Consistency Analysis of Decision-Making Programs *

Swarat Chaudhuri
Rice University
swarat@rice.edu

Azadeh Farzan
University of Toronto
azadeh@cs.toronto.edu

Zachary Kincaid
University of Toronto
zkincaid@cs.toronto.edu

Abstract

Applications in many areas of computing make discrete decisions under *uncertainty*, for reasons such as limited numerical precision in calculations and errors in sensor-derived inputs. As a result, individual decisions made by such programs may be nondeterministic, and lead to contradictory decisions at different points of an execution. This means that an otherwise correct program may execute along paths, that it would not follow under its ideal semantics, violating essential program invariants on the way. A program is said to be *consistent* if it does not suffer from this problem despite uncertainty in decisions.

In this paper, we present a sound, automatic program analysis for verifying that a program is consistent in this sense. Our analysis proves that each decision made along a program execution is consistent with the decisions made earlier in the execution. The proof is done by generating an invariant that abstracts the set of all decisions made along executions that end at a program location l, then verifying, using a fixpoint constraint-solver, that no contradiction can be derived when these decisions are combined with new decisions made at l.

We evaluate our analysis on a collection of programs implementing algorithms in *computational geometry*. Consistency is known to be a critical, frequently-violated, and thoroughly studied correctness property in geometry, but ours is the first attempt at automated verification of consistency of geometric algorithms. Our benchmark suite consists of implementations of convex hull computation, triangulation, and point location algorithms. On almost all examples that are not consistent (with two exceptions), our analysis is able to verify consistency within a few minutes.

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]: Correctness proofs; F.3.2 [*Semantics of Programming Languages*]: Program analysis; F.2.2 [*Computational Geometry and Object Modelling*]: Geometric algorithms, languages, and systems; G.4 [*Mathematical Software*]: Reliability and robustness

Keywords Program Analysis; Uncertainty; Consistency; Geometry; Robustness

* This research was partially supported by NSF Award #1156059 and an NSERC Discover Grant.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

POPL '14, January 22–24, 2014, San Diego, CA, USA.
Copyright © 2014 ACM 978-1-4503-2544-8/14/01...$15.00.
http://dx.doi.org/10.1145/2535838.2535858

$$\begin{array}{l}\textbf{if } (\texttt{Left}(p,q)) \textbf{ then } L_1 \textbf{ else } L_4;\\ \textbf{if } (\texttt{Left}(q,r)) \textbf{ then } L_2 \textbf{ else } L_5;\\ \textbf{if } (\texttt{Left}(p,r)) \textbf{ then } L_3 \textbf{ else } L_6\end{array}$$

Figure 1. A possibly-inconsistent code snippet

1. Introduction

The problem of making decisions under uncertainty has fascinated computer scientists for a long time [13]. Abstractly, a decision-making process is a program that gathers information about an external world using a set of queries, and based on this information, makes changes to its discrete internal state. An issue with such programs is that the information that they gather from the world may be *uncertain*, and this uncertainty may affect program's decisions. Even a single "wrong" decision may lead the program down code paths that were never meant to be followed, leading to unforeseen errors.

For example, consider a program that queries the world for the relative positions of three points p, q, and r on a line. Let $\texttt{Left}(u, v)$ denote the *decision* by the program that u lies to the left of v, and $\mathit{Left}(u, v)$ the *fact* that u is actually to the left of v. Now suppose the program makes the decisions $\texttt{Left}(p, q)$ and $\texttt{Left}(q, r)$. In this case, the program should also decide that $\texttt{Left}(p, r)$, as the laws of geometry demand that

$$(\mathit{Left}(p, q) \wedge \mathit{Left}(q, r)) \implies \mathit{Left}(p, r). \quad (1)$$

However, uncertainty creates a difference between what ought to be and what is. Different queries to the world can be independently uncertain, and if p, q and r are close enough to be within the margin of error, the program may very well decide from the final query that "not $\texttt{Left}(p, r)$" holds. Suppose now that the program in question looks like the one in Fig. 1. Then, in the above scenario, it will execute the code $\texttt{L}_1; \texttt{L}_2; \texttt{L}_6$, a sequence that would not be executed *under any input* in the program's ideal semantics, where the laws of geometry hold. Since algorithm design often does not consider this scenario, the consequence could be a serious error.

A decision-making program is *consistent* if it is immune from errors like the one described above. Abstractly, let $\mathcal{R}$ be an axiomatic specification of the "laws" that we know to hold in a world Θ (e.g. Eqn. 1). A program is consistent if it does not contradict any axioms in $\mathcal{R}$ no matter what answers it receives as the result of its queries to the world Θ.

Consistency is an essential concern in applications where programs make boolean decisions under uncertainty. It is, for example, a thoroughly studied problem in computational geometry [7, 15, 16, 21, 26], where programs make decisions about relative positions of objects in continuous geometric spaces[1]. Often, these positions can

[1] In the geometry literature, consistency is known as *robustness*. We avoid the latter term given its recent use [4, 5, 19] in the program verification literature to denote an entirely different concept.

```
1   if Left(p, q)
2     then L_1;
3          if Left(q, r)
4            then L_2; L_3
5            else L_5; if Left(p, r) then L_3 else L_6
6     else
7          L_4;
8          if Left(q, r)
9            then L_2; if Left(p, r) then L_3 else L_6
10           else L_5; L_6
```

Figure 2. Consistent variant of code in Fig. 1.

only be known with limited degree of certainty and up to a limited level of accuracy. For instance, a robot doing geometric reasoning to detect collisions with other objects must base its discrete decisions on uncertain sensor data. Even on a standalone computer, the outcomes of discrete queries about geometric datasets are typically uncertain due to numerical inaccuracies [22]. It is well-known that inconsistency between such decisions can cause everyday geometric algorithms to crash, go into infinite loops, and violate essential postconditions [21].

At the same time, consistency in our sense is not easily enforced through usual software engineering practices. It is difficult to *test* a program for consistency since, by definition, errors due to uncertainty may not be reproducible. Furthermore, as software is typically designed without uncertainty in mind, consistency is usually outside the scope of manual reasoning about algorithms. For example, most algorithm designers *assume* the axioms of geometry to hold to reason about correctness of their designs, and would not imagine situations where they might break down when these axioms are contradicted. Finally, reasoning about consistency requires one to relate decisions made at different points in a program execution to each other. Most programmers find this type of global reasoning about programs to be inherently difficult.

Given all this, the problem of *automatic formal verification* of the consistency of a decision-making program seems to be important. Intriguingly, there is very little prior work on this problem. Researchers in application areas like computational geometry have traditionally steered clear of static reasoning about consistency, instead focusing on dynamic approaches [21]. While there is an emerging literature on quantitative reasoning about uncertain programs [4, 5, 19] in the formal methods community, to our knowledge there has been no prior work on verifying that a program makes sequences of discrete decisions consistently. In this paper, we present a sound, automatic program analysis to address this verification problem.

Our analysis applies to infinite-state programs that query a world consisting of an arbitrary number of abstractly defined data objects. A *decision* is the outcome of a query. We model uncertainty using nondeterminism, assuming that each query by the program can return either true or false. To see how a program can be consistent under such a model, consider the code in Fig. 2, which is equivalent to the program in Fig. 1 in the absence of uncertainty (i.e., when $\texttt{Left}(x, y)$ if and only if $\mathit{Left}(x, y)$). However, even if the `Left` queries resolve nondeterministically due to uncertainty, this code is still guaranteed to satisfy the axiom in Eqn. (1). The subtle difference between the two codes is that the one in Fig. 1 does not query a fact when, from the context and the set of axioms, the only non-contradictory answer to the query is known.

To automatically prove the consistency of programs like the one in Fig. 1, we must track the path-sensitive dependencies between different decisions made by the program. This is especially challenging when the program manipulates an unbounded number of objects. For example, suppose that a program decides, by repeatedly querying the world, that

$$\texttt{Left}(p_1, p_2), \texttt{Left}(p_2, p_3), \ldots, \texttt{Left}(p_{n-1}, p_n)$$

for objects $p_1, \ldots, p_n$. If the program now also queries for the relative positions of p_1 and p_n, we may have a consistency violation. This is because we have $\texttt{Left}(p_1, p_n)$ as a logical consequence of the program's decisions by Eqn. 1 while due to uncertainty the result of this query may very well be "not $\texttt{Left}(p_1, p_n)$." To prove consistency, we must be able to reason that this type of violations, namely those that are derived through *unbounded trees* of reasoning, cannot occur.

Our approach to the automation of such proofs is based on an inductive argument. First, for each program location l, we compute an invariant $\mathcal{H}_l$ — called the *history invariant* — that abstracts the set of all decisions made along all possible executions leading to l. Inductively, we assume the decisions in $\mathcal{H}_l$ to be mutually consistent. Next we compute a *current-decision invariant* d_l that abstracts the set of *new* decisions that may be made at l. Consistency verification now amounts to establishing that no decision in the set represented by d_l, taken together with a subset of decisions in $\mathcal{H}_l$, can violate an axiom. We show that this check can be accomplished using a standard fixpoint constraint solver [14]. Decisions taken during the execution of a program, over a collection of input objects, are abstracted by tuples of integers (where integers stand for object identifiers); e.g. $\langle 1, 2\rangle$ may stand for the decision $\texttt{Left}(p_1, p2)$. This then accommodates the abstraction of decision invariants (as defined above) by integer linear arithmetic formulae.

We evaluate our analysis by applying it to a suite of programs from the domain of computational geometry [7, 21], including implementations of standard algorithms for convex hull computations, triangulation, and point location algorithms. While small in size, programs like these are at the heart of geometric libraries used by large real-world systems such as CAD, Geographical Information Systems, Computer Graphics, and various Scientific Computing applications. These examples are also complex: the reasons why they satisfy consistency or fail to do so are often subtle, and small, innocent-looking changes to the code can turn a consistent algorithm into an inconsistent one, or vice versa. However, in almost all of the programs that are actually consistent, our analysis is able to prove consistency automatically and efficiently.

We summarize the contributions of this paper as follows:

- We formulate the problem of verifying the consistency of infinite-state decision-making programs (Sec. 2), and give a sound, automatic program analysis to solve it. (Sec. 4).
- We identify an application domain for our analysis where consistency is a critical correctness property, and where program analysis has not been tried before (Sec. 3).
- We provide a prototype implementation of our system, and evaluate the practical utility of the system using a comprehensive suite of convex hull and triangulation algorithms. (Sec. 5).

2. Formalizing consistency

In this section, we formalize the property of consistency for decision-making programs. We start by defining the "worlds" that are queried by our programs. These queries are answered with uncertainty, and it is this uncertainty that may cause a violation of consistency.

2.1 Modelling the world

A world consists of a set of typed objects together with a set of predicates that are used to query various relationships between these objects. Formally, a *world* is a tuple $\Theta = \langle \mathit{Base}, O, \mathcal{Q}, \mathcal{R}\rangle$,

where $Base$ is a set of base types, O is universe of *objects* with types ranging over $Base$, $\mathcal{Q}$ is a set of *predicate symbols* whose arguments have types in $Base$, and $\mathcal{R}$ is a finite set of *axioms*, where an axiom is a *universally quantified* first-order formula over the predicates in $\mathcal{Q}$.

Intuitively, the predicates in $\mathcal{Q}$ form the "interface" between programs and the world Θ. We assume for easier exposition that each $A \in \mathcal{Q}$ has the same arity m. The axioms $\mathcal{R}$ codify the laws of the world — e.g., the laws of geometry. Since the understanding of nontrivial worlds is often partial, we do not expect our axioms to be complete in a mathematical sense.

