(p \neq \bot) \wedge (\forall i,\ A[i] \neq \bot) \vdash \mathit{Cont}(\mathit{Upd}(A, i, p), \{A, i, p\}) \ .$$

Observe that $\mathit{Upd}(A, i, p)$ is judged to be continuous in i. The reason is that as i is drawn from a discrete metric space (the `int` type), the only way to change it *infinitesimally* is to not change i at all. Continuity in i follows trivially. □

Example 2. Consider the program $P = x := x + 1;\ y := z/x$. A sound continuity judgment for P is $(x + 1 \neq 0) \vdash \mathit{Cont}(P, \{x, y, z\}, \{y\})$.

Now consider the following program P': **if** $(x \geq 0)$ **then** $r := y$ **else** $r := z$. Denote by c the formula $(x = 0) \Rightarrow (y = z)$. Then the continuity judgment $c \vdash \mathit{Cont}(P', \{x, y, z\}, \{r\})$ is sound.

To see why, note that for fixed x and z, an infinitesimal change in y either causes no change to the final value of r (this happens if $x < 0$), or changes it infinitesimally. A similar argument holds for infinitesimal changes to z. As for x, the guard $(x \geq 0)$ is continuous in x (i.e., is not affected by small changes to x) at all $x \neq 0$. As a result, under the precondition $x \neq 0$, infinitesimal changes to x, y, z changes the final value of r at most infinitesimally.

At $x = 0$, of course, the guard is discontinuous—i.e., can change value on an infinitesimal change to x. In this case, an infinitesimal change to x may cause the output r to change from the value of y to that of z. However, by our precondition, we have $y = z$ whenever $x = 0$. Thus means that even if the guard evaluates differently, the observable output r is affected at most infinitesimally by infinitesimal changes to x, y, z. In other words, under the assumed conditions, the discontinuous behavior of the guard does not affect the continuity of P' in r.

Now consider the judgment $\mathit{true} \vdash \mathit{Cont}(P', \{y, z\}, \{r\})$. As we only assert continuity in inputs y and z, it is clearly sound. □

3.2 Continuity specifications

As the operators in our programming language can be arbitrary, we need to know their continuity properties to do continuity proofs. This information is provided by the programmer through a set of *continuity specifications*. We define:

Definition 5 (Continuity specification)**.** A *continuity specification* for an operator op, with the signature $\mathit{op} : \tau(p_1 : \tau_1, \ldots, p_n : \tau_n)$, is a term $c \vdash S$, where c is a boolean expression over $p_1, \ldots, p_n$ and $S \subseteq \{p_1, \ldots, p_n\}$.

An operator is allowed to have multiple specifications. Suppose the operator op has a specification $c \vdash S$. The interpretation is that the semantic map $[\![\mathit{op}]\!]$ is continuous in S at each state over $\{p_1, \ldots, p_n\}$ that satisfies c. The specification is *sound* if this is indeed the case. Intuitively, application of the operator *preserves* the continuity properties of arguments corresponding to parameters $p_i \in S$, and can potentially introduce discontinuities in the remaining arguments.

Example 3. Let the real addition operator have the signature $+ : \texttt{real}(x : \texttt{real}, y : \texttt{real})$. A sound specification for it is $\mathit{true} \vdash \{x, y\}$. Now consider the real division operator $/$, with similar signature. Two sound specifications for it are $\mathit{true} \vdash \{x\}$ and $(y \neq 0) \vdash \{x, y\}$. □

Continuity specifications as above have a natural relationship with modular analysis. While operators in programming languages are usually low-level primitives, nothing in our definition prevents an operator from being a procedural abstraction of a program. As the reasoning framework *assumes* its continuity properties, it defines a *level of abstraction* at which continuity is judged. In the current paper, we assume that this procedural abstraction is defined by the programmer. In future work, we will consider an *interprocedural continuity analysis*, where operator specifications are generalized into *continuity summaries* mined from a program.

4. Analysis of expressions and loop-free programs

In this section, we begin the presentation of our analysis. The main contributions presented here are: (1) a continuity analysis of expressions through structural induction; and (2) an analysis of branching programs based on identification of the *discontinuities of a boolean expression*, and queries about *program equivalence* discharged through an SMT-solver.

4.1 Analysis of expressions

The main idea behind continuity analysis of expressions is simple: an expression e is continuous in In if it is obtained by recursively applying continuous operators on variables in In. If e has a subexpression e' that is either discontinuous or an argument to an operator that does not preserve its continuity, then we should judge e to be discontinuous in all variables in e'.

The inference rules for the analysis are presented in Figure 2. Here, for expressions $c, e_1, \ldots, e_n$, the notation $c[x_1 \mapsto e_1, \ldots, x_n \mapsto e_n]$ denotes the expression obtained by substituting each variable x_i in c by e_i. The rule BASE states that a variable is always continuous in itself. WEAKEN observes that a continuity

$$(\text{Base})\ \frac{x \in \mathit{Var}}{\mathit{true} \vdash \mathit{Cont}(x, \{x\})}$$

$$(\text{Weaken})\ \frac{b \vdash \mathit{Cont}(e, \mathit{In}) \quad b' \Rightarrow b \quad \mathit{In}' \subseteq \mathit{In}}{b' \vdash \mathit{Cont}(e, \mathit{In}')}$$

$$(\text{Join})\ \frac{b \vdash \mathit{Cont}(e, \mathit{In}_1) \quad b \vdash \mathit{Cont}(e, \mathit{In}_2)}{b \vdash \mathit{Cont}(e, \mathit{In}_1 \cup \mathit{In}_2)}$$

$$(\text{Frame})\ \frac{b \vdash \mathit{Cont}(e, \mathit{In}) \quad z \notin \mathit{Var}(e)}{b \vdash \mathit{Cont}(e, \mathit{In} \cup \{z\})}$$

$$(\text{Op})\ \frac{\begin{array}{c} \mathit{op} \text{ has parameters } p_1, \dots, p_n \text{ and a specification } c \vdash S \\ \forall p_i \in S.\ b \vdash \mathit{Cont}(e_i, \mathit{In}) \\ \forall p_i \notin S.\ \mathit{In} \cap \mathit{Var}(e_i) = \emptyset \\ c' = c[p_1 \mapsto e_1, \dots, p_n \mapsto e_n] \end{array}}{c' \wedge b \vdash \mathit{Cont}(\mathit{op}(e_1, \dots, e_n), \mathit{In})}$$

Figure 2. Continuity analysis of expressions.

judgment can be soundly weakened by restricting the set of input variables in which continuity is asserted, or the set of states at which continuity is judged. FRAME observes that an expression is always continuous in variables to which it does not refer. As for JOIN, it uses the mathematical fact that if a function is continuous in two sets of input parameters, then it is continuous in their union.

The rule OP derives continuity judgments for expressions $e = \mathit{op}(e_1, \dots, e_n)$, where op is an operator. Intuitively, if e_i is continuous in a set of variables In, and if op is continuous in its i-th parameter, then e is also continuous in each $x \in \mathit{In}$. The situation is complicated by the fact that a variable can appear in multiple e_i's, and that if $x \in \mathit{In}$ appears in any e_j such that op is not continuous in p_j or e_j is not continuous in x, then e is potentially discontinuous in x. Thus, we must ensure that $\mathit{In} \cap \mathit{Var}(e_j) = \emptyset$ for all such e_j; also, each e_i such that $p_i \in S$ must be continuous in *all* the variables in In. The rule OP has these conditions as premises.

Example 4. Consider the expression $e = \frac{x}{x+y}$, where x and y are real-valued variables, and the judgment $(x > 0) \wedge (y > 0) \vdash \mathit{Cont}(e, \{y\})$. To prove it in our system using specifications of real addition and division as before, we first use the rules BASE and FRAME, OP, and the specification of $+$ to prove that $\mathit{true} \vdash \mathit{Cont}((x + y), \{x, y\})$. Now we derive that $\mathit{true} \vdash \mathit{Cont}(x, \{x, y\})$, then use OP and the specification of $/$ to show that whenever $(x + y) \neq 0$, e is continuous in $\{x, y\}$. Finally, we use WEAKEN to get the desired judgment. □

Using induction and ϵ-δ reasoning, we can show that:

Theorem 1. *If all operator specifications are sound, the proof system in Figure 2 only derives sound continuity judgments.*

4.2 Analysis of loop-free programs

The analysis of loop-free programs brings out some subtleties—e.g., to prove the continuity of programs with branches, we must discharge a *program equivalence* query through an SMT-solver.

