Bridging Boolean and Quantitative Synthesis Using Smoothed Proof Search

Swarat Chaudhuri, Martin Clochard, Armando Solar-Lezama

Bridging Boolean and Quantitative Synthesis Using Smoothed Proof Search [*]

Swarat Chaudhuri
Rice University
swarat@rice.edu

Martin Clochard
ENS Paris &
Université Paris-Sud (LRI & Inria Saclay)
martin.clochard@lri.fr

Armando Solar-Lezama
MIT
asolar@csail.mit.edu

Abstract

We present a new technique for parameter synthesis under boolean and quantitative objectives. The input to the technique is a "sketch" — a program with missing numerical parameters — and a probabilistic assumption about the program's inputs. The goal is to automatically synthesize values for the parameters such that the resulting program satisfies: (1) a boolean specification, which states that the program must meet certain assertions, and (2) a quantitative specification, which assigns a real valued rating to every program and which the synthesizer is expected to optimize.

Our method — called *smoothed proof search* — reduces this task to a sequence of unconstrained smooth optimization problems that are then solved numerically. By iteratively solving these problems, we obtain parameter values that get closer and closer to meeting the boolean specification; at the limit, we obtain values that provably meet the specification. The approximations are computed using a new notion of *smoothing* for program abstractions, where an abstract transformer is approximated by a function that is continuous according to a metric over abstract states.

We present a prototype implementation of our synthesis procedure, and experimental results on two benchmarks from the embedded control domain. The experiments demonstrate the benefits of smoothed proof search over an approach that does not meet the boolean and quantitative synthesis goals simultaneously.

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]: Correctness proofs; F.3.2 [*Semantics of Programming Languages*]: Program analysis; I.2.2 [*Automatic Programming*]: Program synthesis; I.2.3 [*Deduction and Theorem Proving*]: Uncertainty, "fuzzy,", and probabilistic reasoning

Keywords Synthesis; Probabilistic Verification; Probabilistic Programs; Program Smoothing; Abstract Interpretation

1. Introduction

Traditional reasoning tasks in formal methods are *boolean*: we are asked to prove that a program satisfies a set of boolean properties (program verification), or to realize a given logical specification in the form of an implementation (program synthesis). However, in many applications, boolean reasoning alone is not enough — we also need *quantitative reasoning*.

The need for combined boolean and quantitative reasoning is especially prominent in program synthesis. Here, a boolean specification is naturally used to set a "lower bound" on the desirability of the synthesized implementation: "the program should *at least* meet the following safety properties." However, there can be many implementations that meet these properties, and some of them are more desirable than others. Given this, it is appropriate to consider synthesis tasks where the synthesized implementation must not only meet a boolean specification, but also be optimal with respect to a *quantitative objective* [3, 6].

A plausible approach to solve such synthesis problems is to separately handle the boolean and quantitative goals, for example by searching for candidates that are locally optimal with respect to the quantitative criterion and then attempting to verify them against the boolean specification. The disadvantage of this approach is that if the search for a function that is good with respect to the quantitative objective happens without regards to the subsequent verification phase it may take a long time to find a solution that can be verified. This suggests that a procedure for quantitative synthesis should aim to meet its boolean and quantitative objectives *simultaneously*. The benefits of such an approach are corroborated by prior work on (boolean) synthesis [28, 34], which combines synthesis and verification to make the overall process more tractable. In this paper, we offer such a combined verification and synthesis procedure for the problem of synthesizing values for unknown program parameters.

Specifically, we introduce *smoothed proof search* as a new technique to combine quantitative synthesis and verification. The key idea here is to reduce the synthesis problem to a *sequence of unconstrained optimization problems* where the objective function for each problem is a continuous approximation of the boolean and quantitative objectives. As the sequence progresses, the approximate objective gets closer to the original objective, and in the limit the method finds parameters that provably meet the boolean specification. At the same time, the continuity of the approximations lets us optimize them effectively using local numerical techniques, which rely heavily on smoothness assumptions, and do poorly on the discontinuous functions that programs often represent.

[*] The authors are ordered alphabetically. This work was supported by NSF Award #1162076.

POPL '14, January 22–24, 2014, San Diego, CA, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2544-8/14/01…$15.00.
http://dx.doi.org/10.1145/2535838.2535859

We clarify these ideas with a simple example. Suppose our goal is to find the value of the parameter c in the procedure

```
double P (double x) {
  c := ??;
  if (x > c) { y := x + 1; } else { y := 10; }
  return y
}
```

while satisfying the safety invariant $B : (c - 10 \leq y \leq c + 10)$, and a quantitative specification that states that the return value of the procedure should be as low as possible. This is an example of a *verified parameter synthesis* problem [25, 30, 32, 33]. The programmer has provided an implementation with missing numerical parameters —also known as a "sketch" [30]— and the synthesis problem is to find values for these parameters such that the resulting program satisfies the combined Boolean and quantitative specification.

The problem described above is not yet well defined, as there are different ways to interpret the optimality requirement. One interpretation is that the programmer wishes to minimize the *worst-case* behavior of the system; alternatively, Chatterjee et al. [6] have argued that a more natural goal is to optimize the expected value of the output. This is potentially a harder problem, because it requires knowledge of the distribution of its inputs, and it requires an analysis capable of deriving the distribution of the outputs from this input distribution. In this paper, we focus on this probabilistic view of the problem; specifically, we focus on problems where the input, say x, is drawn from a given probability distribution μ_x, and the quantitative specification is that "The *expected* return value of P on input x is minimal." Because the analysis is probabilistic, the boolean assertion can also be generalized to a probabilistic one. This new assertion (call it φ) is "$(c - 10 \leq x \leq c + 10)$ with probability greater than or equal to a certain threshold θ."

Our approach to this problem is a refinement of the following idea. Using existing ideas on probabilistic abstract interpretation [20], we can symbolically represent the input distribution μ_x of x, then compute an approximation $\widetilde{P}_c(\mu_x)$ of the actual distribution of outputs $P_c(x)$ of the program on input x drawn from the distribution μ_x (these outputs depend on c; hence the subscript c in P_c). From $\widetilde{P}(\mu_x, c)$, we can compute a *sound* upper bound p on the probability with which $P_c(x)$ violates the assertion B and a real interval I such that the expectation of $P_c(x)$ is guaranteed to fall within I. Given this mapping from $c \mapsto (p, I)$, our goal is to find a value of c that leads to low values of p as well as I. We can frame this as a single-objective optimization problem where the goal is to find a c that minimizes $\sup(I) + Penalty(p, \theta)$, where $\sup(I)$ is the least upper bound on I, $Penalty(p, \theta)$ is a penalty function that is zero when $p \leq \theta$, and a large positive value (larger than any upper bound on I) when p is larger than θ. Minimizing this function gives us a value of c that is desirable according to the quantitative criterion while satisfying the probabilistic assertion φ.

Numerical search techniques like gradient descent or Nelder-Mead search [22] seem like the natural choice for solving this optimization problem, because even though they do not provide any guarantees on the optimality of the result, they are known to work well in practice when the function to optimize satisfies certain continuity requirements. These techniques cannot be applied directly, though, because the results of abstract interpretation are highly discontinuous. To understand the source of the discontinuities, consider our example program from before. Let us consider an abstract interpretation where, following prior work [20], probability distributions are approximated by structures that are essentially histograms: disjunctions of pairs (E_i, w_i), where each E_i is a set of possible values for a random variable (a bin in the histogram), and w_i is an upper bound on the measure concentrated in E_i (*i.e.* the probability that the value will fall in that bin). Fig. 1 depicts an example abstraction of a continuous distribution by such a structure.

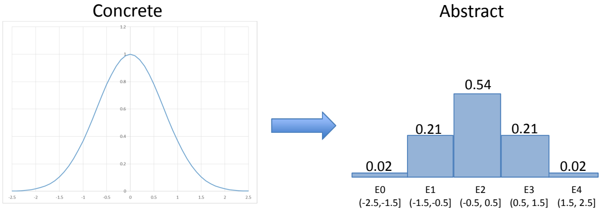


Figure 1. Example of probabilistic abstraction

Now suppose the abstract state of x, right before the conditional "`if (x > c)`" in our example program, is as in Fig. 1. Note that if c is exactly 0.5, the probability that $x > c$ will be bounded from below by 0.23, but for any $\epsilon > 0$, making $c = 0.5 + \epsilon$ will change the lower bound on the probability to 0.02. This is because the abstract state tells us that the probability of falling in the range $(0.5, 1.5]$ is 0.21, but the abstract state has lost the information about the exact probability of falling between 0.5 and $0.5 + \epsilon$. This property of the abstract domain makes the function $c \mapsto (p, I)$ highly discontinuous.

Our approach overcomes this difficulty via a novel notion of *smoothing* for abstract interpretations. Instead of using the abstract transformer $\widetilde{P}_c(\mu_x)$ to generate a target for optimization, we use a series of *approximations* to the abstract transformer that are continuous in the analytical sense. Because of the continuity of these approximations, numerical optimization can be used to compute high-quality minima in objectives generated from them.

We define these approximations by first defining "smooth" analogs of the classic operators of abstract interpretation, such as join and widening. The approximations are neither abstractions or refinements of the sound abstraction $\widetilde{P}_c$ — in fact, they bear no resemblance to operations on abstract interpreters studied in the literature. Notably, each of them is an unsound abstraction. While this may sound like a violation of our stated goals, it is not so. Each of our approximations is parameterized by a real value β; in the limit as β approaches zero, the approximations converge to $\widetilde{P}_c$. Therefore, by iteratively finding local minima on objectives generated from approximations parameterized by lower and lower values of β, we can find parameters that are satisfy our boolean specification with maximal probability. We call the above strategy for meeting a combination of boolean and quantitative synthesis goals *smoothed proof search*.

We have implemented our algorithm in the form of a tool called FERMAT, built on top of the SKETCH program synthesizer. We have used FERMAT to do verified parameter synthesis on two benchmarks from the embedded control domain: a model of a thermostat and a model of an aircraft controller. Our experiments show that smoothing significantly improves the quality of parameters synthesized through search, and that searching simultaneously with respect to the boolean and quantitative objectives gives better results than applying the two kinds of search in sequence.

In summary, this paper makes the following contributions:

- We introduce *smoothed proof search*, a new way to reconcile boolean and quantitative reasoning in program synthesis. The essence of the idea is to reduce a combination of proof and optimization tasks to a sequence of smooth optimization problems.
- We present a concrete smoothed proof search algorithm for verified parameter synthesis in programs with probabilistic inputs. We prove several properties of our algorithm, including

(a) soundness; and (b) the smoothness of the objectives that we generate for numerical optimization.

- We present a prototype implementation of our algorithm, called FERMAT, and perform case studies on two benchmarks from the embedded control domain.

The rest of the paper is organized as follows. In Section 2, we formulate our synthesis problem. In Section 3, we present smoothed proof search as well as a concrete algorithm based on this strategy. Section 4 presents the FERMAT system and experimental results. Section 5 discusses related work; we conclude with some discussion in Section 6. Finally, we had to omit proofs for most of our theorems due to lack of space; these can be found in an online technical report [7].

2. Problem formulation

In this section, we formalize the verified parameter synthesis problem. As outlined earlier, we will be focusing on the probabilistic version of the problem where the goal is to meet the optimality criterion on the average input. The key technical task of this section is the definition of the probabilistic semantics.

Measures To define the semantics of our programs rigorously, we need some definitions from probability theory. For brevity, we only give the most essential of these definitions. A more thorough treatment of this background material can be found in a textbook such as Billingsley's [2].

Definition 1 (Borel sets)**.** A *σ-algebra* σ over $\mathbb{R}^n$ is a set of subsets of $\mathbb{R}^n$ that contains $\emptyset$, and is closed under complementation and countable union. The collection of *Borel sets*, denoted $\mathscr{B}$, is the smallest σ-algebra over $\mathbb{R}^n$ containing the open sets.

Examples of Borel sets include the set $\mathbb{R}^n$, the set of all rational vectors, and the sets of real vectors satisfying conjunctions or disjunctions of polynomial inequalities (in practice, all subsets of the reals of interest in program analysis).

Definition 2 (Measure)**.** A *(finite, nonnegative) measure* μ (over $\mathbb{R}^n$) is a function $\mu : \mathscr{B} \to [0, +\infty)$ such that: (1) $\mu(\emptyset) = 0$, and (2) If $(A_n)_{n\in\mathbb{N}}$ is a countable collection of disjoint subsets of $\mathbb{R}^n$, then $\mu(\bigcup_{n\in\mathbb{N}} A_n) = \sum_{n\in\mathbb{N}} \mu(A_n)$.