Example 1. Let $\Theta_{tc} = \langle\{\texttt{1DPoint}\}, O, \mathcal{Q}_{tc}, \mathcal{R}_{tc}\rangle$ be a world in which `1DPoint` is the type of points arranged on a line, O is a set of such points, and $\mathcal{Q}_{tc}$ consists of a single predicate $Left$. The predicate $Left(u, v)$ holds for `1DPoints` u and v iff u lies to the left of v—i.e., if $u.x < v.x$, where $u.x$ and $v.x$ are respectively the coordinates of u and v with respect to a fixed origin. A sensible set of axioms $\mathcal{R}_{tc}$ for Θ_{tc} is as follows:

1. The transitivity axiom used in the introduction:
$$\forall y_1, y_2, y_3 : \\ y_1 \neq y_2 \wedge y_2 \neq y_3 \wedge y_3 \neq y_1 \implies \\ (Left(y_1, y_2) \wedge Left(y_2, y_3) \implies Left(y_1, y_3)).$$
2. An axiom that asserts that $Left$ is "complete" — i.e., any two distinct points on the line are related by $Left$:
$$\forall y_1, y_2 : y_1 \neq y_2 \implies (Left(y_1, y_2) \vee Left(y_2, y_1)). \quad \square$$

To reduce notation, we adopt two simple syntactic conventions while writing down axioms. First, we drop the explicit universal quantifiers over the axiom variables. Second, without loss of generality, we use distinct variable names in the axioms to represent distinct objects in models of the theory. For example, the first axiom in the above example is written simply as

$$Left(y_1, y_2) \wedge Left(y_2, y_3) \implies Left(y_1, y_3),$$

the assumptions $(y_1 \neq y_2)$, $(y_2 \neq y_3)$, and $(y_1 \neq y_3)$ following from the fact that y_1 and y_2 are different symbols. It is acceptable, however, to write the axiom $R(x, x)$ to state that a predicate R is reflexive.

2.2 Programs

Next we define a core language DMP of decision-making programs. Programs in this language are a symbolic transition systems that can manipulate both individual objects and *lists* of objects.

There is a key modelling question in the definition of DMP: how do we model the uncertainty in a program's queries about the world? Our definition does this via nondeterminism: we assume that *every* call of a program to a world predicate may return an incorrect answer.

This model may appear too conservative at first sight, given that we do not model uncertainty quantitatively. However, it has several advantages. The first and foremost is its *generality*: the model is equally applicable to settings where uncertainty is the result of sensing errors as it is to the settings where floating-point error is the source of uncertainty. Any reasonable quantitative model of uncertainty, on the other hand, would have to be domain-specific, and perhaps consider low-level details of the hardware used for sensing or computation.

Second, our model is simpler and therefore more amenable to automated reasoning than quantitative models. Indeed, given the highly challenging nature of the consistency analysis problem, this model offers a "sweet spot" for static analysis.

Finally, our empirical experience with the model is satisfactory. We ran some experiments with our benchmark examples from Sec. 5 for the case when uncertainty is only due to numerical error. For the examples that are inconsistent under our model, it was always possible to produce a concrete input that forces the program to violate basic invariants under floating-point semantics. In other words, these examples would be inconsistent under any other reasonable definition of consistency.

Program syntax. Let us fix a world $\Theta = \langle Base, O, \mathcal{Q}, \mathcal{R}\rangle$ and a universe of *variables*. Each variable represents either an object in O or a list of such objects. Variables of base types are denoted by $x, x_1, x_2, \ldots$, and variables of list types (representing collections of objects) are denoted by $X, X_1, X_2, \ldots$.

We assume an alphabet[2] $\texttt{Q}$ of symbols whose members have a one-to-one correspondence with the predicates in $\mathcal{Q}$. For $A \in \mathcal{Q}$, we denote the corresponding symbol in $\texttt{Q}$ by $\texttt{A}$. Also, we define a symbol $\overline{\texttt{A}}$ for each $\texttt{A} \in \texttt{Q}$ and let $\overline{\texttt{Q}}$ be the set of such symbols. It is assumed that $\texttt{Q}$ and $\overline{\texttt{Q}}$ do not have any overlap with each other or with $\mathcal{Q}$. Intuitively, for objects $p_1, \ldots, p_m$, the notation $\texttt{A}(p_1, \ldots, p_m)$ represents the decision made by the program that the fact $A(p_1, \ldots, p_m)$ holds, and $\overline{\texttt{A}}(p_1, \ldots, p_m)$ represents the decision that the fact $\neg A(p_1, \ldots, p_m)$ holds.

We allow a program to test and manipulate its local variables using the following primitives:

- We allow boolean expressions that test whether a list is empty: $X = \emptyset$ and $X \neq \emptyset$.
- We allow the following kinds of assignments to variables:
  1. $x_1 := x_2$. An assignment where an object-valued variable x_1 is set to the value of another object-valued variable x_2.
  2. $x := \mathit{first}(X)$ and $x := \mathit{last}(X)$. Respectively set x to the value of the first and last element of the list X. The list X is not updated.
  3. $x := \mathit{next}(X)$ and $x := \mathit{pop}(X)$. Respectively remove the first and last elements of the list X, and set x to the value of this element.
  4. $X := \emptyset$. Sets X to the empty list.
  5. $X_1 := X_2$. Sets the list X_1 to the value of the list X_2.
  6. $\mathit{prepend}(x, X)$. Adds the object x to the front of the list X.
  7. $\mathit{append}(x, X)$. Adds the object x to the end of the list X.

The syntax of DMP programs is now given as follows:

Definition 1 (Programs)**.** *A program over a world Θ is a tuple $P = \langle Loc, l_0, \mathcal{T}\rangle$, where Loc is a finite set of* locations, *$l_0 \in Loc$ is the* initial location, *and $\mathcal{T}$ is a set of* transitions. *A transition $\langle l \,|\, T \,|\, l'\rangle$ from a location l to a location l' is of one of the following forms:*

1. $\langle l \,|\, \texttt{A}(x_1, \ldots, x_m) \,|\, l'\rangle$
2. $\langle l \,|\, \overline{\texttt{A}}(x_1, \ldots, x_m) \,|\, l'\rangle$
3. $\langle l \,|\, assume(b) \,|\, l'\rangle$
4. $\langle l \,|\, U \,|\, l'\rangle$

where $\texttt{A} \in \texttt{Q}$; $\overline{\texttt{A}} \in \overline{\texttt{Q}}$; $x_i \in Var$; b is a boolean expression; and each U is an assignment of one the forms above.

Each transition $\langle l \mid T \mid l'\rangle$ takes the program from location l to location l'. Transitions of forms (1) and (2) represent queries to the world and decisions made on basis of those queries. A transition of form (1) is taken when a call to $\texttt{A} \in \texttt{Q}$ returns $true$, and one of form (2) is taken when it returns $false$. Transitions of form (3) are local tests on program variables while those of form (4) are updates to variables. For instance, the transition $\langle l \mid x := \mathit{first}(X) \mid l'\rangle$ is executed when control is at l, sets x to the first element of X (without updating X), and moves the control to l'.

[2] Note the typewriter font used for $\texttt{Q}$, which distinguishes this set from $\mathcal{Q}$.

The sets of locations and variables of P are respectively denoted by $Loc(P)$ and $Var(P)$. Note that a location may have multiple outgoing or incoming transitions. However, we assume, without loss of generality, that whenever P has a query transition $\langle l \mid \alpha(\dots) \mid l' \rangle$, then it has no other outgoing transition at l. We denote by Loc_P^α the set of locations in P that have an outgoing transition of the above form. Collectively, locations like this are known as *query locations*.

We assume all predicates in P to be well-typed: if a transition is guarded by $\mathtt{A}(x_1, \dots, x_m)$ and x_i is of type τ_i, then the type of $\mathtt{A}$ is $(\tau_1 \times \dots \times \tau_m)$.

Uncertain Semantics. Now we sketch the semantics of programs under uncertainty. Let a *state* of P be a pair $s = (l, \sigma)$, where l is a location and σ is a function that maps variables x_i and X_i to appropriately typed values. The set of states of P is denoted by $\Sigma(P)$.

The semantics of P is defined using a labelled transition system $\langle \Sigma, \rightharpoonup, \Sigma_{in} \rangle$, where $\rightharpoonup$ is a set of *semantic transitions* and Σ_{in} is a set of *initial states*. An initial state of P is a state of the form (l_0, σ) for some σ, and where each object o appears at most once in at most one list in the domain of σ (i.e., initially the contents of the lists are repetition-free and pairwise disjoint).

The set $\rightharpoonup$ is the least relation such that for all transitions $t = \langle l \,|\, T \mid l' \rangle$ in P,

- If T is a query $\alpha(x_1, \dots, x_m)$ (for $\alpha \in \mathtt{Q} \cup \overline{\mathtt{Q}}$), then we have $(l, \sigma) \xrightharpoonup{t} (l', \sigma)$ for all σ.
- If T is an assignment $x := e$ or $X := E$, then for each state (l, σ), we have $(l, \sigma) \xrightharpoonup{t} (l', \sigma')$, where σ' is obtained by updating the variables of P according to the semantics of assignments sketched earlier.
- If T is a test $assume(b)$ and (l, σ) is a state that satisfies b, then we have $(l, \sigma) \xrightharpoonup{t} (l', \sigma)$.

We define an *(uncertain) execution* of P as a finite sequence $\rho = (l_0, \sigma_0) \xrightharpoonup{t_0} (l_1, \sigma_1) \cdots (l_n, \sigma_n)$ such that $(l_0, \sigma_0) \in \Sigma_{in}$ and for all i, $(l_i, \sigma_i) \xrightharpoonup{t_i} (l_{i+1}, \sigma_{i+1})$. For each location l, the set of executions ending at a state (l, σ) (for some σ) is denoted by $Exec(l)$. The set of all executions of P is denoted by $Exec(P)$.

It seems intuitive to define a semantics of programs in *absence of uncertainty* where queries obtain true information about the world — e.g., a query transition $\langle l \mid \mathtt{A}(x_1, \dots, x_m) \mid l' \rangle$ is enabled at a state where x_i has value p_i if and only if the relation $A(p_1, \dots, p_m)$ holds in the world. However, such an ideal semantics is of no use to our analysis where the core concern is the effect of uncertainty, and therefore, we do not develop this semantics.

2.3 Consistency

An uncertain execution of a DMP program P, may issue a sequence of queries whose answers, taken together, violate $\mathcal{R}$. We say that P is consistent if this cannot happen in any uncertain execution.

To formalize this notion of consistency we need a formal definition for decisions: a *decision* in the world Θ is a literal $\delta = \alpha(p_1, \dots, p_m)$, where $\alpha \in (\mathtt{Q} \cup \overline{\mathtt{Q}})$, $p_1, \dots, p_m \in O$, and the type of p_i matches the expected type of the i-th input to $\mathcal{Q}$. We denote the set of decisions over Θ by $Dec(\Theta)$ or simply Dec. For a decision δ, $[\![\delta]\!]$ denotes the literal $A(p_1, \dots, p_m)$ when δ equals $\mathtt{A}(p_1, \dots, p_m)$, and $\neg A(p_1, \dots, p_m)$ when δ equals $\overline{\mathtt{A}}(p_1, \dots, p_m)$.

Now let $\rho = (l_0, \sigma_0) \xrightharpoonup{t_0} (l_1, \sigma_1) \cdots \xrightharpoonup{t_n} (l_{n+1}, \sigma_{n+1})$ be an execution of P. We define a *decision of* ρ to be any decision $\delta = \alpha(p_1, \dots, p_m) \in Dec$ such that for some i, we have:

1. $t_i = \langle l_i \,|\, \alpha(x_1, \dots, x_m) \mid l_{i+1} \rangle$
2. for all $j \in \{1, ..., m\}$, $p_j = \sigma_i(x_j)$.

The set of decisions of ρ is denoted by $Dec(\rho)$.

One can collectively represent the decisions made during an execution ρ of P using a logical formula, which we call the *decision formula* of that execution, which is defined as:

$$\Psi(\rho) = \bigwedge_{\delta \in Dec(\rho)} [\![\delta]\!].$$

Consistency of P is now defined as follows:

Definition 2 (Consistency)**.** *The program P is* consistent *(with respect to the world Θ) if for all executions $\rho \in Exec(P)$, the formula $\Psi(\rho) \wedge \left(\bigwedge_{R \in \mathcal{R}} R\right)$ is satisfiable.*

Thus our goal is to verify that a given program P is consistent under the above definition.

Example 2. Let us consider the code in Fig. 2 once again. It is easy to translate this code to a DMP program over the world Θ_{tc} from Example 1 (we assume L_1-L_6 are simple assignments). This program manipulates a bounded number of objects and has a bounded number of executions; by enumerating these executions, we find that it is consistent.

Now let us consider a more involved program, also over Θ_{tc}, that issues queries over an unbounded number of objects. As with the previous example, we write the program in a more readable structured syntax which can be easily translated into DMP:

```
1  x1 := next(X);  x2 := next(X)
2  while X ≠ ∅
3      do if Left(x1, x2) then skip else skip
4          x1 := x2;  x2 := next(X)
```

In our model, the test in Line 3 can always evaluate incorrectly. However, we can argue that this program is consistent nonetheless.