Example 5. Recall the program in Example 2: $P = x := x + 1; y := z/x$. As argued earlier, the judgment $(x + 1 \neq 0) \vdash \mathit{Cont}(P, \{x, y, z\}, \{y\})$ is sound. One proof for it is as follows:

1. Show that $(x + 1)$ is always continuous in x. From this, derive the judgment $\mathit{true} \vdash \mathit{Cont}(x := x + 1, \{x, y, z\}, \{x\})$.
2. Establish that $(x \neq 0) \vdash \mathit{Cont}(y := z/x, \{x\}, \{y\})$.
3. Propagate backward the condition for continuity of P_2, obtaining the precondition $(x + 1) \neq 0$ for P.
4. Compose the judgments in (1) and (2), using the fact that x is the only observable output in the former and the only input in the latter. This gives us the desired judgment.

$$(\text{Skip})\ \frac{}{\mathit{true} \vdash \mathit{Cont}(\mathbf{skip}, \emptyset, \emptyset)}$$

$$(\text{Join})\ \frac{b \vdash \mathit{Cont}(P, \mathit{In}_1, \mathit{Obs}) \quad b \vdash \mathit{Cont}(P, \mathit{In}_2, \mathit{Obs})}{b \vdash \mathit{Cont}(P, \mathit{In}_1 \cup \mathit{In}_2, \mathit{Obs})}$$

$$(\text{Weaken})\ \frac{\begin{array}{c} b \vdash \mathit{Cont}(P, \mathit{In}, \mathit{Obs}) \quad b' \Rightarrow b \\ \mathit{Obs}' \subseteq \mathit{Obs} \qquad \mathit{In}' \subseteq \mathit{In} \end{array}}{b' \vdash \mathit{Cont}(P, \mathit{In}', \mathit{Obs}')}$$

$$(\text{Frame})\ \frac{b \vdash \mathit{Cont}(P, \mathit{In}, \mathit{Obs}) \quad z \notin \mathit{Var}(P)}{b \vdash \mathit{Cont}(P, \mathit{In} \cup \{z\}, \mathit{Obs} \cup \{z\})}$$

$$(\text{Assign-1})\ \frac{b \vdash \mathit{Cont}(e, \mathit{In})}{b \vdash \mathit{Cont}(x := e, \mathit{In} \cup \{x\}, \mathit{Var}(e) \cup \{x\})}$$

$$(\text{Assign-2})\ \frac{}{b \vdash \mathit{Cont}(x := e, \mathit{Var}(e) \cup \{x\}, \mathit{Var}(e) \setminus \{x\})}$$

$$(\text{Sequence})\ \frac{\begin{array}{c} b_1 \vdash \mathit{Cont}(P_1, \mathit{In}_1, \mathit{Obs}_1) \quad \mathit{In}_2 \subseteq \mathit{Obs}_1 \\ b_2 \vdash \mathit{Cont}(P_2, \mathit{In}_2, \mathit{Obs}_2) \quad \{b_1\}P_1\{b_2\} \end{array}}{b_1 \vdash \mathit{Cont}(P_1; P_2, \mathit{In}_1, \mathit{Obs}_2)}$$

$$(\text{Ite-1})\ \frac{\begin{array}{c} c \vdash \mathit{Cont}(P_1, \mathit{In}, \mathit{Obs}) \quad c \vdash \mathit{Cont}(P_2, \mathit{In}, \mathit{Obs}) \\ c' \vdash \mathit{Cont}(b, \mathit{Var}(b)) \quad (c \wedge \neg c') \vdash (P_1 \equiv_{\mathit{Obs}} P_2) \end{array}}{c \vdash \mathit{Cont}(\mathbf{if}\ b\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2, \mathit{In}, \mathit{Obs})}$$

$$(\text{Ite-2})\ \frac{\begin{array}{c} c \vdash \mathit{Cont}(P_1, \mathit{In}, \mathit{Obs}) \quad c \vdash \mathit{Cont}(P_2, \mathit{In}, \mathit{Obs}) \\ c' \vdash \mathit{Cont}(b, \mathit{In}') \end{array}}{c \wedge c' \vdash \mathit{Cont}(\mathbf{if}\ b\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2, \mathit{In} \cap \mathit{In}', \mathit{Obs})}$$

Figure 3. Continuity analysis of loop-free programs.

Now consider the program P' from Example 2: **if** $(x \geq 0)$ **then** $r := y$ **else** $r := z$. As we argued earlier, the judgment $c \vdash \mathit{Cont}(P', \{x, y, z\}, \{r\})$, where c equals $(x = 0) \Rightarrow (y = z)$. A proof can have the following components:

1. Identify an overapproximation Σ' of the set of states (in this case captured by the formula $(x = 0)$) at which the loop guard is discontinuous—i.e., can "flip" on a small perturbation.
2. Assuming c holds initially, the two branches of P', when executed independently from a state in Σ', terminate at states agreeing on the value of r.
3. Each branch is continuous at all states in c, in the set $\{x, y, z\}$ of input variables and the set $\{r\}$ of observable variables.

Here, condition (2) asserts that even if a state executing along a branch were executed along the other branch, the observable result would be the same. Together with condition (3), it implies that even if an execution and its "perturbed" counterpart follow different branches, they reach approximately equivalent states at the join-point. Thus, it asserts a form of *synchronization*—following a period of divergence—between the original and perturbed executions.

Now consider the sound judgment $\mathit{true} \vdash \mathit{Cont}(P', \{y, z\}, \{r\})$. This time, we establish that: (1) each branch of P' is unconditionally continuous in y and z, and (2) the guard $(x \geq 0)$ is unconditionally continuous in these variables as well.

Let $\min$ be the program **if** $(x \leq y)$ **then** x **else** y computing the *minimum* of two real-valued variables. Using a similar style of proof as for P', we can establish the sound judgment $\mathit{true} \vdash \mathit{Cont}(\min, \{x, y\}, \{x, y\})$—i.e., the fact that $\min$ is unconditionally continuous. A similar argument holds for $\max$. □

Let us now try to systematize the ideas in the above examples into a set of inference rules. We need some more machinery:

Discontinuities of a boolean expression. Our rule for branch-statements requires us to identify the set of states where a boolean expression b is *discontinuous*. For soundness, it suffices to work with an *overapproximation* of this set. This can be obtained by inferring a judgment of the form $c \vdash Cont(b, Var)$ about b— by the soundness of our analysis for expressions, $\neg c$ is overapproximates of the set of states where b is discontinuous in any variable.

As we have not written continuity specifications for boolean operators so far, let us do so now. To judge the continuity of boolean expressions, we plug these into the system in Figure 2.

Example 6. For simplicity, let us only consider boolean expressions over the types `real` and `bool`. We allow the standard comparison operators $=$, $\geq$, and $>$ with signatures such as $\geq$: `bool`$(x$: `real`$, y$: `real`$)$; we also have the usual real arithmetic operators. For boolean arithmetic, we use the standard operators $\wedge$, $\vee$, and $\neg$.

Specifications for operators involving booleans are as follows:

- We specify each comparison operator in $\{=, >, <, \leq, \geq\}$ (let it have formal parameters p and q) as $(p \neq q) \vdash \{p, q\}$. It is easy to see that this specification is sound.
- The operators $\wedge$ and $\vee$ (with formal parameters p and q) have the specification $true \vdash \{p, q\}$; the operator $\neg$ (with parameter q) has the specification $true \vdash \{q\}$. The reason (which also showed up in Example 1) is that these operators have discrete inputs. This implies that the only way to *infinitesimally* change an input x to any of them is to not change x at all. Unconditional continuity in the parameters follows trivially.

For example, consider the boolean expression $\neg(x^2 \geq 4) \vee (y < 10)$ (let us call it b). The reader can verify that using the above specifications, we can derive the judgment $(x^2 \neq 4) \wedge (y \neq 10) \vdash Cont(b, \{x, y\}, \{x, y\})$. It follows that $\neg((x^2 \neq 4) \wedge (y \neq 10))$ *overapproximates* the set of states at which b is discontinuous. □

Hoare triples and program equivalence. To propagate conditions for continuity through a program, we generate Hoare triples as verification conditions. These are computed using an off-the-shelf invariant generator. Additionally, we assume a solver for *equivalence* of programs [1]. Let $V \subseteq Var$ and c be a logical formula; now consider programs P_1 and P_2. We say that P_1 and P_2 are *V-equivalent under c*, and write $c \vdash (P_1 \equiv_V P_2)$, if for each state σ that satisfies c, we have $[\![P_1]\!](\sigma) \equiv_V [\![P_2]\!](\sigma)$. Our rule for branches generates queries about program equivalence as verification conditions.

The rules. Our rules for continuity analysis of loop-free programs are presented in Figure 3. Here, the rule JOIN is the analog of the rule JOIN for expressions. The rule WEAKEN lets us weaken a judgment by restricting either the precondition under which continuity holds, or restricting the sets of input and observable variables. The rule FRAME says that a program is always continuous in variables not in its text. The rule SKIP (taken together with the rule FRAME) says that **skip**-statements are always continuous in every variable. The rule ASSIGN-1 says that if the right-hand side of an assignment statement is continuous in In, then the statement is continuous in In even if the lvalue x is observable. ASSIGN-2 says that if x is not observable, then the statement is unconditionally continuous.

The rule SEQUENCE addresses sequential composition, systematizing the insight in Example 2. Suppose P_1 is executed from a state σ satisfying b_1 and ends at σ'. We have $\{b_1\}P_1\{b_2\}$; therefore, P_2 is continuous in In_2 at σ'. Now suppose we modify σ by infinitesimally changing some variables in In_1; the altered output state σ'' of P_1 approximately agrees with σ' on the values of variables in In_2 (as $In_2 \subseteq Obs_1$). Due to the premise about P_2, this can only change the output of P_2 infinitesimally (assuming Obs_2 is the set of observable variables).

The rule ITE-1 generalizes the first continuity judgment for the program P' made in Example 2. Here, $\neg c'$ is an overapproximation of the set of states at which the guard b is discontinuous. As P_1 and P_2 are equivalent whenever $(c \wedge \neg c')$, it does not matter if the guard "flips" as a result of perturbations—the **if**-statement is continuous in each variable in which *both* branches are continuous.