The *total weight* of μ is $\|\mu\| = \mu(\mathbb{R}^n)$. If $\|\mu\| = 1$, then μ is a *probability measure*; if $\|\mu\| \leq 1$, then μ is a *subprobability measure*. A measure μ over $\mathbb{R}^n$ is *concentrated* over a subset $Y \subseteq \mathbb{R}^n$ if $\mu(\mathbb{R}^n \setminus Y) = 0$. The *support supp*$(\mu)$ of a measure μ is the least closed set $Y \subseteq \mathbb{R}^n$ such that μ is concentrated on Y.

Intuitively, if μ is a probability measure associated with a random variable x, then $\mu(Y)$ is the probability that $x \in Y$. Non-probability measures μ formalize "unnormalized" probability distributions, where the probability of the "certain" event does not have to be 1.

A function $f : \mathbb{R}^n \to \mathbb{R}^n$ is a *measurable function* if for all Borel sets $Y \subseteq \mathbb{R}^n$, $f^{-1}(Y)$ is also a Borel set. For measurable functions $f : \mathbb{R}^n \to \mathbb{R}^n$ and random variables x with measure μ, we can define the *expectation* $\mathbb{E}_\mu[f(x)]$ of $f(x)$ by the standard *Lebesgue integral*: $\mathbb{E}_\mu[f(x)] = \int f \, d\mu$. All the functions that we consider in this paper are measurable.

Programs Now we define the language (call it IMP) of programs that we want to synthesize. IMP is a core imperative language with standard control constructs. Programs here are allowed to update n memory locations containing real values for a fixed but arbitrary constant n; hence, the state of a program is described by a real vector of length n. Assignments in the program correspond to affine transformations applied to this vector.

Let us fix a single variable x, ranging over $\mathbb{R}^n$, that stores the state of a program. The syntax of Boolean expressions B and programs S in IMP are given by:

$$\begin{array}{rcl} B & ::= & {}^t b_v \cdot x + b_0 > 0 \\ S & ::= & \texttt{skip} \mid x := M \cdot x + c \mid \\ & & \texttt{if}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \mid \\ & & \texttt{while}\ B\ \texttt{do}\ S_1 \mid S_1 ; S_2 \end{array}$$

where $b_v, c \in \mathbb{R}^n$, $b_0 \in \mathbb{R}$, M is a real matrix, and ${}^t b_v$ denotes the transpose of b_v.

We assign a probabilistic concrete semantics $[\![S]\!]$ to each IMP program S (we use $[\![S]\!]_{det}$ whenever we need to refer to the standard non-probabilistic semantics). This semantics is defined as a transformation on finite measures μ (*i.e.* measures with $\|\mu\| < \infty$). Specifically, $[\![S]\!](\mu)$ is the output probability measure of S when applied to an input probability measure μ. The semantics of a boolean expression b is a function $[\![b]\!]$ that maps the state vector to a Boolean, but abusing notation, we also use $[\![b]\!]$ to denote the set of vectors v that satisfy b.

We need some more notation. Given a matrix M and a set of vectors $U \subseteq R^n$, we define $M^{-1}U = \{v | Mv \in U\}$ to be the set of vectors that when multiplied times M produce a vector in U. Note that this operation is well defined even for non-invertible matrices. We also abuse notation by using $(U - a)$ to refer to the set $\{y - a | y \in U\}$. The probabilistic semantics of loop-free programs is now given by the following rules:

$$\begin{array}{rcl} [\![\texttt{skip}]\!](\mu) & = & \mu \\ [\![{}^t b_v \cdot x + b_0 > 0]\!] & = & \{x \in \mathbb{R}^n | {}^t b_v \cdot x + b_0 > 0\} \\ [\![x := Mx + c]\!](\mu) & = & \lambda U.\mu(M^{-1}(U - c)) \\ [\![S_1 ; S_2]\!](\mu) & = & [\![S_2]\!]([\![S_1]\!](\mu)) \\ [\![\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!](\mu) & = & [\![S_1]\!](\lambda U.\mu(U \cap [\![b]\!])) \\ & & +[\![S_2]\!](\lambda U.\mu(U \setminus [\![b]\!])) \end{array}$$

To give semantics to loops, let us define an operation $step_{S,b}(\mu) = [\![S]\!](\lambda U.\mu(U \cap [\![b]\!]))$. We use the notation $step^n_{S,b}$ to refer to the composition of $step_{S,b}$ n times, with $step^0_{S,b}$ being the identity function. Now we define:

$$[\![\texttt{while}\ b\ \texttt{do}\ S]\!](\mu)(U) = \sum_{n=0}^{\infty} step^n_{S,b}(\mu)(U \setminus [\![b]\!]).$$

The quantity $step^n_{S,b}(\mu)(U \setminus [\![b]\!])$ corresponds to the probability that some initial state x will cause the loop to iterate exactly n times and after those n iterations, the resulting state belongs to the set U. For simplicity, in this paper we only consider programs S that terminate on *almost every input* — i.e., the set of inputs on which the program does not terminate is of measure 0.

Example 1. Suppose we are trying to analyze the program from the introduction after having replaced c with 0.5.

```
if (x > 0.5) { y := x + 1; } else { y := 10; }
```

Now, suppose the input x is uniformly distributed over the range $[0, 10)$. The distribution can be represented by a probability measure $\mu(A)$ which for any set A produces the probability that $x \in A$. In particular, for any interval (a, b) with $0 < a < b < 10$,

$$\mu((a, b)) = \frac{b - a}{10}.$$

For any set A that does not intersect $[0, 10)$, $\mu(A) = 0$.

According to the semantic rules, the distribution of the output y is given by the following function:

$$\begin{array}{l} \mu_{out} = \lambda U.\, \mu(M_1^{-1}(U - 1) \cap [0.5, \infty)) + \\ \qquad\qquad \mu(M_2^{-1}(U - 10) \cap (-\infty, 0.5)). \end{array}$$

Now $M_1^{-1}(A) = A$ for any A, and since M_2 is the zero matrix, $M_2^{-1}(A) = \mathbb{R}$ if $0 \in A$, and is the empty set otherwise. Thus, on the range $(1.5, 2)$, we have

$$\begin{aligned}\mu_{out}((1.5,2)) &= \mu((0.5,1.0) \cap [0.5,\infty)) + \mu(\emptyset \cap (-\infty, 0.5)) \\ &= \mu((0.5,1.0)) = 0.05\end{aligned}$$

and on the range $(9, 12)$, we have

$$\begin{aligned}\mu_{out}((9,12)) &= \mu((8,11) \cap [0.5,\infty)) + \mu(\mathbb{R} \cap (-\infty, 0.5)) \\ &= \mu((8,11)) + \mu((-\infty,0.5)) = 0.25\end{aligned}$$

i.e. the probability of the output being between 9 and 12 is 0.25, while the probability of the output being between 1.5 and 2.0 is 0.05.

Lemma 1 (Properties of concrete semantics)**.** The concrete semantics of IMP satisfies the following properties. For all programs S,

$$\begin{aligned}[\![S]\!](\mu_1 + \mu_2) &= [\![S]\!](\mu_1) + [\![S]\!](\mu_2) \\ \|\mu_1\| &= \|[\![S]\!](\mu_1)\|\end{aligned}$$

Moreover, $[\![S]\!](\mu)(U) = \mu([\![S]\!]_{det}^{-1}(U))$, where $[\![\cdot]\!]_{det}$ stands for the usual deterministic semantics of programs.

Sketches A sketch is an IMP program with missing parameters. Formally, let H be a special name for a variable storing an n-tuple of (missing) *control parameters*. We define an *implementation sketch* (or simply a *sketch*) to be a term S_H with the syntax

$$\begin{array}{lll} B & ::= & {}^t b_v \cdot x + {}^t b'_v \cdot H + b_0 > 0 \\ S_H & ::= & \texttt{skip} \mid \\ & & x := M \cdot x + M' \cdot H + c \mid \\ & & \texttt{if } B \texttt{ then } S_{1H} \texttt{ else } S_{2H} \mid \\ & & \texttt{while } B \texttt{ do } S_{1H} \mid S_{1H}; S_{2H} \end{array}$$

where $b_v, c, b'_v \in \mathbb{R}^n$, $b_0 \in \mathbb{R}$, and M, M' are real matrices. By substituting H by a constant vector c in S_H, we obtain an IMP program. We denote this program by $S_H[H \mapsto c]$ or S_c. Note that as the sketch performs only linear operations in H, we can treat the concatenation of H and x as a state vector; the sketch then becomes a standard IMP program $S(x : H)$ on that extended vector.

Assertions A *(probabilistic) assertion* for a program S is a pair $\varphi = (B; \theta)$, where B is a boolean expression in IMP and $0 \leq \theta \leq 1$ is a constant.

Definition 3 (Satisfaction)**.** A measure μ satisfies $\varphi = (B; \theta)$ if $\mu([\![B]\!]) \geq \theta$. A program S satisfies φ on the input μ if $\mu' = [\![S]\!](\mu)$ satisfies φ. Intuitively, if S satisfies φ under μ, then the probability that the assertion B will hold on termination of S is greater than or equal to θ. Also, note that if $\theta = 1$, then φ is a non-probabilistic assertion that is expected to be true for all inputs allowed by the distribution.

For simpler notation, we only allow assertions as postconditions in the formal exposition of our method. However, our method easily extends to requirements asserted at intermediate labels within a program, and such assertions are permitted in our implementation.

Error value Our programs compute an error value, *i.e.* a real value reflecting how close the behavior was to the ideal. In general, we assume that there is an error function $Err(x)$ that computes the error value from the final state of the program. Usually, the error value will just be stored as one of the components of the state vector; for example, if the program is storing the error value in the first component of the state vector, $Err(x) = \delta_0 \cdot x$, where δ_0 is the Kronecker delta.

Parameter synthesis Now we define the parameter synthesis problem that we solve.

Problem 1 (Verified parameter synthesis)**.** Given an implementation sketch P, an input measure μ (assumed to be of bounded support for technical convenience), and a boolean requirement φ, find a vector $c \in \mathbb{R}^n$ such that:

1. **[Boolean goal]** $S_c = S_H[H \mapsto c]$ satisfies φ on input μ.
2. **[Quantitative goal]** The expected error value produced by S_c is minimal. Formally, $c = \mathrm{argmin}_c\, \Phi(c)$, where $\Phi(c) = \mathbb{E}_{[\![S_c]\!](\mu)}[Err(x)]$ is the expected value of component zero of x according to the *output* distribution of S_c.

If we use $S_c(x)$ as a shorthand for $Err([\![S_c]\!]_{det}(x))$, *i.e.* the function that maps an input to its error value, then it is easy to prove that $\mathbb{E}_{[\![S_c]\!](\mu)}[Err(x)] = \mathbb{E}_\mu[S_c(x)]$, so we will use the two notations interchangeably.

Example 2. Let us go back to th e example in the introduction:

```
if (x > c) { y := x + 1; } else { y := 10; }
```

The Boolean goal can be expressed as $\varphi : ((c - 10 \leq y \leq c + 10); 1.0)$. As in Example 1, let us assume that the input x is distributed uniformly within $[0, 10)$.

The probability of passing the assertion at the end of the program is equal to $\mu_{out}([c - 10, c + 10])$, and the expected final value of y is $\mathbb{E}_{\mu_{out}}[y]$ (because our stated goal is to minimize the output y, $Err(y)$ is just the identity function). For this simple program, both of these functions can be computed from the definition of μ_{out}.

Using Lebesgue integration, we see that when $0 \leq c \leq 10$ the expected value as a function of c is:

$$\int y \, d\mu_{out} = 5.05 + (c+1) - \frac{(c+1)^2}{20}$$

So, for example, when $c = 1$, the expected value is 6.85, and when $c = 9$, the expected value is 10.05.

From this, we can see that in order to minimize the output y while preserving the invariant, we need to set $c = 1$. Any value of c lower than 1 will reduce the probability of satisfying the invariant below 1. On the other hand, any value of c in $(1, 10]$ will lead to a higher expected value of y by the above calculation, and if $c > 10$, then the output will be the constant 10.

Example 3 (Thermostat)**.** Let us now understand our problem statement using a "real" example — a controller for a thermostat. The thermostat has two inputs: a *target temperature* l_{target} for the room, and the value l_{in} of the outside temperature. These two inputs are probabilistic, because while we do not know what the outside or the target temperatures will be at a given time, we can collect statistics from the local weather and user's preferences to determine an expected distribution of these input values. The joint distribution μ on l_{in} and l_{target} is assumed to be given.