Let the objects in the input list X have identifiers 1 to n. Consider the execution ρ that takes the $true$-branch of the code in all iterations. The decision formula for ρ is

$$\Psi(\rho) = Left(1, 2) \wedge Left(2, 3) \wedge Left(3, 4) \dots Left(n-1, n).$$

Now note that $(\Psi(\rho) \wedge R)$, where R is the conjunction of the axioms in Example 1, is satisfiable. It is not hard to see that for *every* execution ρ' of P_{tc}, $(\Psi(\rho') \wedge R)$ is satisfiable. Therefore, the program is consistent. On the other hand, if the program made an extra decision $\mathtt{Left}(n, 1)$ somewhere in the execution, then the conjunct $Left(n, 1)$ would be added to $\Psi(\rho)$, and the above property would no longer hold.

3. Consistency in geometric programs

In this section, we show how the notion of consistency plays out in the context of computational geometry. We do so using two examples. Both are algorithms for computing a convex hull for a set of points in two dimensions (2D); one of the algorithms is not consistent, while the other is. As before, we describe the algorithms using a more readable, structured syntax rather than that of DMP.

3.1 The world

Any convex hull algorithm queries a geometric space about positions of points in it — this space Θ_{ch} is our world. Formally, we let $\Theta_{ch} = \langle \{\mathtt{2DPoint}\}, O, \mathcal{Q}_{ch}, \mathcal{R}_{ch} \rangle$, where $\mathtt{2DPoint}$ is the type of points on the 2D plane, O is the set of all 2D points, and $\mathcal{Q}_{ch}$ consists of a single *orientation predicate* $Lturn(u, v, w)$. The semantics of this predicate is that $Lturn(u, v, w)$ is true when w is to the left of the infinite directed line $\overrightarrow{uv}$. For example, in Fig. 5(a), we have $Lturn(p_1, p_2, p_3)$. Under the ideal semantics of reals, this

1. **Cyclic symmetry:** $Lturn(x, y, z) \implies Lturn(y, z, x)$.
2. **Antisymmetry:** $Lturn(x, y, z) \implies \neg Lturn(x, z, y)$.
3. **Nondegeneracy:** $Lturn(x, y, z) \vee Lturn(x, z, y)$.
4. **Interiority:** $Lturn(x, y, t) \wedge Lturn(y, z, t) \wedge Lturn(z, x, t) \implies Lturn(x, y, z)$.
5. **Transitivity:** $Lturn(x, y, z) \wedge Lturn(x, y, t) \wedge Lturn(y, z, t) \wedge Lturn(y, z, w) \wedge Lturn(t, y, w) \implies Lturn(x, y, w)$.

Figure 3. Knuth's axioms for convex hull algorithms

predicate can be defined as

$$(v.x - u.x)(w.y - u.y) - (v.y - u.y)(w.x - u.x) > 0$$

where $u.x$ and $u.y$ are respectively the x- and y-coordinates of u.

We define $\mathcal{R}_{ch}$ using an axiomatization of convex hull predicates (Fig. 4) by Knuth [17]. Consider Fig. 5(a) again, and note that $Lturn(p_1, p_2, p_3)$. Axiom 1 says that this implies the facts $Lturn(p_3, p_1, p_2)$ and $Lturn(p_2, p_3, p_1)$; Axiom 2 says that we cannot have $Lturn(p_1, p_3, p_2)$ in this case. Axiom 3 is an *assumption* that no three input points are collinear (collinear points form a degenerate case for convex hull algorithms).

Axiom 4 says that (Fig. **??** (a)) if p_2 is to the left of $\overrightarrow{p_1p_2}$, $\overrightarrow{p_2p_3}$ and $\overrightarrow{p_3p_1}$, then p_4 is *inside* the triangle $\triangle p_1p_2p_3$, and more over the triangle is oriented counter clockwise, or p_3 is to the left of $\overrightarrow{p_1p_2}$. Axiom 5 defines a transitivity property of $Lturn$: to understand its geometric intuition, consider Fig. **??**(b) with $x = p_1$, $y = p_2$, $z = p_3$, $t = p_4$, and $w = p_5$. In this case, we have $Lturn(p_1, p_2, p_3)$, $Lturn(p_1, p_2, p_4)$, $Lturn(p_2, p_3, p_4)$, $Lturn(p_2, p_3, p_5)$, $Lturn(p_4, p_2, p_5)$. The axiom demands that we also have $Lturn(p_1, p_2, p_5)$, which is indeed true here.

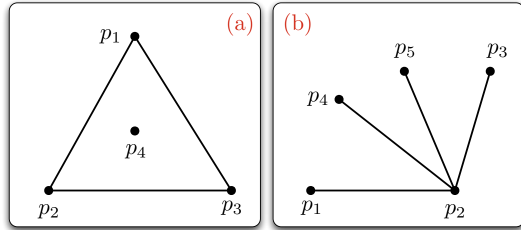

Figure 4. Interiority and transitivity axioms

Axioms 1–5 are easily seen to be consistent with Euclidean geometry. In [17], Knuth argues that if the results of all $Lturn$ predicate checks for a set of points satisfy these axioms, then a convex hull always exists. While the above axioms are specific to convex hulls, similar axiomatizations are known for other classes of geometric computations.

3.2 Inconsistent convex hull

Consider the algorithm SIMPLECONVEXHULL [7] in Fig. 3, which is a naive (and slow) algorithm for computing the convex hull of a set of points S. Note that it is standard for convex hulls to have a counterclockwise orientation. Following the convention developed earlier, we use $\overline{\texttt{Lturn}}(u, v, w)$ to denote the *decision* made by the algorithm that $\neg Lturn(u, v, w)$. The algorithm iterates over all pairs of points (u, v) (as potential convex hull edges); if there exists a point w such that the decision $\overline{\texttt{Lturn}}(u, v, w)$ is made, then (u, v) is removed from the set of edges forming the convex hull (since if (u, v) is an edge of the convex hull (in the counterclockwise direction), then every other point must lie to the left of (u, v)). It is easy to see that if the algorithm is implemented using

Input: A set S of points in the 2D plane.
Output: A list E containing the edges of the convex hull

SIMPLECONVEXHULL(S)

1 $E := \emptyset$
2 **for** all ordered pairs $(u, v) \in S \times S$ with $u \neq v$
3   **do** $valid := true$
4   **for** all $w \in S$ with $u \neq w$ and $v \neq w$
5   **do if** $\neg\texttt{Lturn}(u, v, w)$
6   **then** $valid := false$
7   **if** $valid$
8   **then** add the directed edge (u, v) to E

Figure 5. Inconsistent convex hull

ideal reals, without any uncertainty, then SIMPLECONVEXHULL in fact computes a convex hull.

On the other hand, under uncertainty (for example, due to floating-point error), the decision $\overline{\texttt{Lturn}}(u, v, w)$ can evaluate nondeterministically. Consider the input set in Fig. 5 where the correct hull is illustrated in (a). $Lturn(p_1, p_2, p_3)$ is true, but the points p_1, p_2, p_3 are *nearly* collinear. First, note that the algorithm may evaluate $\texttt{Lturn}$ for all permutations of the three points p_1, p_2, and p_3. If, *due to numerical uncertainty*, the program decides that $\overline{\texttt{Lturn}}(p_1, p_2, p_3)$, $\overline{\texttt{Lturn}}(p_2, p_3, p_1)$, and $\overline{\texttt{Lturn}}(p_1, p_3, p_2)$, then none of the three edges (p_1, p_2), (p_2, p_3), and (p_1, p_3) will belong to the convex hull, and the resulting hull will be as illustrated in (b), which is not even a closed curve. Similarly, we could have a scenario where all three edges end up in the convex hull, where all the three queries return true, and the result will be as in (c); again, completely wrong, since it has two cycles.

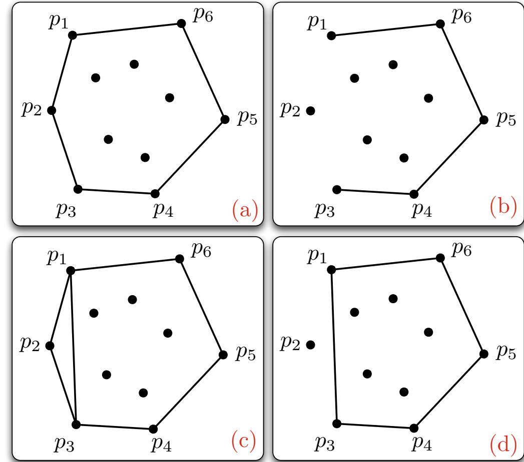

Figure 6. Inconsistency in convex hull

The main problem is that, for both cases (b) and (c), where decisions $\overline{\texttt{Lturn}}(p_1, p_2, p_3)$ and $\overline{\texttt{Lturn}}(p_1, p_3, p_2)$ are both taken, there is an inconsistency with the antisymmetry axiom of Fig. 4. The *consistent* executions of this program result in either (a) or (d). The convex hull in (d) is obtained from deciding that $\overline{\texttt{Lturn}}(p_1, p_2, p_3)$, $\overline{\texttt{Lturn}}(p_2, p_3, p_1)$, and $\texttt{Lturn}(p_1, p_3, p_2)$; these decisions are not all correct, but they are consistent. Consequently, while Figure (d) is slightly different from the ideal answer, it is structurally a convex hull: an output that would be produced by the ideal real-number algorithm on *some* input (this is an input in which p_2 is perturbed to move to the left of line $\overrightarrow{p_1p_3}$). In contrast, (b) and (c) are outputs that the ideal algorithm could never produce.

To see why this difference is substantial, consider a procedure that iterates over the "hull" computed by the convex hull routine, starting with p_2 and terminating when it comes back to p_2 again. In case (d), it will terminate as usual. In case (b), it will not be able to proceed past p_2. In (c), it could get stuck in an infinite loop.

The inconsistency of the above algorithm is by no means a rare anomaly. Many everyday implementations of convex hulls (and other geometric computations) are inconsistent. The crashes, infinite loops, and other errors that inconsistency can lead to are well-documented. We refer the reader to [21] for an in depth discussion of the consequences of inconsistent geometric computations.

3.3 Consistent convex hull: Graham Scan [12]

Graham scan [12] (Fig. 7) is an example of a consistent convex hull algorithm.

The point p_0, which we know belongs to H, is our "sentinel". In the main loop, we check if the last two points u and v of H together with a new point w from S make a left turn (this test is encoded using the `Lturn` predicate). If so, then we *speculatively* assume that these points belong to the hull and add w to H. If not, then v cannot belong to the convex hull of S (since both edges adjacent to v have a point to the right of them, and therefore, they cannot be convex hull edges), and we *delete* it from H. For example, upon considering $\langle p_1, p_2, p_3\rangle$ in the figure, p_2 is deleted from H. In general, we keep deleting until the last two points in H together with w make a left turn. Note that there is no bound on the number of points that may be deleted during this step.

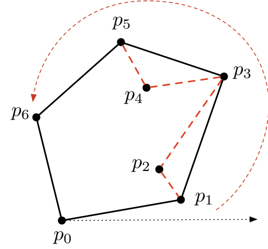

Figure 7. Graham Scan

The Graham Scan algorithm, unlike the naive algorithm, is consistent (we discuss why in Sec. 5). It is very difficult to reason about consistency of this algorithm manually. One needs to reason about all program paths containing unboundedly many facts (dependent on the size of input) and argue that none contains an inconsistency with respect to the axioms. An expert user may be able to argue why none of the first 3 axioms (in Fig. 4) can ever be falsified by this algorithm. The algorithm progresses in a way that, for every three points p, q, and r, it evaluates the $Lturn(p, q, r)$ predicate exactly once (the last point r is always a fresh new point). Therefore, it is not possible to find two predicate evaluations that are not consistent with any of the first three axioms (which all need two different evaluations of the predicate on two different permutations of the same 3 points). But, extending this reasoning to axioms 4 and 5 manually is difficult. In the next section, we present an analysis that is able to easily verify the consistency of the above algorithm.

We note that the assumption that the input points are sorted on basis of their polar angle from p_0 is needed for the algorithm's functional correctness. However, it is irrelevant to the program's consistency: even if the input points were ordered arbitrarily, the decisions made in an execution of the algorithm would not violate Knuth's axioms, and structurally, the output would still be a simple closed cycle. Our analysis chooses to ignore the assumption that the input list is sorted in this way (this information is lost in abstraction), and still discovers an automated proof that the algorithm is consistent.