As for the rule ITE-2, it generalizes the second judgment about P' in Example 2. The (conditional) equivalence of P_1 and P_2 is not a premise here; therefore, the **if**-statement is guaranteed to be continuous only in the variables in which b is continuous. The inferred precondition for continuity is also restricted.

Using induction and ϵ-δ reasoning, we can show that:

Theorem 2. *The inference rules in Figure 3 are sound.*

Example 7. Consider the program $P = P_1; P_2$, where P_1 is $x := y/z$ and P_2 is the program **if** $(x \geq 0)$ **then** $r := y$ **else** $r := z$. Let $V = \{x, y, z, r\}$, and let c be $((x = 0) \Rightarrow (y = z))$. To establish the judgment $(y \neq 0) \wedge (z \neq 0) \vdash Cont(P, V, V)$, we first prove the judgment $(y \neq 0) \wedge (z \neq 0) \vdash Cont(P_1, V, V)$— this requires a continuity proof for the expression y/z, and use of the rules ASSIGN, FRAME and WEAKEN. Next we show that $c \vdash Cont(P_2, V, V)$, using, among others, the rule ITE-1. Finally, we apply the rule for sequential composition. □

[1] We note that program equivalence is a well-studied problem, an important application being *translation validation* [19] of optimizing compilers.

```
FLOYD-WARSHALL($G$ : graph)
1  for $k := 1$ to $n$
2      do for $i, j := 1$ to $n$
3          if $G[i, j] > G[i, k] + G[k, j]$
4             then $G[i, j] := G[i, k] + G[k, j]$;  $prev[i, j] := prev[k, j]$
```

Figure 4. Floyd-Warshall algorithm for all-pairs shortest paths

5. Continuity analysis of programs with loops

In this section, we present our continuity analysis of programs with loops. The main conceptual contributions presented here are: (1) An inductive rule for proving the continuity of loops, based on a generalization of our rule for loop-free programs. (2) A second rule for loops, based of induction where the basic step is a sequence of loop iterations (known as an *epoch*) rather than a single iteration. The latter rule is needed because many important applications cannot be proved continuous by an ordinary inductive argument. A soundness proof for it is also sketched.

5.1 Analysis of loops by induction

We start with a motivating example:

Example 8 (Floyd-Warshall algorithm). Let us consider the Floyd-Warshall all-pairs shortest-path algorithm with path recovery (Figure 4). Let us call this program FW; on termination, $G[i, j]$ contains the weight of the shortest path from i to j, and $prev[i, j]$ contains a node such that for some shortest path from i to j, the node right before j is $prev[i, j]$.

We view a graph G as a function from a set of edges—each edge being a pair (u, v) of natural numbers—to a set of real-valued edge-weights. Thus, G is a *real-valued array* as in Example 1.

Let the metric on real-valued arrays be as in Example 1; the metric on the discrete array $prev$ and the variables i, j is the discrete metric (previously used on the `bool` type). Then: (1) if a graph G has a node or edge that G' does not, then the distance between G and G' is ∞, and (2) otherwise, the distance between them is $\max_{(u,v) \in Dom(G)}\{\ |G[u, v] - G'[u, v]|\ \}$. In other words, a small change to G is defined as a small change to edge-weights keeping the node and edge structure intact.

As the final value of $G[i, j]$ gives the weight of the shortest path between i and j, the continuity claim $true \vdash Cont(FW, \{G\}, \{G\})$ is sound (however, the claim $true \vdash Cont(FW, \{G\}, \{prev\})$ is not—a previously valid shortest path may become invalidated due

$$\text{(Simple-loop)}\ \frac{\mathcal{I}(c) \vdash \mathit{Cont}(R, X, X) \quad c \vdash \mathit{Term}(P, X) \quad c \vdash \mathit{Sep}(P, X)}{c \vdash \mathit{Cont}(P, X, X)}$$

Figure 5. Rule SIMPLE-LOOP. (Here, $P = \mathbf{while}\ b\ \mathbf{do}\ (l : R)$.)

to perturbations.) We can establish this property by induction. Let R be the body of the inner loop (Line 3). Using an analysis as in the previous section, we have $\mathit{true} \vdash \mathit{Cont}(R, \{G\}, \{G\})$. Now let R^i represent i repetitions of R—i.e., the first i iterations of the loop taken together. Inductively assume that R^i is continuous in G—i.e., a small change to the initial value of G leads to a small perturbation of G at the end of the i-th iteration. By the continuity of R in G, R^{i+1} is also continuous in G.

Finally, observe that the expressions guarding the two loops here are *continuous* in G—in other words, small changes to G do not affect them. Therefore, an execution and its perturbed counterpart have the same number of loop iterations. This establishes that the entire loop represents a continuous function. □

Let us now systematize the ideas in this example into an inductive proof rule. Let P be a program of the form $\mathbf{while}\ b\ \mathbf{do}\ (l : R)$. Our goal here is to inductively derive judgments of the form $c \vdash \mathit{Cont}(P, X, X)$, where X is an *inductively obtained* set of variables. However, we need some more machinery.

Trace semantics and invariants. As our reasoning for loops requires inductive invariants, it requires a *trace semantics* of programs. Let us define the *trace* of P from a state σ_{in} as the unique sequence $\pi = \sigma_0\sigma_1 \ldots \sigma_{n+1}$ such that $\sigma_0 = \sigma_{in}$, and for each $0 \leq i \leq n$, $\{\sigma_i\}R\{\sigma_{i+1}\}$ and σ_i satisfies b.

Let c be an initial condition (given as a logical formula). We define a state σ to be *reachable* from c if it appears in the trace from a state satisfying c (note that in all reachable states, control is implicitly at the label l). The *loop invariant* $\mathcal{I}(c)$ of Q is an overapproximation of the set of states reachable from c. Our proofs assume a sound procedure to generate loop invariants.

The analysis. Let us now proceed to our continuity analysis. As before, our strategy is to discharge verification conditions that do not refer to continuity or infinitesimal changes. In particular, the following conditions are used:

- *Separability.* A set of variables X is *separable* in a program P if the value of any $z \in X$ at the terminal state of P *only depends* on the values of variables in X at the initial state. Formally, we say that X *is separable in P under an initial condition* c, and write $c \vdash \mathit{Sep}(P, X)$, if for all states σ, σ' reachable from c such that $\sigma \equiv_X \sigma'$, we have $[\![P]\!](\sigma) \equiv_X [\![P]\!](\sigma')$.

  Under this condition, we have an alternative definition of continuity that is equivalent to the one that we have been using: P is continuous at a state σ in a set X of input variables and the same set X of observable variables if for all $\epsilon \in \mathbb{R}^+$, there exists a $\delta \in R^+$ such that for all σ' satisfying $\sigma \approx_{\delta,X} \sigma'$, we have $[\![P]\!](\sigma) \approx_{\epsilon,X} [\![P]\!](\sigma')$. This definition is useful as we use the relation $\approx$ inductively.

- *Synchronized termination.* P fulfills the *synchronized termination* condition with respect to an initial condition c and a set of variables X (written as $c \vdash \mathit{Term}(P, X)$) if $\mathit{Var}(b) \subseteq X$, and one of the following holds: (1) The loop condition b satisfies $\mathit{true} \vdash \mathit{Cont}(b, X)$; (2) Let the formula c' represent an overapproximation of the set of states reachable from c where b is discontinuous in X. Then we have $c' \vdash R \equiv_X \mathbf{skip}$.

  Intuitively, this condition handles scenarios where an execution from a perturbed state violates the loop condition *earlier or later* than it would in the original execution. Under synchronized termination, the execution that continues does not veer "too far" from the state where it was when the other execution ended.

Of these, separability can be established using simple slicing. In the absence of nested loops, synchronized termination can be checked using an SMT-solver. If the loop body contains a nested loop, it can be checked using an SMT-solver in conjunction with an invariant generator for the inner loop [19].

Our rule SIMPLE-LOOP for inductively proving the continuity of P is now as in Fig. 5. Let us now see its use in an example:

Example 9. We revisit the program FW in Figure 4. (We assume it to be rewritten as a $\mathbf{while}$-program in the obvious way.) Let $X = \{G, i, j\}$. First, we observe that $\mathit{true} \vdash \mathit{Sep}(FW, X)$. As argued before, letting R be the loop body, we have $\mathit{true} \vdash \mathit{Cont}(R, X, X)$. Finally, the loop guard b only involves discrete variables—therefore, by the argument in Example 1, it is always continuous in X, which means that $\mathit{true} \vdash \mathit{Term}(FW, X)$. The sound judgment $\mathit{true} \vdash \mathit{Cont}(FW, X, X)$ follows. Using the WEAKEN rule from before, we can now obtain the judgment $\mathit{true} \vdash \mathit{Cont}(FW, \{G\}, \{G\})$.