The output of the program is the difference between the real temperature and the target temperature over a period of time, and the goal is to design a thermostat that will minimize the expected value of this error.

A natural partial implementation for our controller[1] is in Fig. 2. Here the placeholders `??(`c_1`,`c_2`)` stand for missing real-valued parameters; by using a placeholder like the above, the programmer communicates the additional insight that the parameter is likely to lie in the interval $[c_1, c_2]$. The code captures the simple high-level insight that a thermostat is a system with two discrete modes: one where the heater it controls is off and one where the heater is on. When the temperature goes above a certain threshold `tOff`, it is appropriate to turn off the heater; when the room temperature is below a certain `tOn`, the heater must come back on.

[1] This code is written in SKETCH [30]—the language used in our implementation—but is easily translated to IMP.

```
double thermostat(double lin, double ltarget) {
  double h = ??(0,10);
  double tOn = ltarget + ??(-10,0);
  double tOff = ltarget + ??(0,10);
  double isOn = 0.0; double K = 0.1; double curL = lin;
  assert(tOn < tOff; 0.9);  assert(h > 0; 0.9);
  assert(h < 20; 0.9);
  for(int i = 0; i < 40; i = i + 1) {
    if(isOn > 0.5) {
      curL = curL + (h - K * (curL - lin));
      if(curL > tOff) {  isOn = 0.0; }
    } else {
      curL = curL - K * (curL - lin);
      if(curL < tOn) {  isOn = 1.0;  } }
    assert(curL < 120; 0.9); }
  Error = abs(curL - ltarget);
  return Error;
}
```

Figure 2. Sketch of a thermostat. abs is the absolute value function.

The synthesis problem is to find values for tOn and tOff, as well the heat h given off by the heater in each time step, such that the following two conditions are satisfied. First, after 40 steps the temperature of the room should be as close to the target as possible; this is the Error value computed by the sketch. Second, the values of the parameters must be such that if the inputs follow a distribution μ, then the controller *provably* satisfies each probabilistic assertion in the sketch—e.g., the property that the temperature curL should be below 120° with probability > 0.9.

3. Smoothed proof search

In this section, we describe the smoothed proof search approach to verified parameter synthesis. Of the two goals in the statement of the problem, we view the boolean requirement as a minimum requirement on the solutions, and guarantee that every solution satisfies it. However, the quantitative goal is met approximately using local optimization, and substantiated empirically. This choice is akin to how in most program analyses, one guarantees soundness but leaves completeness as an empirical consideration.

Formally, suppose we are given an instance of an optimal synthesis problem $\langle S_H, \varphi, \mu \rangle$, where the components of the tuple have meanings as before. Let $\varphi = (B, \theta)$, and for any $c \in \mathbb{R}^n$, let $S_c = S_H[H \mapsto c]$.

Now, suppose we have a way to compute both the expected error value at the end of the execution as well as the probability that the execution value satisfies B, both as a function of c.

$$\Phi(c) = \mathbb{E}_{[\![S_c]\!](\mu)}[Err(x)] \qquad \Delta(c) = \mathbb{P}[[\![S_c]\!](\mu) \text{ satisfies } B]$$

The goal is to find a c that minimizes $\Phi(c)$ under the constraint that $\Delta(c) < \theta$. We can also frame this as an unconstrained, single-objective optimization problem by introducing a penalty term when $\Delta(c) \geq \theta$. Specifically, such a problem can have the form:

$$\min_c \left(\Phi(c) + Penalty(\Delta(c), \theta)\right).$$

where $Penalty(p, \theta)$ is 0 for $p \leq \theta$ and an arbitrarily large positive value for $p > \theta$.

Our smoothed proof search algorithm addresses two challenges faced in solving the optimization problem above. The first challenge is that computing the true values of Φ and Δ is prohibitively expensive, so we need to have *sound* approximations that can be computed efficiently but can still guarantee that the result does indeed satisfy the probability bound. Second, while local numerical search algorithms — such as Nelder-Mead simplex search — are the only reasonable way of solving the problem, these algorithms rely very strongly on the continuity of the objective function. Due to the presence of braching control constructs, this assumption does not hold in our setting.

Algorithm 1 Smoothed proof search

```
1. Initialize $In_c$ to a random value.
2. Let $\beta_0, \beta_1, \ldots, \beta_m$ be a series of values where $\beta_0$ is a heuristically chosen real constant and $\beta_i = \beta_0 * \kappa^i$ for a constant $0 < \kappa < 1$, and $\beta_m$ is the first $\beta_i$ below a certain threshold $\epsilon_\beta$
3. For $\beta$ in $\beta_0, \beta_1, \ldots, \beta_m$:
   (a) Obtain representations of $\Phi^\#_\beta(c)$ and $\Delta^\#_\beta(c)$. Let $I^\Phi_\beta = \Phi^\#_\beta(c)$ and $I^\Delta_\beta = \Delta^\#_\beta(c)$.
   (b) Set $G := \lambda(c).(\sup(I^\Phi_\beta) + Penalty^\#_\beta(max(I^\Delta_\beta), \theta))$.
   (c) Minimize $G$ using local numerical search, starting from the initial point $In_c$. Let $c^* = \mathrm{argmin}_c\, G(c)$
   (d) Set $In_c := c^*$
4. Verify that $S_{c^*}$ satisfies $\varphi$. If this fact can be proved, then terminate; report $c^*$ as the optimal parameter value. Otherwise, go to step 1.
```

The key idea behind our algorithm is to derive a set of continuous approximations to $\Phi(c)$ and $\Delta(c)$. The approximations $\Phi^\#_\beta(c)$ and $\Delta^\#_\beta(c)$ are continuous but unsound approximations of $\Phi(c)$ and $\Delta(c)$ parameterized by value β that controls the degree of approximation. Specifically, $\Phi^\#_\beta$ maps each control parameter value to an interval I^Φ_β, and $\Delta^\#_\beta$ maps each control parameter value to an interval I^Δ_β. For values of $\beta > 0$, the mappings $\Phi^\#_\beta$ and $\Delta^\#_\beta$ are continuous, and bigger values of β will lead to "smoother" functions that are easier to optimize numerically. In the limit as β goes to zero, on the other hand, the intervals returned by the approximations converge to sound bounds over the original functions Φ and Δ. In other words:

- Let $\lim_{\beta \to 0} \Phi^\#_\beta(c) = I^\Phi_0$. Then $\Phi(c) \in I^\Phi_0$.
- Let $\lim_{\beta \to 0} \Delta^\#_\beta(c) = I^\Delta_0$. Then $\Delta(c) \in I^\Delta_0$.

Thus, bigger values of β make numerical optimization easier, while small values of β make the function closer to the sound but discontinuous approximation we would like to optimize.

Finally, the function $Penalty(p, \theta)$ is also a source of discontinuity, so it is approximated by a smooth function $Penalty^\#_\beta(p, \theta)$. In particular, we use an approximation of the form $Penalty^\#_\beta(p, \theta) = sigmoid(p, \theta, \beta)$, where $sigmoid(p, \theta, \beta)$ is a smooth sigmoid (S-shaped) function that monotonically increases in value with p, has a single inflection point at $p = \theta$, and smoothly approaches the function $Penalty(p, \theta)$ as $\beta \to 0$.

We exploit these properties in our smoothed proof search algorithm (Algorithm 1). Note that, while the abstraction of the error function returns an interval, the algorithm optimizes the *supremum* or least upper bound on this interval (i.e., if $I^\Phi_\beta = (l^\Phi_\beta, h^\Phi_\beta)$, then we optimize h^Φ_β). However, in principle we could have also chosen to optimize a different real objective derived from the interval. On the other hand, it is important that we optimize the upper bound $\sup I^\Delta_\beta$ on the probability of assertion failure, because our goal is to guarantee that the probability of error is below the threshold.

The check in step 4 of our algorithm is important because the bounds computed by approximation G in Step 3 are not sound approximations: the interval $I^\Phi_{\beta_m}$ may not bound the value of $\Phi(c)$, and the interval $I^{\Delta}_{\beta_m}$ may not bound the the value of $\Delta(c)$. However, we can show that as β approaches zero, the approximation converges to sound bounds. Thus, step 4 can be seen to be a "limit" of the iteration in Step 3.

Now we show how to compute $\Phi^{\#}_{\beta}$ and $\Delta^{\#}_{\beta}$ using abstract interpretation. In Section 3.1, we give a method to compute sound bounds for the expected value of a function on an input measure. The procedure that computes sound bounds is discontinuous, so the parameter β does not play a role here. (However, the soundness of the procedure means that we can use it in the verification step — Step 4 — in the algorithm.) Subsequently, in Section 3.2, we present our method for smooth approximation of programs.

3.1 Sound but discontinuous domain

Our abstract interpretation for computing sound (but discontinuous) bounds on the expected output of a program builds on the work of Monniaux [20]. We improve upon this work by leveraging an assumption of structured control flow to handle conditionals more precisely.

Abstract states An abstract state in our domain is a tuple of the form $\mathcal{A} = (I, \{w_i\}, \{p_i\}, \{E_i\})$. Here I is a set of indices; for each index $i \in I$ there is exactly one weight w_i, one fraction p_i and one subset E_i of $\mathbb{R}^n$.

The meaning of each of these components is best understood by defining the concretization function γ, which maps abstract states to sets of sub-probability measures over $\mathbb{R}^n$:

$$\begin{aligned}&\gamma((I, \{w_i\}, \{p_i\}, \{E_i\})) = \\ &\quad \{\mu \mid \mu = \textstyle\sum_{i \in I} \mu_i \wedge \forall i \in I: \ supp(\mu_i) \subseteq E_i \wedge ||\mu_i|| \leq w_i \\ &\qquad \wedge p_i = 0 \Rightarrow \mu_i = 0 \wedge p_i = 1 \Rightarrow ||\mu_i|| = w_i\}\end{aligned}$$

In other words, the measure can be decomposed into a sum of component measures μ_i, where each of these components is concentrated in the set E_i. The values p_i carry information about abstraction precision and are one of the distinguishing features of our abstract domain compared to that of Monniaux. Each p_i lies between 0 and 1; when p_i is 0 or 1, we know that the total weight of the corresponding component μ_i is respectively 0 and the full weight (w_i). An intermediate value means that we don't know the precise value of $||\mu_i||$ but we have a bound on it.

As we saw in the introduction, when $n = 1$ and each E_i is an interval, the abstract state is essentially a histogram; each E_i corresponds to a bucket in the histogram, and the value w_i bounds the total area of the i^{th} bar in the histogram—or if $p = 1$, w_i gives the exact area of that bar. The abstract domain allows us to symbolically propagate this "histogram" through the program. As this histogram passes through branches, we lose certainty about the weight in any given bucket E_i. To see why, suppose the branch condition b has an intersection with E_i. Given that we do not know how the measure on E_i is distributed, we do not know if the weight on $b \cap E_i$ is the total initial weight on E_i.

However, note that by the time the abstract interpretation reaches the end of the branch, this lost certainty is once again regained. This is because, by this point, all the paths into which E_i could have been split within the branch-statement have been joined. Thus, we know that the total weight on E_i at this point is the same as the initial weight on E_i.

Formally, the partial order $\sqsubseteq$ over the abstract states is defined as follows:

$$\begin{aligned}(I, \{w_i\}\{p_i\}, \{E_i\}) \sqsubseteq (I, \{w_i\}, \{q_i\}, \{F_i\}) \quad &\Leftrightarrow \\ \forall i \in I: \ p_i \sqsubseteq_p q_i \wedge E_i \subseteq F_i&\end{aligned}$$

where the relation $\sqsubseteq_p$ is defined as

$$a \sqsubseteq_p b = \begin{cases} true & \text{if } a = b \\ true & \text{if } b \notin \{0, 1\} \\ false & \text{otherwise.} \end{cases}$$

It is important to note a few properties about the partial order. First, note that the partial order is only defined when the abstract states share the same sets I and w_i. In principle, it is possible to provide a more general partial order that also relates abstract states with different index sets and weights, but for the purpose of our analysis, this partial order will suffice. The second point to note is the rationale behind the definition of $\sqsubseteq_p$. The main idea is that when $p_i \in \{0, 1\}$, it restricts the set of measures in the concretization, but if p_i has a fractional value, it is inconsequential.