4. Verifying consistency

In this section, we describe our method for algorithmic verification of consistency. Let us fix a world $\Theta = (Base, O, \mathcal{Q}, \mathcal{R})$ and a program P. The broad idea of our algorithm is to prove P consistent by an inductive argument: each time a new decision is made, we assume that the set of decisions that have already been made are consistent and prove that the new decision does not violate consistency (in combination with past decisions). We automate this reasoning in two phases as follows.

Input: A point p_0 (right-most bottom-most point in the set) and (non-empty) list of S of points. The points in S are assumed to be sorted on basis of increasing polar angle from p_0.

Output: List H containing the points forming the convex hull, also sorted on basis of their polar angle from p_0

```
GRAHAMSCAN(p0, S)
 1  if |S| < 3 then return [p0; S]
 2  H := [next(S); next(S)]
 3  for w := next(S)
 4      do
 5          v := last(H); u := secondLast(H)
 6          while ¬ Lturn(u, v, w)
 7              do pop(H)
 8                  if |H| ≥ 2
 9                      then v := last(H) ; u := secondLast(H)
10                      else break
11          Append w to H
12  w := p0; v := last(H); u := secondLast(H)
13  while ¬ Lturn(u, v, w)
14      do pop(H)
15          if |H| ≥ 2
16              then v := last(H) ; u := secondLast(H)
17              else break
18  Append p0 to H
19  return H
```

Figure 8. Consistent Convex Hull Algorithm: Graham Scan. The operation *secondLast* returns the second last element of the list, without modifying the list.

In the first phase, we approximate the sets of decisions that can be made by executions of P by constructing an integer abstraction Π of P and then computing numerical invariants for Π. Example 2 illustrates the idea behind the integer abstraction, which models objects as integer identifiers, and sets of decisions with sets of integer tuples. We analyze Π with an abstract interpreter to compute for each query location l of P:

1. A *current-decision invariant* d_l that symbolically represents the set of decisions that can be made at l.
2. A *history invariant* $\mathcal{H}_l$ that symbolically represents the set of all decisions made along executions that end at location l.

Our representation of a set of decisions by a numerical invariant makes use of a set of integer auxiliary variables which represent the arguments of queries (recalling that our encoding represents objects as integer identifiers). A set of decisions (i.e., a set of tuples of objects) is represented by an arithmetic formula over the program variables as well as the auxiliary variables. The set of models of such a formula can be interpreted as a set of decisions (or more accurately, as a function mapping program states to a set of decisions).

For example, let us revisit the program of Example 2 (reproduced here for convenience).

```
1  x1 := next(X);  x2 := next(X)
2  while X ≠ ∅
3      do if Left(x1, x2) then skip else skip
4          x1 := x2;  x2 := next(X)
```

Let us assume that the objects in the list have strictly increasing IDs (e.g. the first point on the list has ID 1, the second ID 2, and so on),

and let us use auxiliary variables $\#^1_{\texttt{Left}}$ and $\#^2_{\texttt{Left}}$ to respectively represent the IDs of the first and second arguments of the call to `Left`. At line 3, we have the invariant $x_1 < x_2$, and we have outgoing query transitions $\texttt{A}(x_1, x_2)$ and $\overline{\texttt{A}}(x_1, x_2)$. The current-decision invariant at this location is:

$$d_l : \#^1_{\texttt{Left}} = x_1 < x_2 = \#^2_{\texttt{Left}}.$$

On the other hand, in a history invariant, $\#^1_{\texttt{Left}}$ and $\#^2_{\texttt{Left}}$ refer to the IDs of the arguments of an arbitrarily selected *past call* to `Left`, while x_1 and x_2 refer to the *current* program variables. Since the program processes list elements in order of increasing IDs, the following history invariant holds at Line 3:

$$\mathcal{H}_l : \#^1_{\texttt{Left}} < \#^2_{\texttt{Left}} \leq x_1 < x_2$$

In the second phase of the algorithm, we verify that for each query location l in the program P, there is a model of the world that is consistent with the decisions d_l and $\mathcal{H}_l$ (i.e., d_l, $\mathcal{H}_l$, and the world axioms $\mathcal{R}$ together cannot derive *false*). The challenge here is that there may be no obvious inconsistency between d_l and $\mathcal{H}_l$, but *false* may be derived using the axioms in $\mathcal{R}$ (and there is no *a priori* bound on the size of such a proof).

To overcome this challenge, we generate a logic program from the world axioms $\mathcal{R}$ that, given a current-decision invariant d_l and a history invariant $\mathcal{H}_l$, generates every decision that can follow from the decisions in d_l and $\mathcal{H}_l$. The problem now reduces to verifying that this system cannot possibly derive a contradiction. This task can be solved automatically using existing fixpoint constraint solvers, such as μZ [14]. Next, we describe the two phases of the algorithm in detail.

4.1 Generating history and current-decision invariants

To generate the history and current-decision invariants used by our consistency verification procedure, we first abstract P by a program Π over the integers. In Π, each object accessed by P is modelled as a integer ID.[3] In addition to simulating the execution of P, Π records the decisions it makes using a set of auxiliary variables (denoted by $\#$).

Constructing Π

We abstract lists λ by pairs consisting of the IDs of the first and last element of λ. The challenge in this list abstraction is to maintain information about a list λ as objects are added to it or removed from it. Our approach to doing so is to maintain the invariant that the objects in λ appear in an order *sorted* by their IDs (note that there is no relation between these IDs and any values associated with the objects themselves; we just assign the IDs to the elements of the list in order in which they appear on the list). For instance, let X^0 and X^1 respectively denote the IDs of the first and last items in a list X. Assuming that our sortedness invariant holds, the integer abstraction of $next(X)$ returns the value of X^0 and sets X^0 to a nondeterministically chosen value between the current values of X^0 and X^1. Note that since the abstraction assigns the integer IDs (they are not part of original object information), all the input lists to the program are initially assumed to satisfy this invariant (based on the precondition about the disjointness of the lists and the elements on each list that we already mentioned in Section **??**).

Of course, the sortedness invariant can be violated when the program tries to insert an item x into a list X. We detect such violations by comparing the ID of x with the value X^1; if sortedness is violated, we conservatively flag the program as inconsistent.

[3] Our analysis can be generalized to one where object IDs are not integers but tuples of integers, or even more generally, elements of a partially ordered set. For simplicity, we stick to integer IDs in this paper.

We now present the construction of Π formally. To avoid introducing more notation, we describe the program Π in a DMP-like syntax. An integer program is a tuple $\langle Loc, Var, l_0, \mathcal{T}\rangle$, where Loc is a set of locations, Var is a set of variables ranging over the integers and booleans, $l_0 \in Loc$ is an initial location, and $\mathcal{T}$ is a set of transitions. Transitions between locations l and l' have the following forms:

- Test transitions $\langle l \mid assume(b) \mid l'\rangle$, where b is a linear arithmetic formula,
- $\langle l \mid havoc(x) \mid l'\rangle$, which changes the value of the variable, x to an arbitrary integer value,
- $\langle l \mid x := e \mid l'\rangle$, an assignment to variable x, and
- $\langle l \mid assert(b) \mid l'\rangle$, which asserts that a property b (expressed as a linear arithmetic formula) holds.

We do not give a detailed semantics for these transitions as they are standard. For simplicity, we sometimes write sequential compositions of transitions as a single transition—e.g., the syntax $\langle l \mid x := e; havoc(x) \mid l'\rangle$ is abbreviation for a pair of transitions $\langle l \mid x := e \mid l''\rangle$ and $\langle l'' \mid havoc(x) \mid l'\rangle$, where l'' is a location that is not used anywhere else in the program.

Now, let $P = \langle Loc_P, Var_P, l_{0,P}, \mathcal{T}_P\rangle$. We construct the program $\Pi = \langle Loc_\Pi, Var_\Pi, l_{0,\Pi}, \mathcal{T}_\Pi\rangle$ as follows:

- The initial location $l_{0,\Pi}$ of Π is a fresh location, not in Loc_P. Loc_Π contains Loc_P and $l_{0,\Pi}$, and also a number of auxiliary locations introduced in the translation of transitions (Fig. 9). Π inherits a notion of query locations from P: for all $\alpha \in \texttt{Q} \cup \overline{\texttt{Q}}$, we define $Loc^\alpha_\Pi = Loc^\alpha_P$.
- Var_Π is the least set such that:
  1. For each object-valued variable x in P, Var_Π contains an integer-valued variable that tracks the ID of x. Abusing notation, we call this variable x as well.
  2. For every list-valued variable X in P, Var_Π contains two integer-valued variables X^0 and X^1 that track the IDs of the first and last elements of X.
  3. For each $\texttt{A} \in \texttt{Q}$ and each $1 \leq i \leq m$, Var_Π has an integer-valued variable $\#^i_{\texttt{A}}$ ranging over the integers. These variables are used to record A-decisions.
  4. Var_Π contains a special boolean-valued variable $flag_{\texttt{A}}$ for each $\texttt{A} \in \texttt{Q}$. This variable is set to $true$ when the $\#^i_{\texttt{A}}$ variables are initialized — i.e., the arguments of some query is assigned to them. Initially, each $flag_{\texttt{A}}$ variable is $false$.
- The set of transitions $\mathcal{T}_\Pi$ of Π is defined by the rules in Fig. 9. Here, the rule INIT captures the fact that the $\#^i_{\texttt{A}}$s are uninitialized at the beginning of program executions. The two rules QUERY-1 and QUERY-2 abstract query transitions. One of them sets the $\#_{\texttt{A}}$ variables while the other does nothing — together, they model a nondeterministic choice to save the arguments of a decision in the $\#_{\texttt{A}}$ variables. The remaining rules are integer abstractions of updates to and tests of object and list-valued variables.

Current-decision and History Invariants

Having constructed an integer program abstraction Π of P, we may use standard techniques to generate for each location $l \in Loc_\Pi$ a (numerical) *history invariant* $\mathcal{H}_l$. Our implementation employs an abstract interpreter over a domain of partitioned octagons, which we describe further in Sec. 5.1. Each $\mathcal{H}_l$ is a linear arithmetic formula over the variables Var_Π, which includes the variables which correspond to object-valued variables of P as well as the auxiliary # and *flag*variables.

$$\text{(Init)}\frac{\mathtt{Q}=\{\mathtt{A}_1,...,\mathtt{A}_{|\mathtt{Q}|}\}}{\langle l_{0,\Pi}\,|\,\mathit{flag}_{\mathtt{A}_1}:=\mathit{false};\ldots;\mathit{flag}_{\mathtt{A}_{|\mathtt{Q}|}}:=\mathit{false}\,|\,l_0\rangle\in\mathcal{T}_\Pi}\qquad\text{(Asgn)}\frac{\langle l\mid x_1:=x_2\mid l'\rangle\in\mathcal{T}_P}{\langle l\mid x_1:=x_2\mid l'\rangle\in\mathcal{T}_\Pi}$$

$$\text{(Test-empty)}\frac{\langle l\mid \mathit{assume}(X=\emptyset)\mid l'\rangle\in\mathcal{T}_P}{\langle l\mid \mathit{assume}(X^0>X^1)\mid l'\rangle\in\mathcal{T}_\Pi}\qquad\text{(Test-not-empty)}\frac{\langle l\mid \mathit{assume}(X\neq\emptyset)\mid l'\rangle\in\mathcal{T}_P}{\langle l\mid \mathit{assume}(X^0\leq X^1)\mid l'\rangle\in\mathcal{T}_\Pi}$$

$$\text{(Query-1)}\frac{\langle l\,|\,\alpha(x_1,\ldots,x_m)\,|\,l'\rangle\in\mathcal{T}_P\qquad \alpha\in\{\mathtt{A}\}\cup\{\overline{\mathtt{A}}\}}{\langle l\mid \#^1_{\mathtt{A}}:=x_1;\ldots;\#^m_{\mathtt{A}}:=x_m;\mathit{flag}_{\mathtt{A}}:=\mathit{true}\mid l'\rangle\in\mathcal{T}_\Pi}\qquad\text{(Query-2)}\frac{\langle l\,|\,\alpha(x_1,\ldots,x_m)\,|\,l'\rangle\in\mathcal{T}_P\qquad \alpha\in\{\mathtt{A}\}\cup\{\overline{\mathtt{A}}\}}{\langle l\mid \mathit{assume}(\mathit{true})\mid l'\rangle\in\mathcal{T}_\Pi}$$