Of course, this example does not illustrate the subtleties of the synchronized termination condition. For a more serious use of this condition, see Examples 11 and 13. □

As for soundness, we have:

Theorem 3. *The inference rule* SIMPLE-LOOP *is sound.*

Proof sketch. Consider an arbitrary trace $\pi = \sigma_0\sigma_1 \ldots \sigma_{m+1}$ of P starting from a state σ_0 that satisfies c, and any $\epsilon \in \mathbb{R}^+$. Let us guess a sequence $\langle \delta_0, \delta_1, \ldots, \delta_{m+1} \rangle$ such that: (1) $\delta_{m+1} < \epsilon$, and (2) for all states s, s' reachable from c, if s and s' are δ_i-close (in X), then if t and t' are the states satisfying $\{s\}R\{t\}$ and $\{s'\}R\{t'\}$, then s' and t' are δ_{i+1}-close (in X). Such a sequence exists as the loop body R is continuous. Now select $\delta = \delta_0$, and consider a state σ'_0 such that $\sigma_0 \equiv_{\delta,X} \sigma'_0$.

Recall that we assume that all programs in our setting are terminating. Therefore, any trace from σ'_0 is of the form $\pi' = \sigma'_0 \ldots \sigma'_{n+1}$ (without loss of generality, assume that $n \leq m$). By the continuity of the loop body, we have $\sigma_1 \approx_{\delta_1,X} \sigma'_1$. As the synchronized termination condition holds, one possibility is that b is continuous at both σ_1 and σ'_1, In this case, either both traces continue execution into the next epoch, or none do. The other possibility is that one of the traces violates b early due to perturbations, but in this case the rest of the other trace is equivalent to a $\mathbf{skip}$-statement. Generalizing inductively, we conclude that $\sigma_{m+1} \approx_{\epsilon,X} \sigma'_{n+1}$. As π, π' are arbitrary traces, the program P is continuous. □

5.2 Continuity analysis by induction over epochs

Unsurprisingly, there are many continuous applications whose continuity cannot be proved by the rule SIMPLE-LOOP. Pleasantly, many applications in this category are amenable to proof by richer form of induction that we have identified, and will now present. We start with two examples:

Example 10 (Dijkstra's algorithm). Consider our code for Dijkstra's algorithm (Figure 1; code partially reproduced in Fig. 6) once again. The one output variable is d—the array that, on termination, contains the weights of all shortest paths from src. The metric on G is as in Example 8.

Note that Line 5 in this code selects a node w such that $d[w]$ is minimal, so that to implement it, we require a mechanism to break ties on $d[w]$. In practice, such tie-breaking is implemented using an arbitrary linear order on the nodes. Such implementations, however, are ad hoc and can easily break inductive reasoning for continuity. To be on the safe side, we *conservatively abstract* the program by replacing the selection in Line 5 by *nondeterministic choice*. It is

```
4  while $WL \neq \emptyset$
5      do choose node $w \in WL$ such that $d[w]$ is minimal
6          remove $w$ from $WL$;
7          for each neighbor $v$ of $w \ldots$
```

Figure 6. Dijkstra's shortest-path algorithm

```
FRAC-KNAP($v$ : int $\to$ real, $c$ : int $\to$ real, $Budget$ : real):
@pre: $|v| = |c| = n$
 1  for $i := 0$ to $(n-1)$
 2      do $used[i] := 0$ ;
 3  $cur_c := Budget$;
 4  while $cur_c > 0$
 5      do choose item $m$ such that $t = (v[m]/c[m])$ is maximal
                    and $used[m] = 0$;
 6         $used[m] := 1$; $cur_c := cur_c - c[m]$
 7         $tot_v := tot_v + v[m]$;
 8         if $cur_c < 0$
 9           then $tot_v := tot_v - v[m]$;
10                $tot_v := tot_v + (1 + cur_c / c[m]) * v[m]$
```

Figure 7. Greedy Fractional Knapsack

easy to see that a proof of continuity for this abstraction implies a proof of continuity of the algorithm with a correct, deterministic implementation of tie-breaking. Let us call this abstraction $Dijk$. Usually, such abstractions correspond to the pseudocode of the algorithm under consideration, and are easily built from code.

While the abstraction $Dijk$ may seem to be nondeterministic, in reality it is not—every initial state here leads to a unique terminal state. Also, as d contains weights of shortest paths on termination, the judgment $true \vdash Cont(Dijk, \{G\}, \{d\})$ is sound. Proving it, however, is challenging. Assume the inductive hypothesis that d only changes slightly due to a small change to G. Now suppose that before the perturbation, nodes w_1 and w_2 were tied in the value of $d[\cdot]$ and we chose w_1, and that after the perturbation, we choose w_2. Clearly, this can completely change the value of d at the end of Line 10. Thus, the continuity of d is not an inductive property.

However, consider a *maximal set* of successive iterations in an execution processing elements tied on $d[\cdot]$. Let us view this collection of iterations—subsequently called an *epoch*—as a map from an input state σ_0 to a final value of d. It so happens that this map is robust to permutations—i.e., if σ_0 is fixed, then however we reorder the iterations in the collection, so is the value of the array d at the state σ_1. Second, small perturbations to σ_0 can lead to arbitrary reorderings of the iterations—however, they only lead to small perturbations to the value of d in σ_1 (on the other hand, the value of $prev$ may change completely). This is the insight we use in our proof rule. □

Example 11 (Greedy Fractional Knapsack). Consider the Knapsack problem from combinatorial optimization. We are given a set of items $\{1, \ldots, n\}$, each item i being associated with a *cost* $c[i]$ and a *value* $v[i]$ (we assume that c and v are given as arrays of non-negative reals). We are also given a non-negative, real-valued $budget$. The goal is to identify a subset $used \subseteq \{1, \ldots, n\}$ such that the constraint $\sum_{j \in used} c[i] \leq Budget$ is satisfied, and the value of $tot_v = \sum_{j \in used} v[i]$ is maximized.

Let the observable variable be tot_v; as small perturbations can turn previously feasible solutions infeasible, a program $Knap$ solving this problem correctly is discontinuous in the inputs c and $Budget$. At the same time, it is continuous in the input v.

Our analysis can establish the continuity of $Knap$ in v (see Section 6). For now we focus on the *fractional* variant of the problem, which has a greedy, optimal, polynomial solution and is more interesting from the continuity perspective. Here the algorithm can pick *fractions* of items, so that elements of $used$ can be any real number $0 \leq r \leq 1$. The goal is to maximize $\sum_{i=i}^{n} used[i] \cdot v[i]$

```
$P$   ::=   ⟨all syntactic forms in IMP⟩ | ⟨the form $Q$ below⟩

  $l$: while $b$
       do $\theta :=$ value $u \in U$ such that $\Gamma[u]$ is minimized;
          $R(\theta, \Gamma, U)$
```

Figure 8. The language LIMP (P represents programs).

while ensuring that $\sum_{i=i}^{n} used[i] \cdot c[i] \leq Budget$. This algorithm is continuous in all its inputs, as we can adjust the quantities in $used$ infinitesimally to satisfy the feasibility condition even when the inputs change infinitesimally.

To see why proving this is hard, consider a program $FracKnap$ coding the algorithm (Fig. 7). Here, cur_c tracks the part of the budget yet to be spent; the algorithm greedily adds elements to $used$, compensating with a fractional choice when cur_c becomes negative. Line 5 involves *choosing* an item m such that $(v[m]/c[m])$ is maximal, and once again, we abstract this choice by nondeterminism. It is now easy to see that continuity of tot_v is not inductive; one can also see that the observations made at the end of Example 10 apply. However, one difference is that the the condition of the main loop (Line 4) here can be affected by slight changes to cur_c. Therefore, proving this program requires a more sophisticated use of the synchronized termination than what we saw before. □

5.2.1 A language of nondeterministic abstractions

Let us now develop a rule that can handle the issues raised by these examples. To express our conservative abstractions, we extend the language IMP with a syntactic form for loops with restricted nondeterministic choice. We call this extended language LIMP. Its syntax is as in Figure 8. Here:

- U is a set—the *iteration space* for the loop in the syntactic form Q. Its elements are called *choices*.
- θ is a special variable, called the *current choice variable*. Every iteration starts by picking an element of U and storing it in θ.
- Γ is a real-valued array with $Dom(\Gamma) = U$. If u is a choice, then $\Gamma[u]$ is its *weight*. The weight acts as a selection criterion for choices—iterations always select minimal-weight choices. Multiple choices can have the same weight, leading to nondeterministic execution.
- $R(\theta, \Gamma, U)$ (henceforth just R) is an IMP program that does not write to θ. It can read θ and read or update the iteration space U and the weight array Γ.

We call a program of form Q an *abstract loop*—henceforth, Q denotes an arbitrary, fixed abstract loop. For simplicity, we only consider the analysis of abstract loops—an extension to all LIMP programs is easy. Also, we restrict ourselves to programs that terminate on all inputs.

The main loops in our codes in Figures 1 and 7 are abstract loops. For example, the workset WL, the node w, and the array d in Figure 1 respectively correspond to the iteration space U, the choice variable θ, and the map Γ of choice weights. While the weight array is not an explicit variable in Figure 7, it can be added as an auxiliary variable. by a simple instrumentation routine.

The functional semantics of Q is defined in a standard way. Due to nondeterminism, Q may have multiple executions from a state; consequently, $[\![Q]\!]$ comprises mappings of the type $\sigma \mapsto \Sigma$, where σ is a state, and Σ the set of states at which Q may terminate on starting from σ. We skip the detailed inductive definition.