Example 4. Consider abstract states A and B below:

$$\text{A:} \quad \begin{aligned} I^A &= \{0\} \\ w^A_0 &= 0.5 \\ p^A_0 &= 1/2 \\ E^A_0 &= [0, 2.5] \end{aligned} \qquad\qquad \text{B:} \quad \begin{aligned} I^B &= \{0\} \\ w^B_0 &= 0.5 \\ p^B_0 &= 1/4 \\ E^B_0 &= [0, 2.5] \end{aligned}$$

In this case, both p^A_0 and p^B_0 are fractional, so the concretization of both abstract states is the same and hence $A \sqsubseteq_p B$ even though they have different values of p_i. If p^A_0 were equal to 1, the concretization of A would be a subset of the concretization of B, so $A \sqsubseteq_p B$ would still hold. However, if p^B_0 was to equal 1 or 0, the partial order would only hold if p^A_0 was also made equal to p^B_0.

Abstraction of loop-free programs The abstract semantics of loop-free programs is given by the following rules:

- $$\begin{aligned}&[\![x := Mx + c]\!](I, \{w_i\}, \{p_i\}, \{E_i\}) \\ &\qquad = (I, \{w_i\}, \{p_i\}, \{ME_i + c\})\end{aligned}$$
- $$\begin{aligned}&[\![S_1; S_2]\!](I, \{w_i\}, \{p_i\}, \{E_i\}) \\ &\qquad = [\![S_2]\!]([\![S_1]\!](I, \{w_i\}, \{p_i\}, \{E_i\}).\end{aligned}$$
- $$\begin{aligned}&[\![\texttt{if } b \texttt{ then } S_1 \texttt{else } S_2]\!](I, \{w_i\}, \{p_i\}, \{E_i\}) \\ &\quad = [\![S_1]\!](I, \{w_i\}, \{p^b_i\}, \{E^b_i\}) \sqcup \\ &\qquad\qquad [\![S_2]\!](I, \{w_i\}, \{p^{\neg b}_i\}, \{E^{\neg b}_i\}).\end{aligned}$$

Here

$$p^b_i = \begin{cases} 0 & \text{if } [\![b]\!] \cap E_i = \emptyset \\ p_i & \text{if } E_i \subseteq [\![b]\!] \\ p_i/2 & \text{otherwise} \end{cases}$$

and $p^{\neg b}_i$ is defined in an analogous way. The set E^b_i is some superset of $E_i \cap [\![b]\!]$. The operator $\sqcup$ on abstract states is defined as follows:

$$\begin{aligned}&(I, \{w_i\}, \{p^a_i\}, \{E^a_i\}) \sqcup (I, \{w_i\}, \{p^b_i\}, \{E^b_i\}) = \\ &\qquad (I, \{w_i\}, \{p^a_i + p^b_i\}, \{join(p^a_i, E^a_i, p^b_i, E^b_i)\})\end{aligned}$$

where

$$\begin{aligned}&join(p^a_i, E^a_i, p^b_i, E^b_i) = \\ &\qquad \{x \mid (x \in E^a_i \wedge p^a_i > 0) \vee (x \in E^b_i \wedge p^b_i > 0)\}.\end{aligned}$$

Finally, the notation $ME_i + c$ is used to describe the set of points $\{Mx + c \mid x \in E_i\}$.

Note how the propagation of abstract states through a branch (say with condition b) tracks precision information. If $[\![b]\!] \cap E_i$ is either empty or equal to E_i, we have not lost any precision by propagation through the branch condition. However, if these conditions not hold, then we have lost precision, and this is tracked by shrinking p_i by half. Note also that the $\sqcup$ operator ensures that we will regain this lost precision at the end of the branch.

We observe that $\sqcup$ is not actually a join operation in the traditional sense, because $p^b_i + p^{\neg b}_i = p_i$ and we could have $p_i \sqsubseteq_p p^b_i$.

We refer to this operation as a *subjoin* because it produces a more precise result than the join. However, the result is still sound because of the relationship between the distributions from the two branches as we prove in the appendix. The use of a sub-join operation and the use of the values p_i to help track the precision of the approximation are the main distinguishing features between our method and prior work on probabilistic abstract interpretation by Monniaux [20].

We also observe that $\sqcup$ is defined only when for all i, $p_i^a + p_i^b \leq 1$. It so happens that at any step where the $\sqcup$ operation is applied, this condition holds. Specifically, $\sqcup$ is used only on abstract states computed from two conditional branches. As abstract interpretation of assignments preserves the values of $\{p_i\}$, the $\{p_i\}$ resulting from both branches will sum to the ones before the branch, which was already lower than 1. The rationale behind the definition of the subjoin is illustrated by the following example.

Example 5. Consider a simple code fragment:

```
if (x >= 0.5) { x = x + 10; }
```

Now, suppose the abstract state before the `if` statement is

$$(I = \{0\}, w_0 = 1.0, p_0 = 1, E_0 = [0, 1])$$

That means that x will fall between 0 and 1, but the abstract state has no information about the exact distribution of x in that range. Now, inside the conditional, the abstract state is

$$(I = \{0\}, w_0 = 1.0, p_0 = 1/2, E_0 = [0.5, 1])$$

i.e. x is now between 0.5 and 1, but we don't know the probability that the branch was taken; all we know is that it is bounded by 1. This uncertainty is reflected by the fractional value of p_0. After the branch, the new abstract state will be:

$$(I = \{0\}, w_0 = 1.0, p_0 = 1, E_0 = [0, 0.5) \cup [10.5, 11])$$

Note that p_0 is again 1, because even though the probability of taking each side of the conditional is unknown, we do know that the probabilities add up to w_0, so after the two branches merge, we know that the probability of $x \in E_0$ is exactly w_0.

While loops. Let us now define the abstract semantics of while-loops. First we define an abstract *step* operation:

$$\mathit{step}((I, w_i, p_i, E_i)) = [\![S]\!](I, w_i, p_i^b, E_i^b).$$

The transformations $(\mathit{step}^n)_{n \in \mathbb{N}}$ generate a sequence of abstract states, $((I, \{w_i\}, \{p_{i,n}\}, \{E_{i,n}\}))$, that are visited in succession in an abstract execution. Now we have:

$$\begin{aligned}&[\![\texttt{while}\ b\ \texttt{do}\ S]\!](I, \{w_i\}, \{p_i\}, \{E_i\}) = \\ &\qquad\qquad (I, \{w_i\}, \{p_i\}, \{\nabla((E_{i,n}^{\neg b})_{n \in \mathbb{N}})\})\end{aligned}$$

where $\nabla((E_{i,n}^{\neg b})_{n \in \mathbb{N}})$ is a superset of $\bigcup_{n \in \mathbb{N}} E_{i,n} \cap [\![\neg b]\!]$. The concrete definition of ∇ relies on a widening policy (see Section 3.2).

We can prove the following properties of the abstract domain. (For proofs, see the technical report version of the paper [7].)

Theorem 1. *Let S be any* IMP *program. For each abstract state $\mathcal{A}$ and for $\mu \in \gamma(\mathcal{A})$, we have $[\![S]\!](\mu) \in \gamma([\![S]\!](\mathcal{A}))$.*

Theorem 2. *Consider an abstract state $(I, \{w_i\}, \{p_i\}, \{E_i\})$ such that for all $i \in I$ we have $p_i = 1$, and μ is a probability measure in the abstract state's concretization. Then $\mathbb{E}_\mu[x_k] \leq \sum_{i \in I} \sup(\delta_k E_i) w_i$, where δ_k is the function mapping x to its k^{th} component x_k, and $\sup(\delta_k E_i)$ is the supremum of the k^{th} component over the set E_i.*

Theorem 3. *Let A be an event, and μ a probability measure in the concretization of an abstract state $(I, \{w_i\}, \{p_i\}, \{E_i\})$. Then $\mathbb{P}_\mu[A] \leq \sum\limits_{i \in IA} w_i$, where $IA = \{i \mid E_i \cap A \neq \emptyset\}$.*

The above theorems lead to a strategy for sound verification of probabilistic assertions $\varphi = (B, \theta)$. Given an input measure μ, let us abstract μ into an abstract state $\mathcal{A}_\mu$, then use abstract interpretation to compute $[\![S]\!](\mathcal{A}_\mu) = (I, \{w_i\}, \{p_i\}, \{E_i\})$. Let us certify the program as satisfying φ if $\sum_{i \in ID} w_i \leq \theta$, where $ID = \{i \mid E_i \cap [\![B]\!] \neq \emptyset\}$. From the above theorems, this strategy is sound — i.e., a program certified as satisfying φ does, in fact, satisfy φ.

This strategy is used to implement Step 4 of the smoothed proof search algorithm (Algorithm 1). By the argument given above, we have:

Theorem 4 (Soundness)**.** *If Algorithm 1 returns a value c^* of the missing parameters, then the implementation S_{c^*} satisfies the assertion φ.*

Ellipsoid Domain: Representing sets of points. The abstract domain described above assumes we have a representation of the sets E_i that can be manipulated efficiently. Our algorithm represents these sets as *ellipsoids*, or n-dimensional generalizations of ellipses. This choice is a matter of pragmatics rather than fundamentals. The reason we use ellipsoids — as opposed to other natural choices like polytopes — is that they offer an attractive tradeoff between efficiency and precision. For instance, with ellipsoids, computing volumes (an essential step for us) is inexpensive, whereas for polytopes, volume computations are #P-hard. We note that efficiency considerations are especially important in our setting, where an abstract interpretation is performed on every query from the top-level numerical search routine, and the total number of calls to the abstract interpreter can be in the hundreds/thousands.

Formally, we represent each set E_i as a collection of N-dimensional open ellipsoids $E_i = \{O_{i,j}\}$. Each ellipsoid is represented by a pair $O = \langle M, t \rangle$ of an invertible matrix and a vector. The pair represents a set of points defined as follows:

$$\langle M, t \rangle = \{x \mid \exists r.\, x = Mr + t \wedge \|r\| \leq 1\}.$$

We also have a special non-ellipsoid element $\top$, standing for $\mathbb{R}^n$.

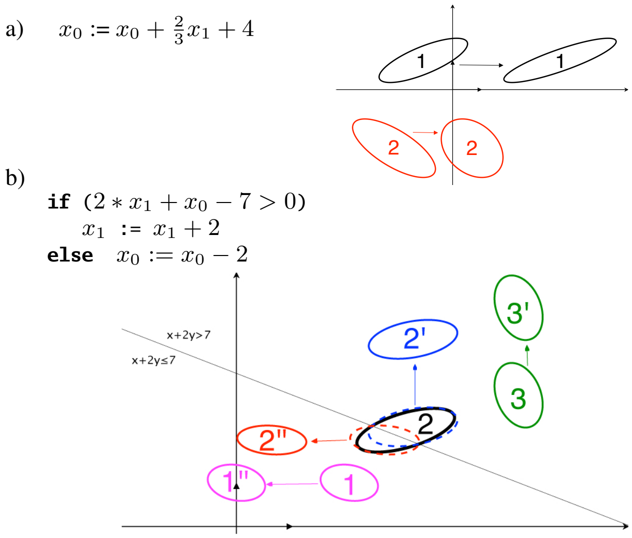
The abstract semantics requires the following operations on the sets E_i: (1) computing the image of set under an affine transformation; (2) checking whether the intersection $E_i \cap [\![b]\!]$ for boolean tests b is empty, and computing the sets E_i^b and $E_i^{\neg b}$, (3) the subjoin operation, and (4) the ∇ operator for loops.

Of these, affine transformations of ellipsoids can be performed exactly in most cases. The one exception is non-invertible assignments, which generate "flat" ellipsoids whose axes along some dimensions equal zero. Our algorithm overapproximates such ellipsoids by ones where each axis is nonzero. For the special case when an ellipsoid equals $\top$, the transformation returns $\top$.

For testing intersections, suppose b equals ${}^t b_v \cdot x + b_o > 0$. Therefore, $[\![b]\!] \cap \langle M, t \rangle$ is nonempty iff $\exists r\ s.t.\ {}^t b_v(Mr + t) + b_o > 0 \wedge \|r\| = 1$. A solution for r in the above equation will exist if and only if we have $\|{}^t M b_v\| + ({}^t b_v \cdot t + b_o) > 0$. By running this check, we can determine if the intersection is empty or not.

As for the sets E_i^b and $E_i^{\neg b}$, we could retain soundness by setting them both to E_i. However, in practice, we achieve higher precision by setting E_i^b and $E_i^{\neg b}$ to the *minimum-volume enclosing ellipsoids* (MVEEs) of the sets $E_i \cap [\![b]\!]$ and $E_i \cap [\![\neg b]\!]$, respectively. We can compute these MVEEs by symbolically transforming E_i — we skip the details.