$$\text{(First)}\frac{\langle l\mid x:=\mathit{first}(X)\mid l'\rangle\in\mathcal{T}_P}{\langle l\mid \mathit{assert}(X^0\leq X^1);x:=X^0\mid l'\rangle\in\mathcal{T}_\Pi}\qquad\text{(Next)}\frac{\langle l\mid x:=\mathit{next}(X)\mid l'\rangle\in\mathcal{T}_P}{\langle l\,|\,x:=X^0;\mathit{havoc}(X^0);\mathit{assume}(x<X^0\leq \mathit{max}(X^1,X^0+1)\,|\,l'\rangle\in\mathcal{T}_\Pi}$$

$$\text{(Last)}\frac{\langle l\mid x:=\mathit{last}(X)\mid l'\rangle\in\mathcal{T}_P}{\langle l\mid \mathit{assert}(X^0\leq X^1);x:=X^1\mid l'\rangle\in\mathcal{T}_\Pi}\qquad\text{(TL)}\frac{\langle l\mid x:=\mathit{pop}(X)\mid l'\rangle\in\mathcal{T}_P}{\langle l\,|\,x:=X^1;\mathit{havoc}(X^1);\mathit{assume}(\mathit{min}(X^0,X^1-1)\leq X^1<x)\,|\,l'\rangle\in\mathcal{T}_\Pi}$$

$$\text{(LA)}\frac{\langle l\mid X_1:=X_2\mid l'\rangle\in\mathcal{T}_P}{\langle l\mid X_1^0:=X_2^0;X_1^1:=X_2^1\mid l'\rangle\in\mathcal{T}_\Pi}\qquad\text{(Empty)}\frac{\langle l\,|\,X:=\emptyset\,|\,l'\rangle\in\mathcal{T}_P}{\langle l\,|\,\mathit{havoc}(X^0);\mathit{havoc}(X^1);\mathit{assume}(X^0>X^1)\,|\,l'\rangle\in\mathcal{T}_\Pi}$$

$$\text{(App)}\frac{\langle l\mid \mathit{append}(x,X)\mid l'\rangle\in\mathcal{T}_P}{\begin{array}{c}\langle l\,|\,\mathit{assume}(X^0>X^1);X^0:=x;X^1:=x\,|\,l'\rangle\in\mathcal{T}_\Pi\\ \langle l\,|\,\mathit{assume}(X^0\leq X^1);\mathit{assert}(x>X^1);X^1:=x\,|\,l'\rangle\in\mathcal{T}_\Pi.\end{array}}\qquad\text{(AF)}\frac{\langle l\mid \mathit{prepend}(x,X)\mid l'\rangle\in\mathcal{T}_P}{\begin{array}{c}\langle l\,|\,\mathit{assume}(X^0>X^1);X^0:=x;X^1:=x\,|\,l'\rangle\\ \langle l\,|\,\mathit{assume}(X^0\leq X^1);\mathit{assert}(x<X^0);X^0:=x\,|\,l'\rangle\end{array}}$$

Figure 9. Construction of the transitions $\mathcal{T}_\Pi$ of Π from $P=\langle \mathit{Loc}_P,\mathit{Var}_P,l_{0,P},\mathcal{T}_P\rangle$

Our construction of Π ensures that any such formula $\mathcal{H}_l$ can be interpreted as a mapping from traces to sets of decisions made along those traces. The intuition behind this interpretation is as follows. Let ρ be a program trace, which ends in a state (l,σ). For any $\mathtt{A}$, consider the set of models of $\mathcal{H}_l$ which send each variable to its value in σ and in which the $\#_{\mathtt{A}}$ variables have been initialized ($\mathit{flag}_{\mathtt{A}}$ is true). Each such model M corresponds to an $\mathtt{A}$-decision, namely, the decision $\mathtt{A}(M(\#^1_{\mathtt{A}}),...,M(\#^m_{\mathtt{A}}))$ (where $M(\#^i_{\mathtt{A}})$ denotes M's interpretation of the variable $\#^i_{\mathtt{A}}$). The set of decisions obtained from models of $\mathcal{H}_l$ in this way is a superset of the set of decisions made along ρ. The precise statement of this interpretation of $\mathcal{H}_l$ is the following lemma.

Lemma 1. *Suppose that ρ is an execution of P, ending in a state (l,σ). Let id be any mapping from the objects accessed in ρ to integer IDs such that IDs increase along every list in the initial state of ρ. Let $\mathtt{A}\in\mathtt{Q}$, and let V be the set of all variables in Var_Π except the $\#_{\mathtt{A}}$ variables. Define*

$$H_{l,\mathtt{A}}=\exists V.\Big(\mathcal{H}_l\wedge \mathit{flag}_{\mathtt{A}}=\mathit{true}\wedge\bigwedge_{x\in \mathit{Var}_P} x=id(\sigma(x))\Big).$$

Then for any $\mathtt{A}(p_1,...,p_m)\in \mathit{Dec}(\rho)$, the model M that sends each $\#^i_{\mathtt{A}}$ to p_i is a model of $H_{l,\mathtt{A}}$ (and similarly for any $\overline{\mathtt{A}}(p_1,...,p_m)\in \mathit{Dec}(\rho)$).

Proof. Let $\rho=(l_0,\sigma_0)\xrightarrow{t_0}(l_1,\sigma_1)\cdots(l_n,\sigma_n)$, with $l_n=l$. Let $\alpha(p_1,...,p_m)\in \mathit{Dec}(\rho)$. Then there is some i such that

- $t_i=\langle l_i\,|\,\alpha(x_1,\ldots,x_m)\,|\,l_{i+1}\rangle$
- for all $j\in\{1,...,m\}$, $p_j=\sigma_i(x_j)$.

With the exception of query, append, and prepend transitions, for any transition of P, there is a unique sequence of corresponding transitions in Π, given in Fig. 9. For append and prepend transitions, there are two corresponding transitions, but their guards are disjoint so there is a unique choice for the translation of these transitions when the pre-state is fixed. For query transitions there are two corresponding transition sequences given by the (Query-1) and (Query-2) rules. Consider the execution ρ_Π of Π which executes the sequence of transitions corresponding to ρ, using the (Query-1) rule for the query transition t_i and (Query-2) for all other query transitions along ρ, (and which chooses the "correct" non-deterministic updates to the list variables to make the executions ρ and ρ_Π agree).

Let (l,σ_Π) be the final state of ρ_Π. Then σ_Π sends each object-valued variable x to $id(\sigma(x))$, sends each $\#^i_{\mathtt{A}}$ to $id(p_i)$, and sends $\mathit{flag}_{\mathtt{A}}$ to true. Since $\mathcal{H}_l$ is an invariant at location l, we must have $\sigma_\Pi\models\mathcal{H}_l$. It follows that M, the restriction of σ_Π to the $\#^i_{\mathtt{A}}$ variables, is a model of $H_{l,\mathtt{A}}$. □

Now we describe how to compute current-decision invariants. Consider a query location l with an outgoing query transition $\langle l\mid\alpha(x_1,\ldots,x_m)\mid l'\rangle$, where $\alpha\in\{\mathtt{A},\overline{\mathtt{A}}\}$ for some $\mathtt{A}\in\mathtt{Q}$. By the syntactic assumptions laid out in Sec. 2, l has no other outgoing transition. The current-decision invariant at l can be computed from $\mathcal{H}_l$ as:

$$d_l=(\exists\#.\mathcal{H}_l)\wedge\left(\bigwedge_{i=1}^m(x_i=\#^i_{\mathtt{A}})\right)$$

where $\exists\#$ denotes the existential quantification of each #-variable. Intuitively, d_l constrains the relationships between the program variables and the objects involved in the "new" decision made at l.

4.2 Proving Consistency of Decisions

Once the history ($\mathcal{H}_l$) and current-decision (d_l) invariants have been generated for each query location l, consistency verification can be reduced to the problem of proving *false* cannot be derived from the combination of $\mathcal{H}_l$, d_l, and the world axioms $\mathcal{R}$. Our reduction makes use of our inductive assumption that there is no derivation of *false* using only decisions from $\mathcal{H}_l$.

Now we show how to automate this proof obligation.

Decision proof systems

For expository purposes, we will begin by defining *decision proof systems*, a Datalog-like normal form for the world axioms $\mathcal{R}$. After, we proceed to the construction of a logic program that can be used to verify consistency of the program P.

Normalized axioms can be of two forms:

- *Generators:* These rules derive new decisions that are logical consequences of decisions taken by the program. Formally, a

generator is a rule of the form

$$\alpha(y_1, \ldots, y_m) \leftarrow \alpha_1(y_{11}, \ldots, y_{1m}) \wedge \cdots \wedge \alpha_k(y_{k1}, \ldots, y_{km})$$

where α, α_i are in $(\mathtt{Q} \cup \overline{\mathtt{Q}})$, and the y_{ij}'s range over objects.

- *Violators:* These rules identify direct contradictions between decisions that have either been made by the program or derived by the generators. Such a rule has the form

$$\bot \leftarrow \mathtt{A}_i(y_1, \ldots, y_m) \wedge \overline{\mathtt{A}_i}(y_1, \ldots, y_m).$$

where $\bot$ is a special symbol (intuitively indicating *false*), $\mathtt{A} \in \mathtt{Q}$ and the y_i's range over objects.

We interpret the relations in the above rules as sets of decisions, and the rule as a way to derive new decisions or derive contradictions.

Now we describe how to construct a decision proof system $\mathcal{M}_\mathcal{R}$ from a set of axioms $\mathcal{R}$. First, we construct the formula $\bigwedge_{R \in \mathcal{R}} R$ and convert it into the conjunctive normal form (per our current notation, we assume that the initial quantifiers are omitted from formulas in $\mathcal{R}$). Let the resultant formula have clauses $C_1, \ldots, C_n$.

Now consider any clause $C_j = t_1 \vee \cdots \vee t_k$ of the formula constructed in the previous step. Here, each t_i is either of the form $A_i(y_{i1}, \ldots, y_{im_1})$ or of the form $\neg A_i(y_{i1}, \ldots, y_{im})$. We define two literals $\mathtt{t}_i$ and $\overline{\mathtt{t}_i}$ for each t_i. If $t_i = A_i(y_{i1}, \ldots, y_{im})$, then we have

$$\mathtt{t_i} = \mathtt{A}_i(y_{i1}, \ldots, y_{im}) \qquad \overline{\mathtt{t_i}} = \overline{\mathtt{A}_i}(y_{i1}, \ldots, y_{im}).$$

If $t_i = \neg A_i(y_{i1}, \ldots, y_{im})$, then we have

$$\mathtt{t_i} = \overline{\mathtt{A}_i}(y_{i1}, \ldots, y_{im}) \qquad \overline{\mathtt{t_i}} = \mathtt{A}_i(y_{i1}, \ldots, y_{im}).$$

$\mathcal{M}_\mathcal{R}$ is defined to be the least set of rules such that:

1. For each clause $C_j = t_1 \vee \cdots \vee t_k$ and each $r \in \{1, ..., k\}$, $\mathcal{M}_\mathcal{R}$ has the generator

$$\mathtt{t}_r \leftarrow \overline{\mathtt{t}_1} \wedge \cdots \wedge \overline{\mathtt{t}_{r-1}} \wedge \overline{\mathtt{t}_{r+1}} \wedge \cdots \wedge \overline{\mathtt{t}_k}.$$

2. For each predicate $\mathtt{A} \in \mathtt{Q}$, $\mathcal{M}_\mathcal{R}$ has the violator

$$\bot \leftarrow \mathtt{A}(y_1, ..., y_m) \wedge \overline{\mathtt{A}}(y_1, ..., y_m).$$

Example 3. Consider once again the world Θ_{tc} of Example 1, which supports the predicate *Left* over 1-D points. The following decision proof system $\mathcal{M}_{\mathcal{R}_{tc}}$ is obtained from the axioms of Θ_{tc}:

$$\begin{array}{rcl}
\overline{\mathtt{Left}}(y_1, y_2) & \leftarrow & \mathtt{Left}(y_2, y_1) \\
\mathtt{Left}(y_1, y_2) & \leftarrow & \overline{\mathtt{Left}}(y_2, y_1) \\
\mathtt{Left}(y_1, y_3) & \leftarrow & \mathtt{Left}(y_1, y_2) \wedge \mathtt{Left}(y_2, y_3) \\
\overline{\mathtt{Left}}(y_2, y_3) & \leftarrow & \mathtt{Left}(y_1, y_2) \wedge \overline{\mathtt{Left}}(y_1, y_3) \\
\overline{\mathtt{Left}}(y_1, y_2) & \leftarrow & \mathtt{Left}(y_2, y_3) \wedge \overline{\mathtt{Left}}(y_1, y_3) \\
\bot & \leftarrow & \mathtt{Left}(y_1, y_2) \wedge \overline{\mathtt{Left}}(y_1, y_2) \quad \square
\end{array}$$

Proving consistency

We now show how to construct (from the current-decision invariants, and history invariants, and the decision proof system) a logic program which derives the symbol $\bot$ if P is inconsistent. If this logic program *cannot* derive $\bot$, this proves that P is consistent.