Continuity. Continuity is defined in the style of Def. 3: Q is *continuous* at a state σ_0 in a set In of input variables and a set Obs of observable variables if for all $\epsilon \in \mathbb{R}^+$, there is a $\delta \in R^+$ such that for all σ_0' satisfying $Pert_{\delta,In}(\sigma_0, \sigma_0')$, all $\sigma_1 \in [\![Q]\!](\sigma_0)$, and all $\sigma_1' \in [\![Q]\!](\sigma_0')$, we have $\sigma_0' \approx_{\epsilon,Obs} \sigma_1'$. Note that if Q is continuous, then for states σ_0, σ_1, and σ_2 such that $\{\sigma_1, \sigma_2\} \subseteq [\![Q]\!](\sigma_0)$, we must have have $\sigma_1 \equiv_{Obs} \sigma_2$. Thus, though Q uses a choice construct, its behavior is not really nondeterministic.

Trace semantics and invariants. Due to nondeterminism, the trace semantics for abstract loops is richer than that for IMP. Let us denote the body of the top-level loop of Q by B. For $u \in U$, let the *parameterized iteration* B_u be the program ($\theta := u;\ R$) that represents an iteration of the loop with θ set to u. For states σ, σ', we say that there is a *u-labeled transition* from σ to σ', and write $\sigma \xrightarrow{u} \sigma'$, if (1) at the state σ, $\Gamma[u]$ is a minimal weight in the array Γ; and (2) the Hoare triple $\{\sigma\}B_u\{\sigma'\}$ holds.

Intuitively, σ and σ' are states at the loop header (label l) in successive iterations. Condition (1) asserts that u is a permissible choice for Q at σ. Condition (2) says that assuming u is the chosen value for θ in a loop iteration, σ' is the state at its end. Note that our transition system is *deterministic*—i.e., for fixed σ and u, there is at most one σ' such that $\sigma \xrightarrow{u} \sigma'$.

Let ρ, ρ' be nonempty sequences over U, and let $u \in U$. We say that there is a ρ-labeled transition from σ to σ' if one of the following conditions holds:

- $\rho = u$ and $\sigma \xrightarrow{u} \sigma'$,
- $\rho = u.\rho'$, and there exists a state σ'' such that: (1) $\sigma \xrightarrow{u} \sigma''$, (2) σ'' satisfies the loop condition b, and (3) $\sigma'' \xrightarrow{\rho'} \sigma'$.

A *trace* of Q from a state σ_{in} is now defined as a sequence $\pi = \sigma_0 \xrightarrow{\rho_0} \sigma_1 \xrightarrow{\rho_1} \dots \sigma_n \xrightarrow{\rho_n} \sigma_{n+1}$, where $\sigma_0 = \sigma_{in}$, and for each $0 \leq i \leq n$, $\sigma_i \xrightarrow{\rho_i} \sigma_{i+1}$ and σ_i satisfies b.

Here, the transition $\sigma_i \xrightarrow{\rho_i} \sigma_{i+1}$ represents a *sequence of loop iterations* leading Q from σ_i to σ_{i+1}. Note that σ_{n+1} may not satisfy b—if it does not, then it is the terminal state of Q. If each ρ_i is of the form $u_i \in U$, then π is said to be a *U-trace*.

Clearly, Q can have multiple traces from a given state. At the same time, if $\rho = u_0 u_1 \dots u_m$ and there is a transition $\sigma_0 \xrightarrow{\rho} \sigma_{m+1}$, then Q has a *unique* U-trace of the form $\sigma_0 \xrightarrow{u_0} \sigma_1 \dots \xrightarrow{u_m} \sigma_{m+1}$. We denote this trace by $Expose(\sigma_0 \xrightarrow{\rho} \sigma_{m+1})$.

For an initial condition c, a state σ is reachable from c if it appears in *some* trace from a state satisfying c. A transition $\sigma \xrightarrow{\rho} \sigma'$ is reachable from c if σ is reachable from c. The loop invariant $\mathcal{I}(c)$ of Q is an overapproximation of the set of states reachable from c.

5.2.2 The analysis

Now we present our continuity analysis. As in Section 5, our goal is to obtain a continuity judgment $c \vdash Cont(Q, X, X)$, where X is an inductively obtained set of variables. As hinted at in Example 10, we perform induction over clusters of successive loop iterations parameterized by choices of equal weight. We call these clusters *epochs*. Pleasantly, while the notion of epochs is crucial for our soundness argument, it is invisible to the user of the rule, who discharges verification conditions just as before.

Verification conditions and rule definition. We start by defining our rule and its verification conditions. Once again, we discharge the conditions of synchronized termination and separability (these are defined as before). In addition, we discharge the conditions of Γ*-monotonicity* and *commutativity*.

The former property asserts that the weight of a choice does not increase during executions of Q. Formally, the program Q is Γ-*monotonic* under the initial condition c if for all states $\sigma, \sigma' \in \mathcal{I}(c)$ such that there is a transition from σ to σ', we have $\sigma(\Gamma[v]) \geq \sigma'(\Gamma[v])$ for all $v \in U$.

The second condition says that parameterized iterations can be *commuted*. Let us define:

$$\begin{tikzcd}
\sigma_0 \arrow[r, "u"] \arrow[d, phantom, "\equiv_{Obs}"] & \sigma_1 \arrow[r, "v"] & \sigma_2 \arrow[d, phantom, "\equiv_{Obs}"] \\
\sigma'_0 \arrow[r, "v"] & \sigma'_1 \arrow[r, "u"] & \sigma'_2
\end{tikzcd}
\qquad
\begin{array}{c}
\sigma_0(\Gamma[u]) = \sigma'_0(\Gamma[v]) = \\
\sigma_1(\Gamma[v]) = \sigma'_1(\Gamma[u])
\end{array}$$

Figure 9. Commutativity

Definition 6 (Commutativity)**.** The parameterized iterations B_u and B_v *commute* under the initial condition c and the set Obs of observable variables if for all states σ_0, σ'_0, σ_1, σ_2 such that: (1) $\sigma_0 \equiv_{Obs} \sigma'_0$; (2) σ_0, σ_1, and σ'_0 satisfy the loop invariant $\mathcal{I}(c)$; (3) $\{\sigma_0\}B_u\{\sigma_1\}$ and $\{\sigma_1\}B_v\{\sigma_2\}$; and (4) $\sigma_0(\Gamma(u)) = \sigma'_0(\Gamma(v)) = \sigma_1(\Gamma(v))$, there are states σ'_1 and σ'_2 such that

- $\{\sigma'_0\}B_v\{\sigma'_1\}$, $\{\sigma'_1\}B_u\{\sigma'_2\}$, and σ'_1 satisfies $\mathcal{I}(c)$
- $\sigma'_1(\Gamma(u)) = \sigma'_0(\Gamma(v))$
- $\sigma_2 \equiv_{Obs} \sigma'_2$.

The program Q is *commutative* under c and the set Obs of variables (written as $c \vdash Comm(Q, Obs)$) if for all u, v, B_u and B_v commute under c.

A *commutation diagram* capturing the relationship between σ_0, σ_1, etc. in the above definition is given in Fig. 9. Note that given σ'_0, u, and v, the states σ'_1 and σ'_2 are unique. Also note that the property defined here is stronger than commutativity in the usual sense, as it asserts properties of weights of choices.

Our proof rule for abstract loops is now presented in Fig. 10. Intuitively, the rule performs induction over sequences of epochs. As we mentioned in Example 10, small perturbations will reorder loop iterations within an epoch; however, a subtle implication of our premises is that such reorderings do not affect continuity at the end of each epoch. Before presenting a soundness argument and defining epochs formally, let us apply the rule to our two examples.

Example 12 (Dijkstra's algorithm). Let us now revisit our implementation *Dijk* of Dijkstra's algorithm, and derive the continuity judgment $true \vdash Cont(Dijk, X, X)$, where $X = \{G, d, WL\}$ (this can be subsequently weakened to judgments like $true \vdash Cont(Dijk, \{G\}, \{d\})$). Here, the array d corresponds to Γ in the syntax of LIMP, and lines 6-10 correspond to the program R.