The subjoin operation is accomplished by simply concatenating the lists of ellipsoids from the two abstract states. If one of the arguments is $\top$, then it returns $\top$. Finally, the ∇ operator for loops is defined via loop unrolling. We define the ∇ operator in such a way that it is equivalent to replacing the loop by the unrolling of its n_0 first iterations, and to return $\top$ in the abstract for the branch corresponding to non-termination in n_0 iterations.



a) $x_0 := x_0 + \frac{2}{3}x_1 + 4$

b)
```
if (2 * x1 + x0 - 7 > 0)
   x1 := x1 + 2
else  x0 := x0 - 2
```

Figure 3. Effect of different statements on ellipsoid domain

Fig. 3 illustrates abstract interpretation using the above domain. Part (a) shows the way an assignment transforms an abstract state with two ellipsoids. Fig. 3-(b) shows the effect of an if-statement. The ellipsoid E_2 that straddles the boundary is split into two parts, with each part translated by the assignment in the corresponding branch. This means that at the join point, the set E_2 now comprises two ellipsoids, while the sets E_1 and E_3 still have one each.

3.2 Continuous approximation

The abstract domain described so far is sound but discontinuous: small changes to a program's inputs could lead to an arbitrarily large change to the expected output. Discontinuities mostly come from ellipsoids appearing and disappearing from the sets E_i. For example, in program (b) on figure Fig. 3, a small change to x_0 that caused ellipsoid 2 to fall entirely below the $x + 2y > 7$ line will cause ellipsoid $2'$ to suddenly vanish from final abstract state. This will result in a discontinuity of $\sup(\delta_0 E_2)$ relative to x_0 and therefore also in the bound computed for $\mathbb{E}[x_0]$. This is the same effect that was illustrated in the introduction with a one dimensional example where the ellipsoids become intervals.

Now we give a domain called *smooth ellipsoids* that provides a continuous, unsound approximation of the above domain. We call the abstract semantics under the smooth ellipsoids domain the *smooth semantics* $[\![\cdot]\!]_\beta$. As stated earlier, the approximation is parameterized by a value β that controls the degree of smoothing. As β approaches zero, the domain converges to the sound domain of Section 3.1.

Compared to the original domain, the smooth ellipsoids domain now associates with each ellipsoid in the set E_i a measure $\alpha_{i,j} \in [0,1]$ that reflects how close each ellipsoid is from disappearing. Specifically, each E_i now corresponds to a multiset $E_i = \{(O_{i,j}, \alpha_{i,j})\}$. The smooth semantics $[\![\cdot]\!]_\beta$ will modify $O_{i,j}$ as before; the $\alpha_{i,j}$ will be initialized to one for all ellipsoids and will be unaffected by assignments but modified by branches. For example, the rule for if-statements will now be as follows:

$$\begin{aligned}&[\![\texttt{if } b \texttt{ then } S_1 \texttt{else } S_2]\!]_\beta(I,\{w_i\},\{p_i\},\{(O_{i,j},\alpha_{i,j})\}) =\\ &\quad [\![S_1]\!]_\beta(I,\{w_i\},\{p_i^b\},\{(O_{i,j},\rho_\beta(O_{i,j},b)*\alpha_{i,j})\})\\ &\qquad \sqcup [\![S_2]\!]_\beta(I,\{w_i\},\{p_i^{\neg b}\},\{(O_{i,j},\rho_\beta(O_{i,j},\neg b)*\alpha_{i,j})\}).\end{aligned}$$

The $\sqcup$ operation at the end of the if-statement will be computed as before, by taking the union of the sets of ellipsoids for each E_i for which p_i is not zero. As for ρ_β, it is a special continuous function. Because the arguments of ρ_β are sets, its continuity needs to be defined with respect to a metric on sets. We choose this metric to be the *Hausdorff metric*, defined below:

Definition 4 (Hausdorff distance)**.** Let A and B be two non-empty subsets of $\mathbb{R}^n$. The Hausdorff distance $D_H(A,B)$ between A and B is defined by

$$D_H(A,B) = \inf_{r\in\mathbb{R}^+}\{A \subseteq B_r \wedge B \subseteq A_r\}$$

where A_r is defined as $\{x|\exists y \in A : ||x-y|| \le r\}$. In other words, the distance between A and B is the smallest r such that if we draw a halo of width r around A, the halo will contain B, and if we draw a halo of width r around B, the halo will contain A.

Formally, the function ρ_β is a function with the following properties:

- ρ_β is continuous with respect to the Hausdorff distance.
- $\rho_\beta(O_{i,j}, b) = \begin{cases} 0 & \text{if } O_{i,j} \cap [\![b]\!] = \emptyset \\ 1 & \text{if } O_{i,j} \subseteq [\![b]\!]. \end{cases}$
- As β approaches zero, ρ_β should smoothly approximate the following function:
$$\rho_0(O_{i,j}, b) = \begin{cases} 0 & \text{if } O_{i,j} \cap [\![b]\!] = \emptyset \\ 1 & \text{otherwise.} \end{cases}$$
- $O_x \cap [\![b]\!] \subseteq O_y \cap [\![b]\!] \Rightarrow \rho_\beta(O_x, b) \le \rho_\beta(O_y, b)$
- For all invertible affine transformations f,
$$\rho_\beta(f(O_x), b \circ f^{-1}) = \rho_\beta(O_x, b).$$
In other words, ρ_β is stable under affine transformation.

(The last requirement is not needed by our proofs, but expresses the fact that we do not want our abstraction to behave differently over programs that are affine transformations of each other.)

The properties imply that if p_x is equal to zero in an abstract state $(I, \{w_i\}, \{p_i\}, \{(O_{i,j}, \alpha_{i,j})\})$, then all the $\alpha_{x,j}$ will be equal to zero as well because the condition under which p_i becomes zero are the same as those under which ρ_β becomes zero.

There are many functions that satisfy the above criteria for ρ_β. A natural choice, which we use in practice, is

$$\rho_\beta(O, b) = \begin{cases} \min(1, \frac{V(O\cap[\![b]\!])}{f(\beta)V(O)}) & \text{if } O \text{ is finite} \\ 1 - f(\beta) & \text{if } O \text{ is the universal set} \end{cases}$$

where $f(\beta) = \min(1/2, \lambda\beta)$ and λ is a constant parameter. Here, $V(S)$ stands for the n-dimensional volume of a subset S of $\mathbb{R}^n$.

Initial distribution A sketch $S_c(x)$ takes two different kinds of inputs: x and c. The initial distribution for inputs x is known and is given as part of the problem definition, but c is unknown. Previous work on smooth interpretation [8] showed that as the search algorithms tries different values c^i for parameter c, it can get better information about the function to be optimized by executing on a distribution centered at c^i. In our context, this means that the initial distribution must be a product between the initial input distribution μ and a distribution $\mu_{(c_i,\beta)}$ centered at c_i and with variance proportional to β. Since we are interested in an abstraction of $\mu \times \mu_{(c_i,\beta)}$, we need a sound product operation over abstract states. In general, this is defined as

$$\begin{aligned}\mathcal{A}_\mu \times \mathcal{A}_{\mu_{(c_i,\beta)}} &= (I, w_i^a, p_i^a, E_i^a) \times (J, w_j^b, p_j^b, E_j^b) = \\ &\qquad (I \times J, w_i^a w_j^b, p_i^a p_j^b, E_i^a \times E_j^b).\end{aligned}$$

However, ellipsoids are not closed under cross product; *e.g.* in the one dimensional case, the product of two ellipsoids $[-a, a]$ and $[-b, b]$ is a rectangle. Instead of overapproximating this rectangle with a bigger ellipsoid, we actually underapproximate it by the largest ellipsoid enclosed by the rectangle. This is unsound in general, but it becomes sound in the limit as β approaches zero.

As for $\mathcal{A}_{\mu_{(c_i,\beta)}}$, for our experiments, we use

$$\mathcal{A}_{\mu_{(c_i,\beta)}} = (\{0\}, \{w_0 = 1\}, \{p_0 = 1\}, \{E_0^{(c_i,\beta)}\})$$

where $E_0^{(c_i,\beta)}$ is the sphere centered at c_i with radius β.

Continuity of $[\![\cdot]\!]_\beta$ Now we establish the continuity of our domain. For space reasons, we only offer a proof sketch of the central theorem behind this property. Full proofs are available in the technical report version of the paper [7].

In order to prove continuity, we first need to define a distance metric $D(\mathcal{A}^a, \mathcal{A}^b)$ between two abstract states $\mathcal{A}^a = (I^a, \{w_i^a\}, \{p_i^a\}, \{E_i^a\})$ and $\mathcal{A}^b = (I^b, \{w_j^b\}, \{p_j^b\}, \{E_i^b\})$, where the sets E are represented as weighted sets of ellipsoids $E_i^x = \{(O_{i,j}^x, \alpha_{i,j}^x)\}$ as discussed earlier.

Definition 5 (Distance)**.** We define the distance to be ∞ between two abstract states with different index sets or different w_i^x. When index sets and w_i^x are identical, the distance $D(\mathcal{A}^a, \mathcal{A}^b)$ between two abstract states is defined by the following equation:

$$\max_i \min_{\sigma_i} \sum_j D((O_{i,j}^a, \alpha_{i,j}^a), (O_{i,\sigma_i(j)}^b, \alpha_{i,\sigma_i(j)}^b))$$

where the σ_i-s are bijective functions. The distance between two weighted ellipsoids is defined in terms of the Hausdorff distance D_H between ellipsoids.

$$D((O^a, \alpha^a), (O^b, \alpha^b)) = \\ \min(D_H(O^a, O^b) + |\alpha^a - \alpha^b|, \max(\alpha^a, \alpha^b)).$$

The definition assumes without loss of generality that matching E_i are represented by the same number of ellipsoids. If this is not the case, we pad with extra ellipsoids with $\alpha = 0$. The choice of the extra ellipsoid does not affect the distance measure because when one of the alphas, say α^a equals zero, the distance $D((O^a, \alpha^a), (O^b, \alpha^b))$ always equals the other alpha α^b irrespective of O^a.

At a high-level, given two abstract states with matching index sets, the definition above computes the distance for each index independently and returns the minimum over all of them. For a given index, the distance function tries to produce the best match between ellipsoids in one state and ellipsoids in the other. For each pair of ellipsoids and their alphas, the distance is dictated by the magnitude of the alphas when the ellipsoids are far apart, and by the distance between the ellipsoids when the ellipsoids are very close together.

Our continuity theorem requires the introduction of an additional technical condition known as *boundedness*.

Definition 6 (Bounded states)**.** Let B be an Euclidean ball; an abstract state is B-bounded if $O_{i,j} \subseteq B$ for all finite ellipsoids $O_{i,j}$ with non-zero measure $\alpha_{i,j}$ that are part of the abstract state. A set of abstract states is bounded if all the abstract states in the set are B-bounded for some Euclidian ball B.

The boundedness condition in the definition of continuity is there because of the previously mentioned property that when distances between ellipsoids are large, the distance metric is dominated by alpha. This means that for any ϵ it is possible to find states that are ϵ-close to $\mathcal{A}^E$ but whose ellipsoids are arbitrarily far and such states could violate the continuity property. Such states, lying outside the B-ball, however, are not relevant from the point of view of smoothed proof search, so this definition of continuity is sufficient for us.

Now we establish that the abstract semantic function $[\![\cdot]\!]_\beta$ is continuous over bounded sets.

Theorem 5. *For any ball B, any ϵ and any B-bounded $\mathcal{A}^E$ there exists a δ such that for all B-bounded $\mathcal{A}^F$,*

$$D(\mathcal{A}^E, \mathcal{A}^F) < \delta \Rightarrow D([\![S]\!]_\beta(\mathcal{A}^E), [\![S]\!]_\beta(\mathcal{A}^F)) < \epsilon.$$

Also, the abstract semantics maps bounded sets to bounded sets.

Proof. (Sketch). We only prove the theorem for loop-free programs. The proof is by induction on the execution of the abstract interpretation. The base case corresponds to assignments; as assignments are linear, the property clearly holds for them.