First, we provide some intuition. Let l be a query location. We compute for each predicate $\mathtt{A} \in \mathtt{Q}$ a relation $Hist_{l,\mathtt{A}}$ which relates the final states of program executions that end at l to the decisions of the form $\mathtt{A}(...)$ or $\overline{\mathtt{A}}(...)$ which are logical consequences of the decisions made along those executions.[4] We also compute for each $\alpha \in \mathtt{Q} \cup \overline{\mathtt{Q}}$ a relation $New_{l,\alpha}$, which is similar to $Hist_{l,\mathtt{A}}$ except that it includes the decision made at l, but not necessarily the decisions which are consequences of decisions in $Hist_{l,\mathtt{A}}$.

The sets $Hist_{l,\mathtt{A}}$ and $New_{l,\alpha}$ can be described using a set of Datalog-style rules, given in the following. We use $\vec{x} = \langle x_1, \ldots, x_n \rangle$ to denote the list of the variables corresponding to object-valued variables of P. These variables will be "threaded through" each of the inference rules, so they may be thought of as symbolic constants that help to mediate relationships between the current decision and previous decisions. Alternately, they may be thought of as a way of adding a degree of path-sensitivity to our analysis: we may think of each valuation of the variables $\vec{x}$ as a "universe", and threading $\vec{x}$ through each inference rule as a way of keeping universes separate.

We begin with rules that "initialize" $Hist_{l,\mathtt{A}}$ and $New_{l,\alpha}$ using the history and current-decision invariants:

$$\begin{array}{rcl}
Hist_{l,\mathtt{A}}(y_1, \ldots, y_m, \vec{x}) & \leftarrow & \mathcal{H}_l[\#^1_\mathtt{A} \mapsto y_1; \ldots; \#^m_\mathtt{A} \mapsto y_m] \\
& & \wedge\, flag_\mathtt{A} \\
New_{l,\alpha}(y_1, \ldots, y_m, \vec{x}) & \leftarrow & d_l[\#^1_\mathtt{A} \mapsto y_1; \ldots; \#^m_\mathtt{A} \mapsto y_m] \\
& & \text{if } l \in Loc^\alpha_\Pi \text{ and } \alpha \in \{\mathtt{A}, \overline{\mathtt{A}}\} \\
New_{l,\alpha}(y_1, \ldots, y_m, \vec{x}) & \leftarrow & \emptyset \qquad \text{otherwise}
\end{array}$$

Now we give rules to derive the consequences of decisions in d_l taken together with decisions in $\mathcal{H}_l$. For each generator $R_g \in \mathcal{M}_\mathcal{R}$ of the form

$$\alpha(y_1, \ldots, y_m) \leftarrow \alpha_1(y_{11}, \ldots, y_{1m}) \wedge \cdots \wedge \alpha_k(y_{k1}, \ldots, y_{km}),$$

where $\alpha \in \{\mathtt{A}, \overline{\mathtt{A}}\}$ and for all i, $\alpha_i \in \{\mathtt{A_i}, \overline{\mathtt{A_i}}\}$, we add:

$$\begin{array}{rcl}
Hist_{l,\mathtt{A}}(y_1, \ldots, y_m, \vec{x}) & \leftarrow & Hist_{l,\mathtt{A}_1}(y_{11}, \ldots, y_{1m}, \vec{x}) \wedge \ldots \\
& & \wedge Hist_{l,\mathtt{A}_k}(y_{k1}, \ldots, y_{km}, \vec{x}) \\
New_{l,\alpha}(y_1, \ldots, y_m, \vec{x}) & \leftarrow & \Omega_{l,\alpha_1}(y_{11}, \ldots, y_{1m}, \vec{x}) \wedge \ldots \\
& & \wedge \Omega_{l,\alpha_k}(y_{k1}, \ldots, y_{km}, \vec{x})
\end{array}$$

where $\Omega_{l,\alpha_i} \in \{New_{l,\alpha_i}, Hist_{l,\mathtt{A}_i}\}$ and such that at least one of the Ω_{l,α_i}'s is New_{l,α_i}. The requirement that at least one of the Ω_{l,α_i}'s is New_{l,α_i} ensures that $New_{l,\alpha}$ does not include decisions that are *solely* derived from decisions in $\mathcal{H}_l$. This is used to encode our inductive assumption that there are no inconsistencies in $\mathcal{H}_l$.

Finally, for each predicate $\mathtt{A} \in \mathtt{Q}$, we generate three violator rules that detect direct contradictions:

$$\bot \leftarrow New_{l,\mathtt{A}}(y_1, ..., y_m, \vec{x}) \wedge Hist_{l,\mathtt{A}}(y_1, ..., y_m, \vec{x})$$

$$\bot \leftarrow New_{l,\overline{\mathtt{A}}}(y_1, ..., y_m, \vec{x}) \wedge Hist_{l,\mathtt{A}}(y_1, ..., y_m, \vec{x})$$

$$\bot \leftarrow New_{l,\mathtt{A}}(y_1, ..., y_m, \vec{x}) \wedge New_{l,\overline{\mathtt{A}}}(y_1, ..., y_m, \vec{x})$$

Let $\mathcal{M}_{\Pi,\mathcal{R}}$ be the system of all rules constructed as above. We have the following lemma:

Lemma 2. *Suppose that program P is inconsistent. Then $\bot$ belongs to the least fixpoint of $\mathcal{M}_{\Pi,\mathcal{R}}$.*

Proof. Let $\rho = (l_0, \sigma_0) \xrightarrow{t_0} (l_1, \sigma_1) \cdots (l_n, \sigma_n)$ be an execution of the program P such that $\mathcal{M}_\mathcal{R}$ can derive $\bot$ from $Dec(\rho)$. Without loss of generality, we may assume no proper prefix of ρ satisfies this property. It follows that t_{n-1}, the last transition of ρ, must be a decision transition $\langle l_{n-1} \mid \alpha(z_1, ..., z_m) \mid l_n \rangle$ for some $\alpha, z_1, \ldots, z_m$. For notational convenience, we use l to denote l_{n-1}, and use ρ' to denote the trace ρ with its last action omitted. Let id be any mapping form the object accessed in ρ to integer IDs such that IDs increase along every list in the initial state σ_0. Let $\vec{x}$ be the vector of the object-valued variables of P, and define

$$\vec{p} = \langle id(\sigma_{n-1}(x_1)), ..., id(\sigma_{n-1}(x_n)) \rangle$$

Let T be a derivation tree for $\bot$ using the inference rules in $\mathcal{M}_\mathcal{R}$, with the elements of $Dec(\rho)$ taken as axioms.

[4] Note that since the outcome of decisions are uncertain, we need not compute a corresponding $Hist_{l,\overline{\mathtt{A}}}$ relations, because it would be they would be identical to the $Hist_{l,\mathtt{A}}$ relations.

Initialization:

$$New_{\texttt{Left}}(y_1, y_2, x_1, x_2) \leftarrow x_1 = y_1 < y_2 = x_2$$
$$New_{\overline{\texttt{Left}}}(y_1, y_2, x_1, x_2) \leftarrow x_1 = y_1 < y_2 = x_2$$
$$Hist_{\texttt{Left}}(y_1, y_2, x_1, x_2) \leftarrow y_1 < y_2 \leq x_1 < x_2$$

Totality:

$$Hist_{\overline{\texttt{Left}}}(y_1, y_2, x_1, x_2) \leftarrow Hist_{\texttt{Left}}(y_2, y_1, x_1, x_2)$$
$$Hist_{\texttt{Left}}(y_1, y_2, x_1, x_2) \leftarrow Hist_{\overline{\texttt{Left}}}(y_2, y_1, x_1, x_2)$$
$$New_{\overline{\texttt{Left}}}(y_1, y_2, x_1, x_2) \leftarrow New_{\texttt{Left}}(y_2, y_1, x_1, x_2)$$
$$New_{\texttt{Left}}(y_1, y_2, x_1, x_2) \leftarrow New_{\overline{\texttt{Left}}}(y_2, y_1, x_1, x_2)$$

Violators:

$$\bot \leftarrow Hist_{\texttt{Left}}(y_1, y_2, x_1, x_2) \wedge New_{\overline{\texttt{Left}}}(y_1, y_2, x_1, x_2)$$
$$\bot \leftarrow New_{\texttt{Left}}(y_1, y_2, x_1, x_2) \wedge Hist_{\overline{\texttt{Left}}}(y_1, y_2, x_1, x_2)$$
$$\bot \leftarrow New_{\texttt{Left}}(y_1, y_2, x_1, x_2) \wedge New_{\overline{\texttt{Left}}}(y_1, y_2, x_1, x_2)$$

Transitivity:

$$Hist_{\texttt{Left}}(y_1, y_3, x_1, x_2) \leftarrow Hist_{\texttt{Left}}(y_1, y_2, x_1, x_2) \wedge Hist_{\texttt{Left}}(y_2, y_3, x_1, x_2)$$
$$Hist_{\overline{\texttt{Left}}}(y_2, y_3, x_1, x_2) \leftarrow Hist_{\texttt{Left}}(y_1, y_2, x_1, x_2) \wedge Hist_{\overline{\texttt{Left}}}(y_1, y_3, x_1, x_2)$$
$$Hist_{\overline{\texttt{Left}}}(y_1, y_2, x_1, x_2) \leftarrow Hist_{\texttt{Left}}(y_1, y_2, x_1, x_2) \wedge Hist_{\overline{\texttt{Left}}}(y_1, y_3, x_1, x_2)$$
$$New_{\texttt{Left}}(y_1, y_3, x_1, x_2) \leftarrow New_{\texttt{Left}}(y_1, y_2, x_1, x_2) \wedge New_{\texttt{Left}}(y_2, y_3, x_1, x_2)$$
$$New_{\texttt{Left}}(y_1, y_3, x_1, x_2) \leftarrow Hist_{\texttt{Left}}(y_1, y_2, x_1, x_2) \wedge New_{\texttt{Left}}(y_2, y_3, x_1, x_2)$$
$$New_{\texttt{Left}}(y_1, y_3, x_1, x_2) \leftarrow New_{\texttt{Left}}(y_1, y_2, x_1, x_2) \wedge Hist_{\texttt{Left}}(y_2, y_3, x_1, x_2)$$
$$New_{\overline{\texttt{Left}}}(y_2, y_3, x_1, x_2) \leftarrow New_{\texttt{Left}}(y_1, y_2, x_1, x_2) \wedge New_{\overline{\texttt{Left}}}(y_1, y_3, x_1, x_2)$$
$$New_{\overline{\texttt{Left}}}(y_2, y_3, x_1, x_2) \leftarrow Hist_{\texttt{Left}}(y_1, y_2, x_1, x_2) \wedge New_{\overline{\texttt{Left}}}(y_1, y_3, x_1, x_2)$$
$$New_{\overline{\texttt{Left}}}(y_2, y_3, x_1, x_2) \leftarrow New_{\texttt{Left}}(y_1, y_2, x_1, x_2) \wedge Hist_{\overline{\texttt{Left}}}(y_1, y_3, x_1, x_2)$$
$$New_{\overline{\texttt{Left}}}(y_1, y_2, x_1, x_2) \leftarrow New_{\texttt{Left}}(y_1, y_2, x_1, x_2) \wedge New_{\overline{\texttt{Left}}}(y_1, y_3, x_1, x_2)$$
$$New_{\overline{\texttt{Left}}}(y_1, y_2, x_1, x_2) \leftarrow Hist_{\texttt{Left}}(y_1, y_2, x_1, x_2) \wedge New_{\overline{\texttt{Left}}}(y_1, y_3, x_1, x_2)$$
$$New_{\overline{\texttt{Left}}}(y_1, y_2, x_1, x_2) \leftarrow New_{\texttt{Left}}(y_1, y_2, x_1, x_2) \wedge Hist_{\overline{\texttt{Left}}}(y_1, y_3, x_1, x_2)$$

Figure 10. $\mathcal{M}_{\Pi,\mathcal{R}_{tc}}$ for the program in Example 2.