First, we observe that *Dijk* is d-monotonic (and that the reasoning establishing this is simple). Also, X-separability is obvious. As in case of the Floyd-Warshall algorithm, synchronized termination holds as the loop condition, only involving a discrete variable, is unconditionally continuous in the set of input variables X. Finally, we observe that lines 6-10 are also commutative by our definition. By the rule LOOP, the desired continuity judgment follows. □

Example 13 (Fractional Knapsack). Now we consider the program *FracKnap* (Fig. 7), recast as a LIMP program using an auxiliary array Γ such that at the beginning of each loop iteration, we have $\Gamma[i] = c[i]/v[i]$. Let us verify the judgment $true \vdash Cont(FracKnap, X, X)$, where $X = \{\Gamma, Items, cur_c, tot_v, c, v\}$. Once again, separability of X is obvious, and Γ-monotonicity and commutativity can be verified with some effort. The synchronized termination condition, however, is more interesting that in the proof of *Dijk*, as the loop condition $(cur_c > 0)$ is not always continuous in X. To see that the condition holds, let c be the formula $(cur_c = 0)$ capturing the set of states where the loop condition is discontinuous. Under this condition, Lines 6–10, taken together, are equivalent to a **skip**-statement. Therefore, we have $true \vdash Term(FracKnap, X)$. By the rule LOOP, we have $true \vdash Cont(FracKnap, X, X)$. □

Soundness. Now we sketch an argument for the soundness of the rule LOOP. Let us start by defining epochs formally:

$$\text{(Loop)}\ \frac{\begin{array}{c} U,\Gamma \in X \quad \mathcal{I}(c) \vdash \mathit{Comm}(Q,X) \quad Q \text{ is } \Gamma\text{-monotonic under } c \\ c \vdash \mathit{Sep}(Q,X) \quad \mathcal{I}(c) \vdash \mathit{Cont}(R,X,X) \quad c \vdash \mathit{Term}(Q,X) \end{array}}{c \vdash \mathit{Cont}(Q,X,X)}$$

Figure 10. Proof rule LOOP for programs with loops (Q is an abstract loop, and $S \subseteq \mathit{Obs}_Q$)

Definition 7 (Epochs)**.** Consider a transition $\eta = \sigma_0 \xrightarrow{\rho} \sigma_{m+1}$, with $\mathit{Expose}(\sigma_0 \xrightarrow{\rho} \sigma_{m+1}) = \sigma_0 \xrightarrow{u_0} \sigma_1 \ldots \xrightarrow{u_m} \sigma_{m+1}$. The transition η is an *epoch* if:

1. For all $0 \leq j < m$, we have $\sigma_j(\Gamma[u_j]) = \sigma_{j+1}(\Gamma[u_{j+1}])$.
2. Q has no transition $\sigma_{m+1} \xrightarrow{u_{m+1}} \sigma_{m+2}$ such that $\sigma_m(\Gamma[u_m]) = \sigma_{m+1}(\Gamma[u_{m+1}])$. The epoch is said to have *weight* $\sigma_0(\Gamma(u_0))$.

Intuitively, an epoch is a *maximal* sequence of iterations that agree on choice-weights. For our proofs, we also need a notion of δ*-epochs*, which are just like epochs, except they allow a margin of error δ between the weights of the choices made in successive iterations. Formally, for $\delta \in \mathbb{R}^+$, a transition η as in Definition 7 is a δ*-epoch* of Q if for some $W \in \mathbb{R}$, we have:

1. For all $0 \leq j < m$, $|\sigma_j(\Gamma(u_j)) - W| < \delta$.
2. There is no transition $\sigma_{m+1} \xrightarrow{u_{m+1}} \sigma_{m+2}$ in Q such that $|\sigma_{m+1}(\Gamma(u_{m+1})) - W| < \delta$.

Note that every U-trace $\pi = \sigma_0 \xrightarrow{u_0} \sigma_1 \ldots \xrightarrow{u_m} \sigma_{m+1}$ corresponds to a *unique* trace $\mathit{Epochize}(\pi) = \sigma'_0 \xrightarrow{\rho_0} \sigma'_1 \ldots \xrightarrow{\rho_n} \sigma'_{n+1}$ such that $\sigma_0 = \sigma'_0$, $\sigma_{m+1} = \sigma'_{n+1}$, and for each i, $\sigma'_i \xrightarrow{\rho_i} \sigma'_{i+1}$ is an epoch. This trace represents the breakdown of π into epochs. For $\delta \in \mathbb{R}^+$, the trace $\mathit{Epochize}_\delta(\pi)$, representing the breakdown of π into δ-epochs, is similarly defined.

Now we define a notion of continuity for epochs.

Definition 8 (Continuity of epochs)**.** An epoch $\eta = \sigma_0 \xrightarrow{\rho} \sigma_1$ of Q is *continuous* with respect to a set In of input variables and a set Obs of observable variables if for all $\epsilon \in \mathbb{R}^+$, there exists a $\delta \in \mathbb{R}^+$ such that for all states σ'_0 satisfying $\mathit{Pert}_{\delta,\mathit{In}}(\sigma_0, \sigma'_0)$, every δ-epoch $\sigma'_0 \xrightarrow{\rho'} \sigma'_1$ satisfies the property $\sigma_1 \approx_{\epsilon,\mathit{Obs}} \sigma'_1$.

The crux of our soundness argument is that under the premises of the rule LOOP, every epoch of Q is continuous. This is established by the following theorem:

Theorem 4. *Suppose the following conditions hold for a set of variables $X \subseteq \mathit{Var}(Q)$ and an initial condition c:*

1. Q is Γ-monotonic under c	*4. $U, \Gamma \in X$*
2. $\mathcal{I}(c) \vdash \mathit{Comm}(Q,X)$	*5. $c \vdash \mathit{Sep}(Q,X)$*
3. $\mathcal{I}(c) \vdash \mathit{Cont}(R,X,X)$	*6. $c \vdash \mathit{Term}(Q,X)$*

Then every epoch of Q reachable from c is continuous in input variables X and observable variables X.

The proof involves a lemma proving the determinism of epochs:

Lemma 1. *Suppose the premises of Theorem 4 hold. Then if $\eta = \sigma_0 \xrightarrow{\rho} \sigma_1$ is an epoch reachable from c, then for all epochs $\eta' = \sigma'_0 \xrightarrow{\rho'} \sigma'_1$ such that $\sigma'_0 \equiv_X \sigma_0$, we have: (1) ρ' is a permutation of ρ; and (2) $\sigma_1 \equiv_X \sigma'_1$.*

Proof sketch. Let W be the weight of η, and define a variable U_W whose value at a state is the set of choices in U with weight W. As $U \in X$, U_W has the same value in X-equivalent states; as epochs are maximal, η terminates only when b is violated or U_W is empty.

Without loss of generality, assume that ρ and ρ' are sequences of *distinct* choices. Suppose u is the first choice in ρ that does not appear in ρ'; let $\rho = \rho_1 u \rho_2$. Now we have the following possibilities: (a) the execution of ρ_1 added the choice u to U_W by setting $\Gamma[u] = W$; (b) some iteration B_v in ρ', where $v \neq u$, removed the choice u from U_W by setting $\Gamma[u] > W$ or removing u from U; (c) at some point during the execution of ρ' before u could be selected, the loop condition b was violated.

Each of these scenarios are ruled out by our assumed conditions. We only show how to handle case (b). As Q is Γ-monotonic, we have the property that if $\Gamma[u] > W$ at some point in ρ', then $\Gamma[u] > W$ at *all prior points* in ρ'—i.e., u never had the weight W in η'. As for u being removed from U in ρ' before ever being selected, this violates commutativity.

As for postcondition (2), it follows from commutativity if ρ is of length two or more. If $\rho = u$ for some choice u, then the postcondition follows from the separability of X. □

$$\begin{tikzcd}
\sigma_0 \arrow[r, "\rho_0"] \arrow[d, phantom, "\cdot\approx_{\delta_0,X}"] & \sigma_1 \arrow[r, "\rho_1"] \arrow[d, phantom, "\cdot\approx_{\delta_1,X}"] & \sigma_2 \arrow[r, "\rho_2"] \arrow[d, phantom, "\cdot\approx_{\delta_2,X}"] & \cdots \mathit{Epochize}(\pi) \\
\sigma'_0 \arrow[r, dash, "\rho'_0"'] & \sigma'_1 \arrow[r, dash, "\rho'_1"'] & \sigma'_2 \arrow[r, dash, "\rho'_2"'] & \cdots \mathit{Epochize}_\delta(\pi')
\end{tikzcd}$$

Figure 11. Induction over epochs

(As an aside, the above implies that under the premises of the rule LOOP, epochs are *observationally deterministic*: epochs starting from X-equivalent states always end in X-equivalent states.)

Now we establish a lemma connecting each δ-epoch to an epoch to which it is "close." (The proof is quite involved—due to lack of space, we only give the intuitions here.) Consider, for sufficiently small δ, an arbitrary δ-epoch η such that $\mathit{Expose}(\eta) = \sigma_0 \xrightarrow{u_1} \sigma_1 \ldots \xrightarrow{u_n} \sigma_{n+1}$ and a state σ'_0 such that $\sigma'_0 \approx_{\delta,X} \sigma_0$. As η is a δ-epoch, it is possible to perturb every state appearing in $\mathit{Expose}(\eta)$ by an amount less than δ to get a U-trace π such that: (1) π starts with σ'_0; and (2) if σ'_i is the i-th state in π, then for all i, we have $\sigma'_i(\Gamma[u_i]) = \sigma'_{i+1}(\Gamma[u_{i+1}])$. We can now show that, if the premises of Theorem 4 hold, then this trace can be executed by Q and, in fact, is of the form $\mathit{Expose}(\eta')$ for some epoch η' of Q. Thus we have:

Lemma 2. *Assume that the premises of Theorem 4 hold. Then for all $\epsilon \in \mathbb{R}^+$, there exists a $\delta \in \mathbb{R}^+$ such that for all δ-epochs $\eta = \sigma_0 \xrightarrow{\rho} \sigma_1$ and all states σ'_0 such that $\sigma'_0 \approx_{\delta,X} \sigma_0$, there is an epoch $\eta' = \sigma'_0 \xrightarrow{\rho'} \sigma'_1$ such that: (1) $\rho = \rho'$, and (2) $\sigma_1 \approx_{\epsilon,X} \sigma'_1$.*