The interesting inductive case corresponds to conditionals. Let B be any euclidean ball, ϵ any positive real, and $(I, \{w_i\}, \{p_i\}, \{E_i\})$ be any B-bounded abstract state. We have

$$[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!]_\beta(I, \{w_i\}, \{p_i\}, \{E_i\}) = \\ [\![S_1]\!]_\beta(I, \{w_i\}, \{p_i^b\}, \{E_i^b\}) \sqcup \\ [\![S_2]\!]_\beta(I, \{w_i\}, \{p_i^{\neg b}\}, \{E^{\neg b}\})$$

where E_i^b is defined as $\{((O_{i,j}^E)^b, \rho_\beta(O_{i,j}, b) * \alpha_{i,j}^E)\}$ from $E_i = \{(0_{i,j}^E, \alpha_{i,j}^E)\}$.

By the induction hypothesis, $[\![S_1]\!]_\beta$ and $[\![S_2]\!]_\beta$ are continuous maps from bounded sets to bounded sets. So we only need prove this property for $\sqcup$ and the functions $(I, \{w_i\}, \{p_i\}, \{E_i\}) \rightarrow (I, \{w_i\}, \{p_i^{b'}\}, \{E_i^{b'}\})$, where $b' \in \{b, \neg b\}$.

Let us first focus on $\sqcup$. The only possible source of discontinuity comes from the fact that we only merge components with $p_i \neq 0$. However, when $p_i = 0$, every $\alpha_{i,j} = 0$, so distance between union with and without this component is 0. So it is equivalent to componentwise multiset union, which is obviously continuous over the pairs of abstract states. Moreover, if both abstract states are B_1-bounded and B_2-bounded, then resulting abstract states is B-bounded for any ball B enclosing both B_1 and B_2. Since such a ball exists those properties are true for $\sqcup$.

Thus, we only need to prove that the transformation $(I, \{w_i\}, \{p_i\}, \{E_i\}) \mapsto (I, \{w_i\}, \{p_i^b\}, \{E_i^b\})$ maps bounded sets to bounded sets and is continuous over them. The `else` branch case is symmetrical.

The first property directly follows from properties of Minimum Volume Enclosing Ellipsoids (MVEEs). For a full-dimensional convex set S, $\frac{1}{n} MVEE(S) \subseteq S \subseteq MVEE(S)$. Since our resulting ellipsoids are MVEEs of subsets of a common ball, they are included inside this ball scaled by a factor n around its center.

As for the second property, we note that the intersection operation between a convex-set and a half-space is continuous with respect to the Hausdorff distance. Moreover, one can show that the function that computes MVEEs is continuous with respect to Hausdorff distance on full-dimensional sets, so $O_{i,j} \rightarrow O_{i,j}^b$ is continuous on ellipsoids intersecting $[\![b]\!]$.

Now, let us restrict to a distance δ small enough such that any ellipsoid δ-close to some $O_{i,j}^E$ that intersects $[\![b]\!]$, also intersects $[\![b]\!]$. A value δ satisfies this property if for every $O_{i,j} \cap [\![b]\!]$, the furthest point of b hyperplane is at least δ far from it, so such a δ exists. We then restrict this distance to be lower than one so index sets and weights must be identical.

Let $(I, \{w_i\}, \{q_i\}, \{F_i\})$ be any B-bounded abstract state with distance lower than δ from $(I, \{w_i\}, \{p_i\}, \{E_i\})$. By definition of distance, we have

$$D((I, \{w_i\}, \{p_i\}, \{E_i\}), (I, \{w_i\}, \{q_i\}, \{F_i\})) = \\ \max_i \sum_j D((O_{i,j}^E, \alpha_{i,j}^E), (O_{i,j}^F, \alpha_{i,j}^F)).$$

Here we assume, without loss of generality, an indexing for sets E_i and F_i such that minimum over bijective functions are reached at identity maps. Then

$$D((I,\{w_i\},\{p_i^b\},\{E_i^b\}),(I,\{w_i\},\{q_i^b\},\{F_i^b\})) \leq \\ \max_i \textstyle\sum_j D(((O_{i,j}^E)^b, \alpha_{i,j}^E * \rho_\beta(O_{i,j}^E, b)), \\ ((O_{i,j}^F)^b, \alpha_{i,j}^F * \rho_\beta(O_{i,j}^F, b)).$$

In the above, in order to simplify notation, $(O_{i,j}^E)^b$ is defined as any padding ellipsoid when $O_{i,j}^E$ do not intersect $[\![b]\!]$. This is justified by corresponding $\alpha_{i,j}^E * \rho_\beta(O_{i,j}^E, b)$ being zero.

To complete the proof (for the loop-free case), we only need to show how to construct a δ_g such that for any such B-bounded $(I,\{w_i\},\{q_i\},\{F_i\})$,

$$D((I,\{w_i\},\{p_i\},\{E_i\}),(I,\{w_i\},\{q_i\},\{F_i\})) < \delta_g \Rightarrow \\ D((I,\{w_i\},\{p_i^b\},\{E_i^b\}),(I,\{w_i\},\{q_i^b\},\{F_i^b\})) < \epsilon.$$

This construction is somewhat involved and hence we skip it. Interested readers will find the details in the technical report version of the paper [7]. □

Smooth expectation and probability The measures $\alpha_{i,j}$ are used to compute a smooth approximation of the expected value. Recall that a sound upper approximation of the expected error can be computed as follows:

$$\mathbb{E}[x_k] \leq \sum_{i \in I} (\sup(\delta_k E_i)) w_i.$$

In the smooth approximation of this expression, we use the following expression in place of $\sup((\delta_k E_i))$:

$$\mathbb{E}_\beta^\#(k,(I,\{w_i\},\{p_i\},\{E_i\})) = \sum_{i \in I} wsup_\beta(k, \{(O_{i,j}, \alpha_{i,j})\}) w_i$$

The function $wsup_\beta$ is defined as follows. First, let $S_i = [(b_{i,l}, \alpha_{i,\lfloor \frac{l}{2} \rfloor})]$ be a list that contains the supremums and infimums of every $\delta_k O_{i,j}$ with their respective measures; *i.e.*, $b_{i,2*j}$ is the supremum and $b_{i,2*j+1}$ the infimum of coordinate k of ellipsoid $O_{i,j}$ and they are both associated with $\alpha_{i,j}$.

The function $wsup_\beta$ is defined in terms of S_i as follows.

$$wsup_\beta(k, E_i) = \frac{\sum_l \alpha_{i,\lfloor \frac{l}{2} \rfloor} * b_{i,l} * e^{\frac{b_{i,l}}{\beta}}}{\sum_l \alpha_{i,\lfloor \frac{l}{2} \rfloor} * e^{\frac{b_{i,l}}{\beta}}}$$

Note that as β approaches ∞, the function reduces to a weighted average of ellipsoid centers weighted by α_i. In the limit as β approaches zero, on the other hand, the function converges to the supremum. This can be seen by considering the equivalent formula

$$wsup_\beta(k, E_i) = \frac{\sum_l \alpha_{i,\lfloor \frac{l}{2} \rfloor} * b_{i,l} * e^{\frac{b_{i,l} - b_{max}}{\beta}}}{\sum_l \alpha_{i,\lfloor \frac{l}{2} \rfloor} * e^{\frac{b_{i,l} - b_{max}}{\beta}}}$$

where b_{max} is the maximum of the $b_{i,l}$, i.e the actual supremum. When β is close to 0, all the exponential coefficients will converge to zero except for the ones where $b_{i,l} = b_{max}$, so the result will converge to b_{max}.

Note that the universals elements should have special handling here. Instead of using $b_{i,l} * e^{\frac{b_{i,l}}{\beta}}$ factors as for other ellipsoids on the numerator, we use $f(\alpha_{i,l})$, where f is an increasing continuous function with the properties that $f(0) = 0$ and $f(1) = \infty$. On the denominator, $e^{\frac{b_{i,l}}{\beta}}$ are removed for universal elements.

A smooth approximation to an upper bound on the probability of an event, specified as an affine boolean expression b, is defined in a similar way:

$$\mathbb{P}_\beta^\#([\![b]\!]) = \sum_{i \in I} w_i * \sum_j \alpha_{i,j} * \rho_\beta(O_{i,j}, b)$$

From the definition of these bounds and from Theorem 5, we have:

Theorem 6. *For each $\beta > 0$, the expressions $\mathbb{E}_\beta^\#$ and $\mathbb{P}_\beta^\#$ are continuous over bounded sets of abstract states.*

This in turn implies the continuity requirement that we demanded in Section 3:

Theorem 7 (Continuity of $\Phi_\beta^\#$ and $\Delta_\beta^\#$)**.** *Assuming programs are abstracted using smooth ellipsoids and that the abstraction of the initial distribution is B-bounded for some B, the functions $\Phi_\beta^\#(c)$ and $\Delta_\beta^\#(c)$ in Algorithm 1 are continuous.*

Example 6. To illustrate the effect of α, consider the following 1D example where ellipsoids become intervals. Consider two abstract states that are relatively close to each other

$$A_1 = (I = \{0,1\}, \{w_0 = 0.4, w_1 = 0.6\}, \\ \{p_0 = 1.0, p_1 = 1.0\}, \{E_0 = ([0,1], 1.0), E_1 = ([2,3], 1.0)\})$$
$$A_2 = (I = \{0,1\}, \{w_0 = 0.4, w_1 = 0.6\}, \\ \{p_0 = 1.0, p_1 = 1.0\}, \{E_0 = ([0,0.99], 1.0), E_1 = ([2,3], 1.0)\})$$

Now, consider the following code

```
if (x > 0.99) { x = x + 5; }  else { x = x - 5; }
```

In the original discontinuous domain, the state at the end of the conditional would be:

$$[\![P]\!](A_1) = (I = \{0,1\}, \{w_0 = 0.4, w_1 = 0.6\}, \\ \{p_0 = 1.0, p_1 = 1.0\}, \{E_0 = [-5,-4.01] \cup (5.99,6], E_1 = [7,8]\})$$
$$[\![P]\!](A_2) = (I = \{0,1\}, \{w_0 = 0.4, w_1 = 0.6\}, \\ \{p_0 = 1.0, p_1 = 1.0\}, \{E_0 = [-5,-4.01], E_1 = [7,8]\})$$

That means that under A_1, the upper bound on the expected value will be $0.4 * 6 + 0.6 * 8 = 7.2$, whereas the under the slightly different A_2, the upper bound on the expected value is now $0.4 * -4.1 + 0.6 * 8 = 3.16$.

By contrast, under the continuous domain using $\beta = 10$, the abstract states at the end of the program will be:

$$[\![P]\!](A_1) = (I = \{0,1\}, \{w_0 = 0.4, w_1 = 0.6\}, \\ \{p_0 = 1.0, p_1 = 1.0\}, \\ \{E_0 = \{([-5,-4.01], 1), ((5.99,6], 0.01)\}, E_1 = ([7,8], 1.0)\})$$
$$[\![P]\!](A_2) = (I = \{0,1\}, \{w_0 = 0.4, w_1 = 0.6\}, \\ \{p_0 = 1.0, p_1 = 1.0\}, \{E_0 = ([-5,-4.01], 1.0), E_1 = ([7,8], 1.0)\})$$

This means that the smooth upper bound over the expected value for $[\![P]\!](A_1)$ will now be

$$0.4 * \frac{1.0 * -5 * e^{-0.5} + 1.0 * -4.01 * e^{-0.401} + 0.01 * 5.99 * e^{0.599} + 0.01 * 6 * e^{0.6}}{1.0 * e^{-0.5} + 1.0 * e^{-0.401} + 0.01 * e^{0.599} + 0.01 * e^{0.6}} \\ + 0.6 * \frac{7 * e^{0.7} + 8 * e^{0.8}}{e^{0.7} + e^{0.8}} = 2.87$$

and for $[\![P]\!](A_2)$ it will now be

$$0.4 * \frac{-5 * e^{-0.5} + -4.01 * e^{-0.401}}{e^{-0.5} + e^{-0.401}} + 0.6 * \frac{7 * e^{0.7} + 8 * e^{0.8}}{e^{0.7} + e^{0.8}} = 2.76$$

so whereas the original sound approximation was discontinuous, the smoothed approximation changes only slightly when we make a small change to the input distribution. As β decreases, however, the smooth approximation approaches the sound bound, so for example, with $\beta = 1$, the approximate bounds will be 7.02 and 2.92 respectively.

Conservative merges and widening One problem with the domain presented so far is that the representation of each E_i as a set of ellipsoids can potentially double in size after every `if`-statement. The partial order in the discontinuous domain gives us some leeway to define less precise merge functions that prevent some of this blowup. For example, multiple ellipsoids can be replaced by a single one that cover all of them without losing soundness. In the case of the smooth domain, however, such an operation can potentially introduce discontinuity. Our approach described in this section follows the same strategy we followed to achieve continuity in the original semantics. Namely, we introduce a smooth join operation, and show that the abstract semantics using the parameterized join are continuous. Moreover, as β goes to zero (and every α converges to one), the operation converges to a sound join operation.