For any subtree T' of T, we say that T' is *new* if one of its axioms is $\alpha(\sigma_{n-1}(z_1), ..., \sigma_{n-1}(z_m))$; otherwise, we say T' is *old*. We first prove that for any *old* subtree T' of T with root $\alpha'(q_1, ..., q_m)$, we have

$$\langle id(\sigma_{n-1}(z_1)), ..., id(\sigma_{n-1}(z_m)), \vec{p}\rangle \in Hist_{l,\texttt{A}}$$

(where $\alpha' \in \{\texttt{A}, \overline{\texttt{A}}\}$).

- Base case: T' is an axiom, so we must have

  $$\alpha'(p_1, ..., p_m) \in Dec(\rho')$$

  It follows from Lemma 1 and the fact that $Hist_{l,\texttt{A}}$ is closed under the following inference rule:

  $$Hist_{l,\texttt{A}}(y_1, \ldots, y_m, \vec{x}) \leftarrow \mathcal{H}_l[\#_1^{\texttt{A}} \mapsto y_1; \ldots; \#_m^{\texttt{A}} \mapsto y_m] \wedge \mathit{flag}_{\texttt{A}}$$

  that $\langle p_1, \ldots, p_m, \vec{p}\rangle \in Hist_{l,\texttt{A}}$.
- Inductive step: T' ends with the application of a generator rule $R_g \in \mathcal{M}_{\mathcal{R}}$, say

  $$\alpha(q_1, \ldots, q_m) \leftarrow \alpha_1(q_{11}, \ldots, q_{1m}) \wedge \cdots \wedge \alpha_k(q_{k1}, \ldots, q_{km})$$

  (where $\alpha \in \{\texttt{A}, \overline{\texttt{A}}\}$ and $\alpha_i \in \{\texttt{A}_\texttt{i}, \overline{\texttt{A}_\texttt{i}}\}$). Let $T_1, \ldots, T_k$ be the subtrees of T corresponding to the premises of R_g. Since each of $T_1, \ldots, T_k$ is a proper subtree of T, we have $\langle q_{i1}, ..., q_{im}, \vec{p}\rangle \in Hist_{l,\texttt{A}_\texttt{i}}$ for each i by our inductive hypothesis. Since $Hist_{l,\texttt{A}}$ is closed under the following inference rule:
  $$Hist_{l,\texttt{A}}(y_1, \ldots, y_m, \vec{x}) \leftarrow Hist_{l,\texttt{A}_1}(y_{11}, \ldots, y_{1m}, \vec{x}) \wedge \cdots \wedge Hist_{l,\texttt{A}_k}(y_{k1}, \ldots, y_{km}, \vec{x})$$
  we have that $\langle q_1, \ldots, q_m, \vec{p}\rangle \in Hist_{l,\texttt{A}}$.

We can use a similar argument to show that for any *new* subtree T' of T with root $\alpha'(q_1, ..., q_m)$, we have $\langle q_1, ..., q_m, \vec{p}\rangle \in New_{l,\alpha'}$.

The final inference rule used in T must be the application of a violator rule. At least one of the immediate subtrees of T must be *new*, since if they are all old, the axioms of T are contained in $Dec(\rho')$ which violates our minimality assumption. It follows that we may use one of the violator rules to derive $\bot$, and therefore $\bot$ belongs to the least fixpoint of $\mathcal{M}_{\Pi,\mathcal{R}}$. □

Example 4. The logic program $\mathcal{M}_{\Pi,\mathcal{R}_{tc}}$ constructed for the program program P_{tc} in Example 2 (with respect to the specification of the world Θ_{tc}) is given in Figure 10. Since the only query location is Line 3 of the program, we omit location subscripts in this figure. μZ is able to determine that this program cannot derive $\bot$, so the program P_{tc} is certified consistent.

Summarizing, our algorithm CHECK-CONSISTENCY for verifying the consistency of P is as follows. First we construct an integer abstraction Π and generate numerical invariants using an abstract interpreter. We check that each assertion in the program is safe according to the invariants produced by the abstract interpreter; if a assertion violation is found (indicating a violation of our sorted list abstraction), we report P to be possibly inconsistent. Second, we construct the logic program $\mathcal{M}_{\Pi,\mathcal{R}}$ and check if $\mathcal{M}_{\Pi,\mathcal{R}}$ can derive $\bot$. If it can, we report P to be possibly inconsistent, otherwise we report P to be consistent.

By Lemma 2, we immediately have:

Theorem 1. *If the algorithm* CHECK-CONSISTENCY *certifies a program P to be consistent, then P is consistent.*

4.3 Limitations

Our algorithm is sound but incomplete. There are several possible sources of loss of precision here: (1) the abstraction of lists by pairs of integers, which require to maintain the invariant that the lists are sorted by IDs; (2) the limitations of the abstract interpretation used to compute the history and current-decision invariants; and (3) the incompleteness of μZ. In our empirical experience so far, these inaccuracies have not mattered much. The only substantial source of imprecision has been the abstraction of lists by pairs, but as elaborated in the next section, only in two of our benchmark examples does this pose a problem. Also, one could presumably eliminate this issue with a more sophisticated list abstraction.

5. Implementation and evaluation

We have implemented our approach and experimented with a collection of standard geometric algorithms. In this section, we report on the results. It is important to bear in mind that (1) it is very difficult to declare an algorithm consistent/inconsistent by a light inspection (even by an expert user), and (2) it is even more difficult to manually prove it is consistent.

5.1 Implementation

Our tool is implemented in OCaml on top of a CIL [23] frontend. In addition to taking a C file as input, the tool requires the user to supply a proof system $\mathcal{M}$ (in the form of a set of rules; see Fig. 4). The tool operates in three phases. First, an integer C program is generated from the original program. Then, numerical invariants are generated for the integer program using an abstract domain we will describe below. Then, for each control point in the program at which a decision is made, two Datalog programs are generated (using the invariants from the first phase and the decision proof system $\mathcal{M}$): one Datalog program for the case that the new decision is positive, and one for the case that its negative. Lastly, we use μZ [14] to check whether a query in any of these Datalog programs succeeds; if all queries fail, then the input program is consistent.

The abstract domain we use to generate numerical invariants is a partitioned octagon domain. Partitioning is a technique used to reduce the precision lost in an abstract domain due to an imprecise join operator [3]. An element of this domain is a partial function f from a finite set of *cells* to octagons. The cells form a covering of the state space, and we require that for every cell c, the concretization of the octagon $f(c)$ lies inside c; such a function represents a finite disjunction of octagons, where each octagon belongs to a different cell. In our partitioning scheme, the cells corresponds to sets of program variable equalities. We built our partitioned octagon domain on top of the octagon domain implemented in APRON [2].

5.2 Benchmarks

We collected a set of geometric algorithms from computational geometry textbooks [8, 9]. These include several convex hull algorithms, a few point location algorithms, and a few triangulation algorithms. Algorithms like this are the core building blocks of geometric libraries such as CGAL [1].

The algorithms are presented in the books as pseudocode, and therefore, there is occasionally more than one way of implementing them. Note that we are not referring to implementation details such as choice of data structures; we included more than one version of the same algorithm whenever there was an algorithmic choice about list traversal strategies and other similar notions that can make a difference in the outcome of our verification algorithm. Below, we provide a high level summary of the benchmarks.

Convex Hulls. We have already discussed two convex hull algorithms: SIMPLECONVEXHULL and GRAHAMSCAN in Sec. 3. The former is not consistent. We discuss the details of the consistency check for the latter in the following. The set of benchmarks consists of three other convex hull algorithms: the Gift Wrapping algorithm, the Incremental Hull algorithm, and Fortune's algorithm. For the Gift Wrapping algorithm, we present two different variations on how the point set is traversed. The incremental hull is a recursive algorithm which has three variations on how the recursive step is performed. Fortune's algorithm uses a major subroutine that computes a half of a convex hull (which is effectively called twice); we include this subroutine as a separate benchmark. Note that Fortune's consistency is not trivially implied by the consistency of this subroutine. In [8], a slight variation of Fortune's original algorithm [10] was presented which is inconsistent (in contrast to the original algorithm being designed specifically to be consistent). Since all other examples were collected from textbooks, we included this version in the set of benchmarks as well.

Triangulation. A *triangulation* of a planar point set P is a subdivision of the plane determined by a maximal set of non-crossing edges whose vertex set is P. The word maximal in the definition indicates that any edge not in triangulation must intersect the interior of at least one of the edges in the triangulation.

(Rule G1) $\texttt{Lturn}(y,z,x) \leftarrow \texttt{Lturn}(x,y,z)$

(Rule G2) $\texttt{Lturn}(y,x,z) \leftarrow \overline{\texttt{Lturn}}(x,y,z)$

(Rule G4) $\texttt{Lturn}(x,y,z) \leftarrow \texttt{Lturn}(x,y,s) \wedge \texttt{Lturn}(y,z,s) \wedge \texttt{Lturn}(z,x,s)$

(Rule G5) $\texttt{Lturn}(x,y,t) \leftarrow \texttt{Lturn}(x,y,z) \wedge \texttt{Lturn}(x,y,s) \wedge \texttt{Lturn}(y,z,s) \wedge \texttt{Lturn}(y,z,t) \wedge \texttt{Lturn}(s,y,t)$

(Rule V1) $\perp \leftarrow \texttt{Lturn}(x,y,z) \wedge \overline{\texttt{Lturn}}(x,y,z)$

Figure 11. Decision proof system for the $Lturn$ predicate.

In our benchmarks we have Triangle Splitting, Incremental triangulation, and Delaunay triangulation. Triangle Splitting makes use of a *point location* algorithm (see below), and therefore, its consistency depends on the consistency of the underlying point location algorithm (i.e., if the latter is inconsistent, then the former is also inconsistent). Since we have two different *point location* algorithms in our benchmarks, we have two versions of the Triangle Splitting algorithm respectively. Incremental triangulation is algorithmically similar to the incremental convex hull and we similarly included three versions of it depending on the recursing strategy. Both Triangle Splitting and Incremental triangulation algorithms use the same `Lturn` predicate that is used by the convex hull algorithms.

Delaunay triangulation is a specific type of triangulation that has special uses in many areas, e.g. in terrain reconstruction. These algorithms use an operation called *flipping* to turn an arbitrary triangulation into a Delaunay one or to construct a Delaunay triangulation incrementally. Flipping relies on a new predicate called `InCircle` that tests if a given point lies within a given circle. The predicate comes with its own complete set of axioms [17] — for brevity, we omit the definition of this new predicate and its corresponding axioms here and refer the interested reader to [17]. Unfortunately, all textbook Delaunay triangulation algorithms are inconsistent, and this inconsistency leads to nontermination under some inputs.

Point Location. In a *point location* algorithm, the input is usually a triangulation (of a point set) and a point p, and the goal is to find the particular triangle that contains p. Point location queries arise in various settings, such as finding one's location on a map and computer graphics (it is an essential part of ray-tracing algorithms). In the example that we now consider, the location is a target triangle. There are various inconsistent point location algorithms; Fortune [10] proposed a consistent algorithm for point location. We included Fortune's algorithm and a standard inconsistent algorithm that is routinely used in ray tracer implementations.

Axioms and decision proof system. All our benchmarks use the $Lturn$ predicate introduced earlier; we assume an axiomatization as in Fig. 4 for this predicate. Fig. 11 offers a sketch of the decision proof system derived from these axioms. Here, Rules G1 and G2 represent the nondegeneracy case of axiom 3 in Fig. 4. Rules G1 corresponds to the cyclic symmetry and antisymmetry axioms, while rules G4 and G5 represent the interiority and transitivity axioms. Rule V1 is the violator rule for the `Lturn` predicate.

5.3 Experimental Results

Table 1 presents the result of the experiments carried out using our tool. The upper, middle, and the lower part of the table respectively show the results for a variety of convex hull, point location, and triangulation algorithms.