Proof sketch for Theorem 4. Now we can establish Theorem 4. Let $\eta = \sigma_0 \xrightarrow{\rho} \sigma_1$ be any epoch, and let $\epsilon \in \mathbb{R}^+$. Select a δ small enough for Lemma 2 to hold. Consider any δ-epoch $\eta' = \sigma'_0 \xrightarrow{\rho'} \sigma'_1$ such that $\sigma_0 \approx_{\delta,X} \sigma'_0$. By Lemma 2, there is an epoch $\eta'' = \sigma_0 \xrightarrow{\rho'} \sigma''_1$ such that $\sigma''_1 \approx_{\epsilon,X} \sigma'_1$. As η and η'' are epochs from the same state, by Lemma 1, we have $\sigma_1 \equiv_X \sigma''_1$. But this means that $\sigma_1 \approx_{\epsilon,X} \sigma'_1$. This establishes the continuity of η. □

Soundness for rule LOOP now follows in a straightforward way. The argument is similar to that for Theorem 3—however, this time we use epochs, rather than individual loop iterations, as the basic steps of induction. While continuity may be broken inside an epoch, it is, by Theorem 4, reinstated at its end. Intuitively, any two traces of Q starting from arbitrarily close states "synchronize"—reaching observationally close states—at the ends of epochs (the situation is sketched in Figure 11). We have:

Theorem 5. *The proof rule* LOOP *is sound.*

Proof sketch. Consider an arbitrary U-trace π of Q starting from a state σ_0 that satisfies c, the trace $\mathit{Epochize}(\pi) = \sigma_0 \xrightarrow{\rho_0} \ldots \xrightarrow{\rho_m} \sigma_{m+1}$, and any $\epsilon \in \mathbb{R}^+$. Select a sequence of δ_i's, with $\delta = \delta_0$, just as in Theorem 3, and consider a state σ'_0 such that $\sigma_0 \equiv_{\delta,X} \sigma'_0$.

Example	Time	Simple-loop or No Loop or $(U, \Gamma(u), \theta)$	Term.	Expressions
BubbleSort	0.035	Simple-loop	No	
InsertionSort (Outer)	0.028	Simple-loop	Yes	
InsertionSort (Inner)	0.027	Simple-loop	Yes	`Update`(A,j+1,z)
SelectionSort (Outer)	0.092	Simple-loop	No	A[s]
SelectionSort (Inner)	0.249	Simple-loop	No	$A[1..i]$, `Array2Set` $(A[i \mathrel{..} n-1])$
MergeSort	0.739	$(\{(u, u') \mid 1 \leq u \leq n, 1 \leq u' \leq m\}, \texttt{Min}(A[u], B[u']), (i, j))$	No	
Dijkstra	0.177	$(Q, \texttt{dist}[u], m)$	No	
Bellman-Ford	0.029	Simple-loop	No	
Floyd-Warshall	0.032	Simple-loop	No	
Kruskal	1.069	$(Q = Edges(G), W(u, u'), (i, j))$	No	
Prim	0.248	$(\{u, u' \mid u \in G - F, u' \in F\}, W(u, u'), (v, v'))$	No	
Frac. Knapsack	0.38	$(\{1 \mathrel{..} n\}, v[u]/c[u], m)$	Yes	
Int. Knapsack	3.22	No Loop	No	

Table 1. Benchmark Examples

Let π' be any U-trace from σ'_0, and let $Epochize_\gamma(\pi') = \sigma'_0 \xrightarrow{\rho'_0} \ldots \xrightarrow{\rho'_n} \sigma'_{n+1}$ (without loss of generality, assume that $n \leq m$). By the continuity of epochs we have $\sigma_1 \approx_{\delta_1, X} \sigma'_1$. Generalizing inductively, and using the synchronized termination condition as before, we conclude that $\sigma_{m+1} \approx_{\epsilon, X} \sigma'_{n+1}$. As π, π' are arbitrary traces, Q is continuous. $\square$

6. Experiments

We chose several classic continuous algorithms (mostly from a standard undergraduate text on algorithms [3]) to empirically evaluate the precision and completeness of our proof rules. Our rules were able to prove the continuity of 11/13 examples that we tried.

An important step before the application of our proof rules LOOP is the transformation of loops into abstract loops as described in Section 5.2, which requires identifying the iteration space U, the current choice variable θ, and the weight function Γ. Of course, if the rule SIMPLE-LOOP is applicable, then these steps are not needed. Table 1 describes the rules, and parameters U, θ, and Γ, needed in each of our applications. In some cases, we also needed to introduce some auxiliary variables since our framework tracks continuity of program fragments with respect to a set of observation variables. (An alternative would have been to define our framework to track continuity of expressions, and in fact, this is another interesting aspect of continuity proofs for programs. Such an extension to our framework is not difficult, but we avoided this to keep the presentation of the framework simpler.) The column "Expressions" contains the expressions represented by auxiliary variables in the various examples. Transformation of loops into abstract loops and introduction of auxiliary variables were performed manually. However, there are heuristics that can be used to automate this step.

Sorting Algorithms. Consider a sorting algorithm that takes in an array A_{in} and returns a sorted array A_{out}. Such an algorithm is continuous in A_{in}—a small change to A_{in} can only result in a small change to $A_{out}[i]$, for all i. The observation requires a bit of thought as the *position* of a given element in the output array A_{out} may change completely on small perturbations. Indeed, consider an algorithm that returns a representation of the sorted array in the form of an array of indices In into the original input array (i.e., $i < j \Rightarrow A_{in}[In[i]] < A_{in}[In[j]]$), rather than a sort of the original array. Such an algorithm is discontinuous in A_{in}.

Our proof rules can establish continuity of the three standard iterative sorting algorithms: BubbleSort, InsertionSort, SelectionSort. Our proof rules can also establish continuity for MergeSort, but are unable to establish the continuity of Quicksort. This is because the continuity proof for MergeSort is inductive with respect to the recursive calls (i.e., continuity holds at every recursive call to MergeSort), but this is not the case with QuickSort. This is not unexpected since we have not addressed the interprocedural variant of continuity analysis in this paper. The proof of continuity for each of the sorting algorithms turns out to be quite different from each other (suggesting the fundamentally different ways in which these algorithms operate). We point out some of the interesting aspects of the continuity proofs for each of these examples.

Bubblesort is the simplest of all where the continuity proof is inductive for both its loops, and the interesting part is to establish the continuity of the loop-body. This involves proving that the `Swap` operation that swaps two elements of an array is continuous, which requires an application of proof rule ITE-1.

The proof of continuity of InsertionSort is also inductive for both its loops. However, establishing continuity of the inner loop has two interesting aspects. It requires an application of the synchronized termination condition, and requires establishing continuity of (the auxiliary variable representing) the expression $\texttt{update}(A, j+1, z)$ (note that the loop is actually discontinuous in A, but to inductively prove the continuity of the outer loop, we in fact need to prove continuity of the inner loop with respect to $\texttt{update}(A, j+1, z)$).

The proof of continuity of SelectionSort is also inductive for both its loops. The interesting part is to note that the outer loop is not continuous in A. It is actually continuous in the expressions $A[1..i]$ and the set $\texttt{Array2Set}(A[i, .., n-1])$, which suffices to establish the continuity of $A[1..n]$ when the loop terminates since $i = n$ outside the loop. Similarly, the inner loop is continuous in $A[s]$ as opposed to s, but this suffices to prove the desired continuity property of the outer loop.

For MergeSort, the challenging part is to establish the continuity of the Merge procedure, which is not inductive, and requires using the proof rule LOOP in its generality.

Shortest Path Algorithms. The path returned by any shortest path algorithm is susceptible to small perturbations in the edge weights of the input graph. However, the value of the shortest path returned by the shortest path algorithms is actually continuous in the input graph. Our proof rules can establish this property for each of the three shortest path algorithms that we considered. Among these, Dijkstra's algorithm is the most interesting one, requiring use of the proof rule LOOP in its generality. The continuity proof of Bellman-Ford and Floyd Warshall is relatively easy since it is inductive.

Minimum Spanning Tree Algorithms. The spanning tree returned by minimum spanning tree algorithms can vary widely upon small perturbations in the edge weights of the input graph. However, the weight of the minimum spanning tree returned by the minimum spanning tree algorithms is actually continuous in the edge weights of the input graph. Our proof rules can establish this property for Kruskal and Prim algorithms, but fail for Boruvka's algorithm. The continuity proofs for both Kruskal and Prim algorithms are not inductive and require an application of the proof rule LOOP in its generality. However, our proof rules are not precise enough to establish the continuity of Boruvka's algorithm.