In order to define join, we define an operation $wcover(\{(O_i, \alpha_i)\})$. The function's input is a non-empty multiset of weighted ellipsoids. Each ellipsoid $O_i = \langle M_i, c_i \rangle$ is represented by a matrix-vector pair as described earlier. The output (O_{out}, α_{out}) is a single weighted ellipsoid with the following properties :

$$\begin{array}{rcl} \alpha_{out} & = & min(\sum_i \alpha_i, 1) \\ \alpha_i = 1 & \Rightarrow & O_i \subseteq O_{out}. \end{array}$$

In order to define $wcover$, we first define a function $wadjust$ which adjust the size and position of each ellipsoid based on its weight. The output $O_i^{'}$ is defined as follows. Let $c_{out} = \frac{\sum_i \alpha_i c_i}{\sum_i \alpha_i}$ be the center of gravity of ellipsoid centers. Then,

$$O_i^{'} = (\alpha_i^{'} * M_i, \alpha_i^{'} * c_i + (1 - \alpha_i^{'}) * c_{out})$$

where $\alpha_i^{'} = \frac{\alpha_i}{\max \alpha_i}$. Note that the uniform center of gravity of resulting center is now c_{out}. The ellipsoid O_{out} produced by $wcover$ is then the Minimum Volume Enclosing Ellipsoid (MVEE) of the ellipsoids $O_i^{'}$. The computation of MVEE is performed using an adaptation of an algorithm by Jambawalikar and Kumar [15].

Fig. 4 illustrates the merge process. Here, consider the ellipsoids labeled 2, $2'$, and $2''$. For ellipsoids $2'$ and $2''$, $\alpha < 1$; therefore, during the merge they are "shrunk" to the small, purple ellipsoids. Ellipsoid 2 does not shrink as $\alpha = 1$ for it. Now we compute the MVEE of 2, $2'$, and $2''$, leading to the dashed red ellipsoid. This is the output of the merge. A similar process applies to ellipsoids 1, $1'$, and $1''$.

The universal elements are handled differently: they are combined in a single universal element, with α_i combined as before.

Now, let us add to our language a statement $ABSTRACT$, whose concrete semantics are equivalent to *skip* and whose abstract semantics is defined as follows:

$$\begin{array}{l} [\![ABSTRACT]\!]((I, \{w_i\}, \{p_i\}, \{E_i\})) = \\ \qquad\qquad (I, \{w_i\}, \{p_i\}, \{wcover(E_i)\}). \end{array}$$

We can prove that $[\![ABSTRACT]\!]$ satisfies the inductive hypothesis of Theorem 5. Furthermore, this is obviously sound on the discontinuous domain.

Theorem 8 (Continuity of conservative merge)**.** $[\![ABSTRACT]\!]$ *maps bounded set of abstract states to bounded sets and is continuous over bounded set of abstract states.*

In that case, Theorem 5 is preserved even if we add the conservative merge operation $ABSTRACT$ to our language. Now we do the same for loops. To get loops to converge, we need a widening strategy. Our strategy is very simple. After a constant number of iterations, we widen our representation to the universal element, then apply $[\![ABSTRACT]\!]$. To be precise, we replace every ellipsoid representation by the universal element before applying $[\![ABSTRACT]\!]$.

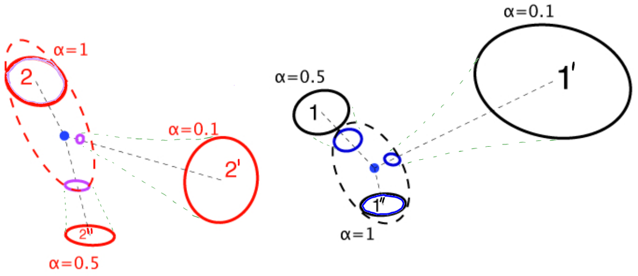


Figure 4. Merge process

Note that this strategy is equivalent to peeling the first k iterations of the loop and then replacing the rest of the loop with the statement $WIDEN$(`while` b `do` S) followed by $ABSTRACT$. This transformation preserves the concrete semantics and is sound for the discontinuous abstract semantics. We show that it does not break the continuity of the smooth semantics by showing that:

Theorem 9 (Continuity of widening)**.** $[\![WIDEN$(`while` b `do` S)$]\!]$ *is continuous over abstract states and maps bounded sets to bounded sets.*

4. Implementation and evaluation

We conducted two case studies using our implementation of FERMAT. The case studies both come from the embedded control domain, and include the thermostat controller introduced in Section 2, and a model of an aircraft collision avoidance maneuver [24].

FERMAT uses the SKETCH program synthesis infrastructure [31] to parse sketches into an AST which is then translated into a C++ program that implements the smooth abstract semantics described earlier. For numerical search, FERMAT uses the Nelder-Mead simplex search [22] available in GNU Scientific Library (GSL) as the underlying local optimization method.

Our two case studies were designed to help answer the following questions:

1. *Does our algorithm really compute better-quality parameters (i.e., parameters that lead to a lower upper bounds on average-case error) than those computed by regular numerical search*? Note that the outcome of any local search routine depends on the starting point of the search, and we cannot expect our algorithm to perform better than the competing approach on *all* starting points. Instead, it is the *distribution* of the expected error value, across starting points, that should interest us. The question is whether this distribution skews to lower expected error values.

2. *In practice, what are the "highest-quality" proofs that the algorithm can find?*

   Consider a probabilistic assertion $(B; 1 - \theta)$, which states that "B is violated with probability $< \theta$". As θ becomes smaller, one would expect the task of finding parameters that provably satisfy this assertion to get harder. The question is to find the lowest value of θ (an "upper bound on the probability of assertion failure") for which different synthesis algorithms can find a parameter meeting this proof goal. Naturally, this bound depends on the start point of the search. The question is whether the distribution of the bounds computed using our algorithm is better than the distribution one would get running a less sophisticated technique.

3. *In our method, a search for optimal parameters is interleaved with a search for a boolean proof. Is this really needed, or could*

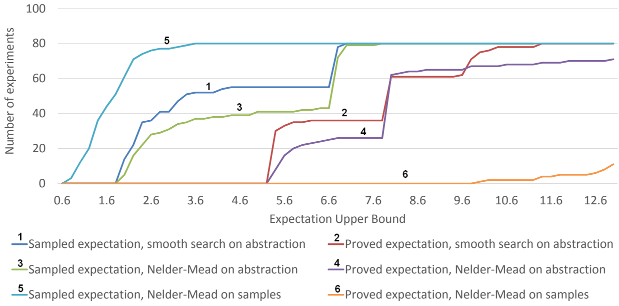


Figure 5. CDF for upper bound on the expected value of the fitness function for the Thermostat experiment.

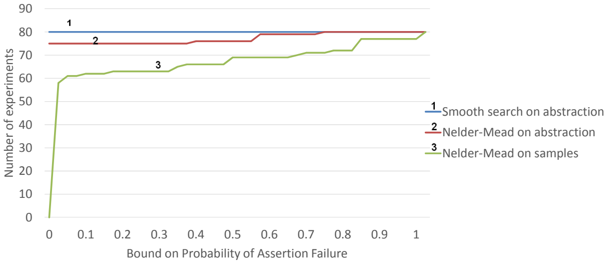


Figure 6. CDF for upper bound on the probability of assertion failure for the Thermostat experiment.

we first find optimal program parameters and then *establish the boolean goal for these parameters?*

For each of the two case studies, we compared three different approaches for parameter synthesis:

- **Smoothed Proof Search** is the algorithm described in this paper, where the synthesizer is using smoothed numerical search to find parameters that allow it to jointly optimize the expected fitness and the probability of assertion failure.
- **Nelder-Mead with Abstract Interpretation** uses a standard Nelder-Mead numerical search to jointly optimize the expected fitness and the probability of assertion failure as computed by the non-smoothed abstract interpretation presented earlier.
- **Nelder-Mead with Sampling** also tries to jointly optimize the expected fitness and the probability of assertion failure, but computes these by running the program on a fixed sample of inputs drawn from the input distribution.

Using each of these approaches, we solved the parameter synthesis problem for the two problems 80 different times from different random starting points in order to get a representative sample of the distribution of solutions that each of these three methods can produce. The results of these experiments are summarized in Figures 5, 6, 7 and 8. The rest of this section describes each of the benchmarks individually and discusses the results.

Thermostat

Recall our thermostat example, where the goal is to shift the temperature of a room from an initial temperature `lin` to a target temperature `ltarget`, and the sketch to be completed is as in Fig. 2. Here, `lin` and `target` are both probabilistic inputs. The distribution of `lin` is trimodal and bell-shaped with modes centered at 30, 35, and 50 and spread ± 3. The distribution of `ltarget` is unimodal and bell-shaped, with mean 75 and spread ± 1. The parameters to be synthesized are the temperatures `tOn` and `tOff` at which the heater respectively switches off and on, and the heat `h` released in a time step. Our safety property states that `tOn` is lower than `tOff` with high probability, and sets limits on the temperature of the room.

Note that the code in the figure permits variables of discrete types. This is syntactic sugar. When such a variable depends on the inputs, we just cast it to a real. Otherwise, the compiler encodes it using control flow—for example, the variable `i` in Fig. 2 is eliminated by unrolling the main loop to a depth of 40.

Fig. 5 shows the cumulative distribution function for the expected value of the fitness function for the thermostat example for each of the three methods above. For each of these methods, we computed the expected value in two ways: a) for the lines labeled "Sampled expectation" we computed the expected fitness by running the synthesized algorithms on a random sample of inputs. And b) for the lines labeled "Proved expectation" we plotted the upper bound of the expected value as computed with abstract interpretation. Fig. 6, shows the probability of assertion failure as proved by abstract interpretation for the solutions generated by the three methods. Together these two graphs allow us to answer the three questions as they apply to the thermostat benchmark.

The results are surprising at first sight; Fig. 5 shows that Nelder-Mead with sampling is able to produce parameters with better average fitness than any of the methods that rely on abstract interpretation, whether smoothed or not. However, while the parameters led to good behavior in practice, the resulting implementations were difficult to analyze, leading to the huge gap between their empirically observed behavior and the probabilities that can be proved for these implementations. This is reinforced by Fig. 6; for the methods where the search used abstract interpretation, the proved probability of assertion failure was close to zero for most instances; by contrast, of the implementations generated by Nelder-Mead there were 60 that had probabilities of failure greater than 2.5%, and a full 10 that had probabilities of failure higher than 50

Of the two methods that use the results of abstract interpretation, we find that smoothed numerical proof was able to produce better results both in terms of their empirically evaluated fitness as well as in the bounds on the expected fitness can prove. The differences are not very big, but they are statistically significant. Using the Wilcoxon signed-rank test, we computed a p-value of 0.027 for the for the expectation upper bound determined empirically and a p-value of less than 0.01 for the proved bounds respectively, supporting the hypothesis that smoothing helped us find better parameters. In the case of the probability of assertion failure, the difference between smooth and non-smooth search was not statistically significant (p-value of 0.25). So in short, we were able to find good bounds for the probability of failure (Question 2), and the use of smoothing had an effect on the fitness of the solutions we got, but not on the probability of assertion failure (Question 1).

Aircraft collision avoidance

In this problem, we want to synthesize parameters for an airplane that performs maneuvers to avoid a collision with a second plane [24]. The controller has four control states called *CRUISE*, *LEFT*, *STRAIGHT*, and *RIGHT*. Normally, our plane flies in the *CRUISE* control state. If it gets within a distance $\delta(\mathtt{x1},\mathtt{y1},\mathtt{x2},\mathtt{y2}) < \mathtt{criticalDist}$ of the other plane, then it starts flying left (in the *LEFT* state) at an angle of 45°, continuing for `delay` number of timesteps. Then it flies *STRAIGHT* for a while. After `delay2` steps it turns *RIGHT*, comes back to its original course, and changes state back to *CRUISE* (see Fig. 10).

Our error value here is the total delay, as in an idealized setting, our airplane returns to its course immediately. Our safety assertions set a minimum distance between the two planes, and ensure that the plane does not start the maneuver too early or too late.