The first two columns (after the benchmark names) indicate whether the benchmark is consistent and whether our tool managed to prove it consistent. A "NO" answer by the tool means that a counterexample was found for the consistency of the *integer* C

Benchmark	Consistent?	Proved Consistent?	Invariant Generation Time	Consistency Analysis Time	Invariant Size
SlowHull	NO	NO	14s	2s	780 (2)
Graham Scan	YES	YES	22s	38s	4188 (4)
Gift Wrapping v1	NO	NO	24s	2s	1494 (2)
Gift Wrapping v2	NO	NO	—	—	—
Incremental Hull v1	YES	YES	1m5s	44s	4454 (2)
Incremental Hull v2	YES	YES	1m1s	2m23s	5904 (2)
Incremental Hull v3	YES	NO	1m35s	22s	22658 (4)
Fortune (half) Hull	YES	YES	10s	13s	774 (2)
Fortune Hull	YES	YES	2m43s	5m17s	30088 (8)
Fortune Hull (t)	NO	NO	48s	10s	10440 (6)
Point Location v1	YES	YES	25s	2m28s	2608 (4)
Point Location v2	NO	NO	1m26s	2s	566 (6)
Hull Triangle Location	NO	NO	1m38s	4s	2202 (6)
Incremental Triangulation v1	YES	YES	1m12s	44s	4454 (2)
Incremental Triangulation v2	YES	YES	1m8s	2m23s	5904 (2)
Incremental Triangulation v3	YES	NO	1m38s	31s	22658 (2)
Triangle Splitting v1.1	YES	YES	40s	2m28s	2608 (4)
Triangle Splitting v1.2	NO	NO	1m45s	2s	566 (6)
Triangle Splitting v2	NO	NO	—	—	—
Delaunay Triangulation v1	NO	NO	6s	0s	152 (2)
Delaunay Triangulation v2	NO	NO	—	—	—

Table 1. Automated Consistency Verification Results. Invariant Generation Time refers to the time that it takes for the invariants to be generated for the integer program. Consistency Analysis time refers to the time that it takes μZ to prove the consistency claims. Invariant Size refers to the size of the invariants (in number of conjuncts); where there is more than one decision location in the program, we add the invariants up into a total number, but indicate the number of claims within parentheses.

program. However, since this integer program is an overapproximation of the behaviour of the original C program, the counterexample may not be a real one. Note that since our tool is *sound*, a NO/YES combination is not a possibility for these two columns. And, since it is not complete, a YES/NO option is a possibility.

For some of the benchmarks, no times are reported. This is because the benchmark was declared inconsistent in the integer program generation phase, since the proper ordering constraints were not followed while manipulating lists in the program. Interestingly, in all such cases, the program is indeed inconsistent.

Summary of Results. Out of the 21 benchmarks, 9 are consistent. There are 3 inconsistent benchmarks that are declared inconsistent when the integer program generation fails to generate a meaningful program. Note that the μZ terminates much faster in the case where an inconsistency is found since it finds a satisfiable assignment relatively quickly in these cases, whereas for the consistent cases, it has to effectively prove the lack of a satisfiable assignment.

For almost all (except two) benchmarks, if the program is consistent, our tool succeeds to prove it. For a variation of Incremental convex hull (v3) and the corresponding incremental triangulation algorithm (v3), the list abstraction is too coarse to prove the example consistent. The reason for inconsistency is that in the *integer* program, the program is practically making repeat queries. In our uncertain semantics, we assume that repeat queries may get different result from the original ones. If we drop this assumption, then the tool can be easily tuned to prove these two benchmarks consistent as well. The fact that we can prove almost all consistent examples correct using our tool suggests that our proposed abstraction technique is powerful despite its simplicity.

5.4 Detailed Discussion of Graham Scan

We discussed the Graham Scan algorithm as an example in Sec. 3. Here we provide a more detailed consistency argument, to provide some intuition why our tool succeeds in proving it consistent.

First, an integer program is generated from this program. Here, we assume that the points in S are assigned integer IDs in the order that they are fetched from the input list (from 1 to $|S|$). The point p_0 has an ID 0. As stated earlier, we ignore the assumption that the input points are sorted according to their polar angle. From this point on, we just refer to the points by their numeric identifiers.

Now consider line 6 of the Graham Scan code (Fig. 7), where the query $\texttt{Lturn}(u, v, w)$ is made. We note that $u < v < w$ is an invariant here, and from this observation, we can deduce the following history invariant for this location:

$$\mathcal{H}(\#_1, \#_2, \#_3, u, v, w, p_0) \equiv p_0 < \#_1 < \#_2 < \#_3.$$

Now consider the new decision that is about to be made at line 6: $Lturn(u, v, w)$. Invariant generation provides us with the following facts about values of u,v, and w:

$$d_{\overline{\texttt{Lturn}}}(\#_1, \#_2, \#_3, u, v, w, p_0) \equiv u < v < w \wedge w > \#_3 \wedge p_0 < \#_1 < \#_2 < \#_3$$

Note that the actual invariant generated by our tool contains many more conjuncts and variables. Here, we are stripping it to the essential core to demonstrate the reasoning. Also, this is a specific case where the outcome of the query was false. There will be a similar case where the outcome of the query is true,

$$d_{\texttt{Lturn}}(\#_1, \#_2, \#_3, u, v, w, p_0) \equiv u < v < w \wedge w > \#_3 \wedge p_0 < \#_1 < \#_2 < \#_3$$

but the reasoning below will be the same (though done separately for each of) the two cases.

Combining the information for the history and the current decision, it is easy to argue that if the history is already consistent, then the new decision added will not create any new inconsistencies with respect to any of the rules in Fig. 11. Since the third argument w is (provably) strictly greater than the arguments $\#_1, \#_2, \#_3$ which characterize all past decisions, it is easy to see why this is the case. Each argument of the $Lturn$ predicate appears at least twice among the premises of every single rule in Fig. 11; therefore, any decision with fresh new point w cannot be involved in generating any new facts or causing a violation. The actual reasoning for this is done by μZ in our implementation.

Now consider the decision at line 14. Here, w never changes and always points to p_0 (i.e. $id_w = 0$). Naturally, the history for line 6 is also part of the history for line 14. But the history for line 14 also includes the decisions made previously in the same loop.

Our analysis produces the following history invariant:

$$\mathcal{H}(\#_1, \#_2, \#_3, u, v, w, p_0) \equiv (p_0 < \#_1 < \#_2 < \#_3) \vee (p_0 < \#_1 < \#_2 \wedge \#_3 = p_0)$$

where the first disjunct refers to the history based on the decisions made at line 6 and the second disjunct refers to the history based on the decisions made at line 14. The current decision invariant is:

$$d_{\overline{\texttt{Lturn}}}(\#_1, \#_2, \#_3, u, v, w, p_0) \equiv ((u < v) \wedge w = p_0 \wedge (p_0 < \#_1 < \#_2 < \#_3)) \vee ((u < v) \wedge (w = p_0) \wedge (v > \#_2) \wedge (p_0 < \#_1 < \#_2) \wedge \#_3 = p_0))$$

and, similar to above, there is a positive version of this decision. It is a bit harder to *manually* reason about this case. Intuitively, it is easy to reason that the decision, where the last argument is always a fresh new value 0, cannot generate any new facts or violations when considered together with the history decisions that are generated at line 6 (in which 0 never appears).

The argument about why no new violations are created as a result of combining the new decision with the part of history that comes from line 14 is similar to the case of line 6.

6. Related work

The problem of consistency has been studied in depth in the computational geometry literature [15, 16, 21, 26]. This literature has developed several subtly different notions of consistency for geometric programs; our definition coincides with a definition used by, among others, Fortune [10]. However, so far as we know, there is no prior work on static verification of consistency in this area: to the extent that verification is mentioned at all, it is stated to be "too difficult to be realistic" [20]. Instead, existing approaches focus on dynamic techniques that are often based on high-precision library operations. The inherent limitation of such approaches is that they do not reason globally about uncertain decisions made at different program points, and hence do not give end-to-end guarantees of consistency. The only work on formal verification of geometric programs that we know of comes from the theorem proving community [24], and this work does not study consistency.

There is an emerging body of work on reasoning about program behaviour in the presence of uncertainty [4, 5, 18, 19, 25]. However, none of these analyses can reason about the notion of consistency considered in this paper; instead, they focus on quantitative differences in a program's behaviour due to uncertainty. Also not applicable are abstract-interpretation-based techniques for quantifying numerical errors in programs [6, 11], as none of these methods reason about divergence in control flow caused by uncertainty. In contrast, uncertain control flow is perhaps the most central aspect of the geometric programs studied here.

7. Conclusion

We have introduced the problem of automatically verifying the consistency of programs that make decisions under uncertainty, and taken the first steps towards solving the problem. Our solution can automatically verify the consistency of a comprehensive set of algorithms for computing convex hulls and triangulations.

While this paper focused on finding *proofs* of robustness, the dual problem of finding inputs that cause a geometric program to violate robustness is also of interest. Also, we restricted ourselves in this paper to a model of uncertainty where *every* predicate can evaluate nondeterministically. As we showed, even under this highly adversarial execution model, many everyday computations are consistent. However, future work should also study less hostile models of numerical uncertainty. There are several ways to account for this fact. For instance, a possibility is to consider quantitative models of uncertainty properties where a query "flips" with a certain probability, or under certain conditions on the inputs.

References

[1] CGAL, Computational Geometry Algorithms Library. http://www.cgal.org.

[2] J. Bertrand and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *CAV*, pages 661–667, 2009.

[3] François Bourdoncle. Abstract interpretation by dynamic partitioning. *Journal of Functional Programming*, 2(04):407–435, 1992.

[4] S. Chaudhuri, S. Gulwani, and R. Lublinerman. Continuity analysis of programs. In *POPL*, pages 57–70, 2010.

[5] S. Chaudhuri, S. Gulwani, and R. Lublinerman. Continuity and robustness s of programs. *Commun. ACM*, 55(8):107–115, 2012.

[6] L. Chen, A. Miné, J. Wang, and P. Cousot. Interval polyhedra: An abstract domain to infer interval linear relationships. In *SAS*, 2009.

[7] M. De Berg, O. Cheong, and M. Van Kreveld. *Computational geometry: algorithms and applications*. Springer-Verlag, 2008.

[8] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 2000.

[9] S. Devadoss and J. O'Rourke. *Discrete and Computational Geometry*. Princeton University Press, 2011.

[10] S. Fortune. Stable maintenance of point set triangulations in two dimensions. In *FOCS*, pages 494–499, 1989.

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

[12] Ronald L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Inf. Process. Lett.*, 1(4):132–133, 1972.

[13] J. Halpern. *Reasoning about uncertainty*. The MIT Press, 2003.

[14] K. Hoder, N. Bjørner, and L. de Moura. μZ - an efficient engine for fixed points with constraints. In *CAV*, 2011.

[15] C. Hoffmann, J. Hopcroft, and M. Karasick. Towards implementing robust geometric computations. In *SoCG*, pages 106–117, 1988.

[16] C.M. Hoffmann. The problems of accuracy and robustness in geometric computation. *Computer*, 22(3):31–39, 1989.

[17] D.E. Knuth. *Axioms and Hulls (LNCS #606)*. Springer-Verlag, 1992.

[18] R. Majumdar, E. Render, and P. Tabuada. A theory of robust software synthesis. *CoRR*, abs/1108.3540, 2011.

[19] R. Majumdar and I. Saha. Symbolic robustness analysis. *Real-Time Systems Symposium, IEEE International*, 0:355–363, 2009.

[20] K. Mehlhorn. *The reliable algorithmic software challenge RASC*, pages 255–263. 2003.

[21] K. Mehlhorn and C. Yap. *Robust geometric computation*. `http://cs.nyu.edu/~yap/book/egc`, 2011.

[22] D. Monniaux. The pitfalls of verifying floating-point computations. *ACM Trans. Program. Lang. Syst.*, 30(3), 2008.

[23] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC*, pages 213–228, 2002.

[24] D. Pichardie and Y. Bertot. Formalizing convex hull algorithms. In *TPHOLs*, volume 2152 of *LNCS*, pages 346–361. Springer, 2001.

[25] J. Reed and B. Pierce. Distance makes the types grow stronger: A calculus for differential privacy. In *ICFP*, 2010.

[26] R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry*, 18(3):305–363, 1997.