Knapsack Algorithms. The integer-knapsack algorithm takes as input a weight array c and a value array v containing the weight and value respectively of various objects, and a knapsack capacity $Budget$ and returns the set of items with maximum combined value tot_v such that their combined weight is less than the knapsack capacity. The value of tot_v is discontinuous in c and $Budget$ since small perturbations may make an object now no longer fit in the knapsack (or the other way round). However, it is interesting to

```
KRUSKAL($G$ : graph)
for each node $v$ in $G$ do $C[v] := \{v\}$;
$Q :=$ set of all edges in G; $cost := 0$; $T := \emptyset$;
while $Q \neq \emptyset$
        do choose edge $(v, w) \in Q$
           such that $G(v, w)$ is minimal;
remove $(v, w)$ from $Q$;
if $C[v] \neq C[w]$
    then add edge $(v, w)$ to $T$;
          $cost := cost + G(v, w)$;
          $C[v] := C[w] := C[v] \cup C[w]$;
```

```
PRIM($G$ : graph)
for each node $v$ in $G$
     do $d[v] := \perp$; $parent[v] :=$ UNDEF;
$s :=$ arbitrary node in G; $d[s] := 0$;
$cost := 0$; $F := \{s\}$;
while $|F| < |G|$
     do choose node $v \notin F$
                with a minimal-cost edge $(v, v')$ into $F$
        $F := F \cup \{v\}$; $cost := cost + G(v, v')$;
        for each neighbor $w$ of $v$
             do if $d[w] > d[v] + G(v, w)$;
                  then $d[w] := d[v] + G(v, w)$;
                       $parent[w] := v$;
```

```
MERGESORT($A$ : realarr)
if $|A| \leq 1$
   then return A;
$m := \lfloor |A|/2 \rfloor$;
$A_1 := A[0\,..\,m]$; $A_2 := A[m+1\,..\,|A|-1]$;
$B_1 :=$ MERGESORT($A_1$);
$B_2 :=$ MERGESORT($A_2$);
return MERGE($B_2$, $B_2$);
```

```
MERGE($A_1$ : realarr, $A_2$ : realarr)
$i := 0$; $j := 0$; $k := 0$;
while $k < |A_1| + |A_2|$
      do if ($i \geq |A_1|$) or ($A_1[i] > A_2[j]$)
           then $result[k] := A_2[j]$;
                 $j := j + 1$; $k := k + 1$;
           else $result[k] := A_1[i]$;
                 $i := i + 1$; $k := k + 1$;
return $result$;
```

```
BELLMAN-FORD($G$ : graph, $src$ : node)
for each node $v$ in $G$
     do $d[v] := \perp$; $parent[v] :=$ UNDEF;
$d[src] := 0$;
for each node in $G$
     do for each edge $(v, w)$ of G
           do if $d[v] + G(v, w) < d[w]$
                then $d[w] := d[v] + G(v, w)$;
                     $parent[w] := v$;
```

```
KNAPSACK($v$ : realarr, $c$ : realarr, $j$ : int, $W$ : real, )
if $j = 0$ then return 0
   else if $W = 0$ then return 0
   else if $(c[j] > W)$ then return
        KNAPSACK($v, c, j-1, W$)
   else $z_1 :=$ KNAPSACK($v, c, j-1, W$);
        $z_2 :=$ KNAPSACK($v, c, j-1, W - c[j]$);
        return $\max\{z_1, z_2\}$
```

```
INSERTION-SORT($A$ : realarr)
for $i := 1$ to $(|A| - 1)$
     do $z := A[i]$; $j := i - 1$;
         while $j \geq 0$ and $A[j] > z$
              do $A[j+1] := A[j]$; $j := j - 1$;
         $A[j+1] := z$;
```

```
SELECTION-SORT($A$ : realarr)
for $i := 1$ to $(|A| - 1)$
     do $s := i$;
         for $j := i + 1$ to $(|A| - 1)$
              do if $(A[j] < A[s])$ $s := j$;
         swap($A[i]$, $A[s]$);
```

```
BUBBLE-SORT($A$ : realarr)
for $i := 1$ to $(|A| - 1)$;
     do for $j := 1$ to $(|A| - 1)$;
            do if $(A[i] > A[i+1])$
                 then swap($A[i]$, $A[i+1]$);
```

Figure 12. Pseudocode for experiments

note that tot_v is actually continuous in v. Our proof-rules are able to establish this property inductively across the different recursive calls of Knapsack after proving that continuity of the recursion-free part, which requires an application of proof rule ITE-1. As for fractional knapsack, it was proved as in Example 13.

Implementation and Experimental Setup. The method has been implemented in C# relying on the Z3 SMT solver to discharge proof obligations and the Phoenix Compiler Framework to process the input program. The analysis is implemented as a fixpoint computation to find the solution of dataflow equations derived from the proof rules, where some proof obligations are submitted to the SMT-solver in the process. The SMT-solver is used by only some of the proof rules—e.g., the rule ITE-1 requires the proof of equivalence of branches at the discontinuities of the condition variable, and the use of a continuity specification for, say, division requires us to determine if the divisor is non-zero. Commutativity and early termination proofs, required to prove continuity of loops, are also submitted to the SMT-solver. As mentioned earlier, we manually rewrote some of the programs to fit the abstraction language LIMP. Also, a few examples involved a nested loop inside the abstract loop. In these, we hand-wrote an invariant for the inner loop (however, this step can be automated using more powerful invariant generators). The performance results reported in table 1 were obtained on a Core2 Duo 2.53 Ghz with 4GB of RAM.

7. Conclusion and future work

We have presented a program analysis to automatically determine if a program implements a *continuous* function. The practical motivation is the verification of *robustness* properties of programs whose inputs can have small amounts of error and uncertainty.

This work is the first in a planned series of papers on the analysis of *robustness and stability of programs*, and its applications in the verification of software running on cyber-physical systems [12]. In particular, we plan to explore the following questions:

Quantitative analysis. Rather than knowing whether a program is continuous, what if we want *bounds* on changes to outputs on small changes to inputs? We plan to answer this question by developing a *quantitative continuity analysis*. Such an analysis will be closely related to the problem of differentiating or finite-differencing programs.

Safe handling of discontinuities. Many practical programs are discontinuous but still safe. For example, a controller that is otherwise continuous might switch itself off—discontinuously—when it changes mode. How do we reason about robustness in such settings?

A possible solution is to allow a *specification* for how the program should behave at discontinuities. A robustness proof now establishes that at every input state, the program is either continuous, or follows the specification.

Counterexample generation. Can we generate inputs that, when changed slightly, cause large changes in the program's behavior?

Modular analysis. What about interprocedural continuity analysis, hinted at in Sec. 3?

Applications outside robustness. Does our proof methodology have applications in a contexts outside of robustness? In particular, our proof rule LOOP establishes the observational determinism of non-deterministic abstractions expressible in the language LIMP. Can it be used in determinism proofs for *concurrent programs* [22]?

Stability. Can we extend the techniques here to do program analysis with respect to control-theoretic stability properties—e.g., asymptotic and Lyapunov stability [16, 17]?

References

[1] Rajeev Alur, Costas Courcoubetis, Thomas A. Henzinger, and Pei-Hsin Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems*, 1992.

[2] Yamine Aït Ameur, Gérard Bel, Frédéric Boniol, S. Pairault, and Virginie Wiels. Robustness analysis of avionics embedded systems. In *LCTES*, pages 123–132, 2003.

[3] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to algorithms.* MIT Press and McGraw-Hill, 1990.

[4] Patrick Cousot. Proving the absence of run-time errors in safety-critical avionics code. In *EMSOFT*, pages 7–9, 2007.

[5] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTREÉ analyzer. In *ESOP*, pages 21–30, 2005.

[6] Eric Goubault. Static analyses of the precision of floating-point operations. In *SAS*, pages 234–259, 2001.

[7] Eric Goubault, Matthieu Martel, and Sylvie Putot. Asserting the precision of floating-point computations: A simple abstract interpreter. In *ESOP*, 2002.

[8] Joseph Halpern. *Reasoning about uncertainty*. The MIT Press, 2003.

[9] Dick Hamlet. Continuity in sofware systems. In *ISSTA*, pages 196–200, 2002.

[10] Mats Per Erik Heimdahl, Yunja Choi, and Michael W. Whalen. Deviation analysis: A new use of model checking. *Autom. Softw. Eng.*, 12(3):321–347, 2005.

[11] Myron Kayton and Walter R. Fried. *Avionics navigation systems*. Wiley-IEEE, 1997.

[12] Edward A. Lee. Cyber physical systems: Design challenges. In *ISORC*, pages 363–369, 2008.

[13] Matthieu Martel. Propagation of roundoff errors in finite precision computations: A semantics approach. In *ESOP*, pages 194–208, 2002.

[14] Antoine Miné. Relational abstract domains for the detection of floating-point run-time errors. In *ESOP*, pages 3–17, 2004.

[15] Bradford Parkinson and James Spiker. *The global positioning system: Theory and applications (Volume II)*. AIAA, 1996.

[16] Stefan Pettersson and Bengt Lennartson. Stability and robustness for hybrid systems. In *Decision and Control*, volume 2, pages 1202–1207, Dec 1996.

[17] Andreas Podelski and Silke Wagner. Model checking of hybrid systems: From reachability towards stability. In *HSCC*, pages 507–521, 2006.

[18] Mardavij Roozbehani, Alexandre Megretski, Emilio Frazzoli, and Eric Feron. Distributed lyapunov functions in analysis of graph models of software. In *HSCC*, pages 443–456, 2008.

[19] Ofer Strichman. Regression verification: Proving the equivalence of similar programs. In *CAV*, 2009.

[20] Wilson Sutherland. *Introduction to metric and topological spaces*. Oxford University Press, 1975.

[21] John Taylor. *An introduction to error analysis: the study of uncertainties in physical measurements*. University Science Books, 1997.

[22] Tachio Terauchi and Alex Aiken. A capability calculus for concurrency and determinism. In *CONCUR*, pages 218–232, 2006.

[23] Glynn Winskel. *The formal semantics of programming languages*. The MIT Press, 1993.