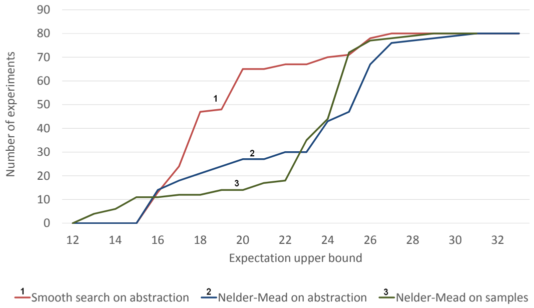


Figure 7. CDF for upper bound on the fitness function for the Aircraft experiment.

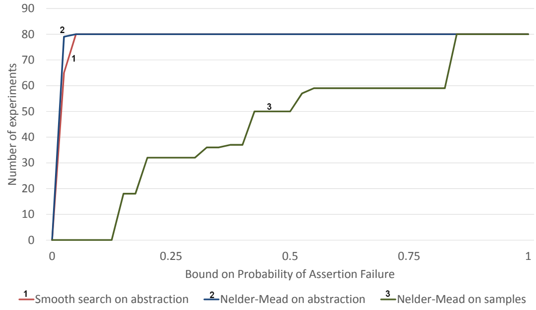


Figure 8. CDF for upper bound on the probability of assertion failure Aircraft experiment.

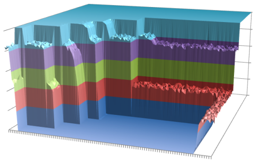


Figure 9. Landscape of safe and unsafe states (lower values imply lower probability of assertion failure)

A sketch of this controller is provided in Fig. 10. Here, the probabilistic inputs are the initial velocities of the two planes. The distribution of the velocity v1 of our plane is unimodal with mean 8 and spread ±0.1; that of the other plane is trimodal with modes centered at 7, 8, and 8.5 and spreads 0.3, 0.4, and 0.2 respectively. Our goal is to synthesize values for the optimal distance from the other plane at which our program's plane should embark on this maneuver, as well as the amount of time that this plane should spend in control states *LEFT*, *RIGHT* and *STRAIGHT*.

The landscape of expected error turns out to be fairly regular in this benchmark. Nonetheless, the presence of assertions leads to a discontinuous search landscape for an optimizer that aims to find "safe" program parameters. Fig. 9 plots the probability of assertion failures at various points in the space of parameter values, as estimated by a sampling of the input space. Note that this probability changes in an highly discontinuous manner.

Because the fitness function is a very simple linear function of the unknown values, so the proved bound is the same as the empirically determined bound for all methods, which is why we only show one line per method in Fig. 7. However, we can see that for this benchmark, smoothing had a big effect on the quality of the parameters we were able to obtain. In terms of the probability of assertion failure, using abstract interpretation as part of the search had a big effect in allowing us to find parameters for which we could prove good bounds for assertion failures, but again, the difference between search with smoothing and without smoothing were not statistically significant.

```
double aircraft(double v1,double v2) {
  ...
  double criticalDist = ??(6,9); double safetyDist = 3.0;
  double delay = ??(10,15); double delay2 = ??(9,14);
  assert(delay > 0.0; θ); assert(delay2 > 0.0; θ);
  assert(criticalDist > safetyDist; θ);
  assert(criticalDist < 10; θ);
  for(int i = 0; i < 50; i = i+1) {
    if (stage == CRUISE) {
      move_straight (x1, y1, x2, y2, v1, v2);
      if(δ(x1, y1, x2, y2) < criticalDist) {
        stage = LEFT; assert(!haveLooped; θ); steps = 0; }
    }
    if(stage == LEFT) {
     move_left (x1, y1, x2, y2, v1, v2); steps = steps + 1;
     if (delay - steps < 0) {
       stage = STRAIGHT; steps = 0; }
    }
    if (stage == STRAIGHT) {
     move_straight (x1, y1, x2, y2, v1, v2); steps = steps + 1;
     if(delay2 - steps < 0) {
       stage = 3; steps = 0; }
    }
    if(stage == RIGHT) {
      move_right (x1, y1, x2, y2, v1, v2);
      steps = steps + 1;
      if(delay - steps < 0) {
        stage = CRUISE;
        haveLooped = 1; }
     }
    assert
    (δ(x1, y1, x2,y2) < safetyDist; θ);
  }
  return(2 * delay + delay2);
}
```

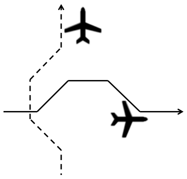


Figure 10. Sketch of collision-avoiding controller

5. Related work

Symbolic smoothing of programs was previously studied by Chaudhuri and Solar-Lezama [8–10]. Unlike the present paper, these prior approaches did not allow any boolean requirement. Also, goal of those papers was to find program parameters that are optimal for a *fixed* set of input values — this question reduces to a function minimization problem of the form $\min_c P(c)$. In contrast, our present approach allows the program inputs to vary, and this leads to a harder optimization problem.

Quantitative synthesis has been studied in the past by the automata-theoretic synthesis community [3–6]. Specifically, the statement for our problem is derived from prior work by Chatterjee et al [6]. However, the solutions provided by these papers are restricted to finite-state systems, whereas our programs and specifications are infinite-state.

Also related is an emerging literature at the interface of machine learning and program analysis. In probabilistic programming [11, 19, 35], the goal is to do probabilistic inference on statistical models written as programs, and synthesis of missing parameters can be viewed as a form of probabilistic inference on programs. However, unlike our paper, prior work on probabilistic programming is not concerned with proving satisfaction of boolean properties. On the other hand, machine learning techniques have also been used recently to prove program correctness [23, 26, 27]. However, the goal there is verification with respect to a boolean objective, as opposed to synthesis with respect to boolean and quantitative goals.

Another thread of related work comes from the embedded software community. The closest match to this paper is Jha et al's work on optimal switching logic synthesis for hybrid systems [17] and synthesis of fixed-point code from floating-point designs [16]. Unlike our work, both these approaches focus on non-stochastic systems, and do not reason about average-case system behavior. Earlier efforts on parameter synthesis in these areas include [13, 14]. However, these papers do not consider stochastic system models and do not have optimality as a goal. While there is an emerging literature on stochastic hybrid systems, researchers here do not appear to have studied the synthesis problem of interest to us.

Smoothed proof search assumes a procedure for sound analysis of probabilistic programs. The particular abstract domain we use for this purpose builds on Monniaux's work on probabilistic abstract interpretation [20, 21]. However, sound analysis of probabilistic systems has also been explored by other researchers [1, 12, 18, 29], and it is plausible that our smoothing technique could be combined with these other abstractions.

6. Conclusion

We have presented a new approach to combined boolean and quantitative reasoning in parameter synthesis. Our method has demonstrated that both reasoning tasks can be accomplished using numerical optimization. The discrete task of finding a proof can be "smoothed" into a continuous optimization task. While this process is unsound, we have offered a way to make it sound in the limit.

Although our approach has used abstract interpretation for proof search, the idea of smoothing can apply to other proof search strategies as well. We are especially interested in applying smoothing to constraint-based approaches to verification and synthesis.

Finally, we are keen on exploring connections between the ideas of this paper and probabilistic programming [19, 35]. Probabilistic programming aims to allow Bayesian inference on models written in a general-purpose programming language; this is accomplished using a combination of machine learning techniques that do the inference, and program analysis techniqes that generate auxiliary information about how probabilities propagate through programs. It is possible that the probabilistic abstraction and symbolic smoothing techniques introduced in this paper can be adapted to this end.

References

[1] A. Adje, O. Bouisseau, J. Goubault-Larrecq, E. Goubault, and S. Putot. Static analysis of programs with imprecise probabilistic inputs. In *VSTTE*, 2013.

[2] Patrick Billingsley. *Probability and measure*. John Wiley & Sons, 2008.

[3] R. Bloem, K. Chatterjee, T. Henzinger, and B. Jobstmann. Better quality in synthesis through quantitative objectives. In *CAV*, pages 140–156, 2009.

[4] P. Cerný, K. Chatterjee, T. Henzinger, A. Radhakrishna, and R. Singh. Quantitative synthesis for concurrent programs. In *CAV*, pages 243–259, 2011.

[5] P. Cerný and T. Henzinger. From boolean to quantitative synthesis. In *EMSOFT*, 2011.

[6] K. Chatterjee, T. Henzinger, B. Jobstmann, and R. Singh. Measuring and synthesizing systems in probabilistic environments. In *CAV*, pages 380–395, 2010.

[7] S. Chaudhuri, M. Clochard, and A. Solar-Lezama. Bridging boolean and quantitative synthesis using smoothed proof search. Technical report, Rice University, 2014.

[8] S. Chaudhuri and A. Solar-Lezama. Smooth interpretation. In *PLDI*, pages 279–291, 2010.

[9] S. Chaudhuri and A. Solar-Lezama. Smoothing a program soundly and robustly. In *CAV*, pages 277–292, 2011.

[10] S. Chaudhuri and A. Solar-Lezama. Euler: A system for numerical optimization of programs. In *CAV*, 2012.

[11] G. Claret, S. Rajamani, A. Nori, A. Gordon, and J. Borgström. Bayesian inference using data flow analysis. In *ESEC/SIGSOFT FSE*, pages 92–102, 2013.

[12] P. Cousot and M. Monerau. Probabilistic abstract interpretation. In *ESOP*, pages 169–193, 2012.

[13] A. Donzé, B. Krogh, and A. Rajhans. Parameter synthesis for hybrid systems with an application to Simulink models. In *HSCC*, 2009.

[14] T. Henzinger and H. Wong-Toi. Using HyTech to synthesize control parameters for a steam boiler. In *Formal Methods for Industrial Applications*, pages 265–282, 1995.

[15] S. Jambawalikar and P. Kumar. A note on approximate minimum volume enclosing ellipsoid of ellipsoids. In *ICCSA*, pages 478–487, 2008.

[16] S. Jha and S. Seshia. Synthesis of optimal fixed-point implementations of numerical software routines. In *NSV*, 2013.

[17] S. Jha, S. Seshia, and A. Tiwari. Synthesis of optimal switching logic for hybrid systems. In *EMSOFT*, pages 107–116, 2011.

[18] J. Katoen, A. McIver, L. Meinicke, and C. Morgan. Linear-invariant generation for probabilistic programs: Automated support for proof-based methods. In *SAS*, pages 390–406, 2010.

[19] A. McCallum, K. Schultz, and S. Singh. Factorie: Probabilistic programming via imperatively defined factor graphs. In *NIPS*, pages 1249–1257, 2009.

[20] D. Monniaux. Abstract interpretation of probabilistic semantics. In *SAS*, pages 322–339, 2000.

[21] D. Monniaux. Backwards abstract interpretation of probabilistic programs. In *ESOP*, 2001.

[22] J.A. Nelder and R. Mead. A simplex method for function minimization. *The computer journal*, 7(4):308, 1965.

[23] A. Nori and R. Sharma. Termination proofs from tests. In *ESEC/SIGSOFT FSE*, pages 246–256, 2013.

[24] A. Platzer. *Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics*. Springer-Verlag, 2010.

[25] S. Seshia. Sciduction: combining induction, deduction, and structure for verification and synthesis. In *DAC*, pages 356–365, 2012.

[26] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, P. Liang, and A. Nori. A data driven approach for algebraic loop invariants. In *ESOP*, pages 574–592, 2013.

[27] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, and A. Nori. Verification as learning geometric concepts. In *SAS*, pages 388–411, 2013.

[28] R. Singh and A. Solar-Lezama. Synthesizing data structure manipulations from storyboards. In *SIGSOFT FSE*, pages 289–299, 2011.

[29] M. Smith. Probabilistic abstract interpretation of imperative programs using truncated normal distributions. *Electron. Notes Theor. Comput. Sci.*, 220(3):43–59, 2008.

[30] A. Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, UC Berkeley, 2008.

[31] Armando Solar-Lezama. Program sketching. *STTT*, 15(5-6):475–495, 2013.

[32] S. Srivastava, S. Gulwani, and J. Foster. From program verification to program synthesis. In *POPL*, pages 313–326, 2010.

[33] M. Vechev and E. Yahav. Deriving linearizable fine-grained concurrent objects. In *PLDI*, pages 125–135, 2008.

[34] M. Vechev, E. Yahav, and G. Yorsh. Abstraction-guided synthesis of synchronization. In *POPL*, pages 327–338, 2010.

[35] J. Winn and T. Minka. Probabilistic programming with infer .NET. *Machine Learning Summer School lecture notes, available at http://research.microsoft.com/˜minka/papers/mlss2009*, 2009.