1 Fundamental Idea

One of the central quantities in the software optimization framework used in the PRINCESS project is the *component model*. Roughly speaking, this model characterizes the conditional probability of program outputs given inputs, $P(Y|X)$. So far, we have considered two alternatives for working with this distribution. The simplest option is to simulate/sample from the procedure under study, f. First, an input sample x is drawn from an input distribution $P(X)$ and the function is evaluated to yield $y = f(x)$. This approach represents a gold standard in accuracy, but has several disadvantages. First, we may not be able or willing to evaluate f during inference, due to computational cost of evaluating f or due to side effects. Second, we may be interested posing probability queries, for instance $P(X|Y = y)$, which are related to f but cannot be efficiently estimated simply by running (forward-simulating) f.

The second alternative for characterizing $P(Y|X)$ we have explored involves generating a sample of X values from $P(X)$, probing f at these points, and fitting a model to the input-output relationship, such as a Gaussian process regression model. This model allows us to *extrapolate* beyond the training data, so we can query the model at points without evaluating the function at those points. We can also more efficiently estimate likely inputs for a given output, $P(X|Y = y)$. One strategy uses a simple application of Bayes rule, $P(X|Y = y) \propto P(Y = y|X)P(X)$, the right hand side of which can be readily evaluated. By contrast, estimating this quantity with simulation from f alone is often impossible when f is a smooth, real-valued function.

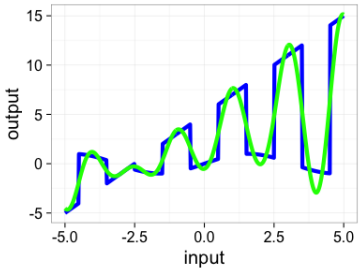
Of course, some degree of accuracy is lost when extrapolating from observed inputs to novel inputs. We can view the model-based approach and the direct simulation approach as two extremes on a spectrum of approximations. f is a golden-standard for accuracy, but can be slow and is not amenable to general probabilistic inference. A model of $P(Y|X)$ can be quite fast, but utilizes no information about the internal structure of the software, and loses accuracy as a result. In between these two extremes, there is a host of techniques which use some information about the internal structure of the software and some probabilistic models.

2 Piecewise Function Example

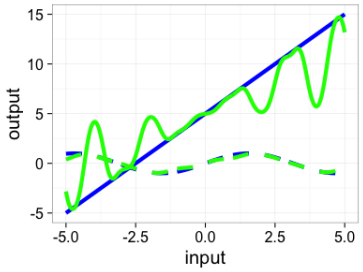
Consider the piecewise function in Figure 1. This example is fairly contrived, but serves as a demonstration of a case where knowledge of the structure of software can be beneficial for probabilistic modeling. For inputs in that round to even integers, the function returns $\sin(in)$. For inputs that round to odd integers, the function returns $2in + 5$. Each of these functions can be

```
public double piecewise(double in) {
    boolean roundsEven =
        Math.round((float) in) % 2 == 0;
    if(roundsEven) {
        return Math.sin(in);
    } else {
        return 2 * in + 5;
    }
}
```

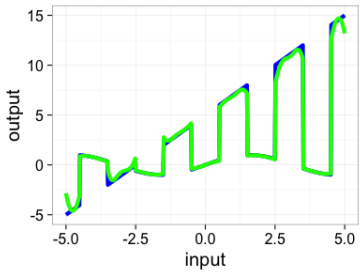
Figure 1: Source code for a piecewise function



(a) Relationship between inputs and outputs of the function given in Figure 1. The true function is shown in blue. The mean of a Gaussian process estimated from 1000 samples is shown in green.



(b) Individual Gaussian process models, in green, independently fit to the sine component (dashed line) and the linear component (solid line).



(c) Individual linear/sine Gaussian process models combined to form the green curve using the rounded even/odd test. In this case, a depth-1 approximation would yield a much more accurate model than the depth-0 approximation.

Figure 2: Illustration of the Hierarchy of Approximations on the Piecewise Function

well-approximated with simple models. However, the piecewise combination of the two yields a challenging non-smooth function. The blue curve in Figure 2b represents the input-output relationship encoded by this function. The green curve is a Gaussian process model of the function, learned from 1000 samples. Even with a relatively large amount of training data, the Gaussian process model is fairly inaccurate, as it fails to model the lack of smoothness in the function.

Our current Figaro translation framework allows us to convert this Java function into an equivalent Figaro model, depicted graphically in Figure 3. This representation is not immediately more beneficial than the original procedure f—it simply represents the program in a different language. However, suppose that the "pieces" of the piecewise function were computationally demanding subroutines. To improve inference performance, we would like to avoid explicitly evaluating these functions at inference time. One strategy would be to approximate these subroutines with a machine learning algorithm by sampling from the distribution of inputs to each subroutine. In the simple example of Figure 1, we can sample some inputs from the distribution $P(in|roundsEven = true)$, evaluate at those inputs, and learn a model which approximates the return value for the true branch. Similarly, we can learn a model which approximates the return value for the false branch. Then, we can replace the portions of the Figaro model that correspond to the exact computation within those branches with the learned models. As a result, we have a model similar to that in Figure 4 which can be more efficient to evaluate than the original program.

This model is also more accurate than a model that approximates the input-output relationship at the highest level (a "depth-0" approximation), like the green curve in Figure 2a. The "pieces" of the piecewise function are simple to approximate independently, as shown in Figure 2b. Then, the Figaro model symbolically combines these pieces to form the much-improved approximation in Figure 2c.

This approximation process can be beneficial even when the original procedure does not have computationally expensive subroutines. Since the unaltered Figaro model of Figure 3 consists of purely deterministic operations applied to a stochastic input, it does not directly admit a sensible value for queries like $P(Y = y|X = x)$. Instead, it behaves more like a simulator—we can generate values of Y given an input x, but we do not have a conditional *density* over Y. Probabilistic regression models such as Gaussian processes provide a means to compute $P(Y = y|X = x)$ and

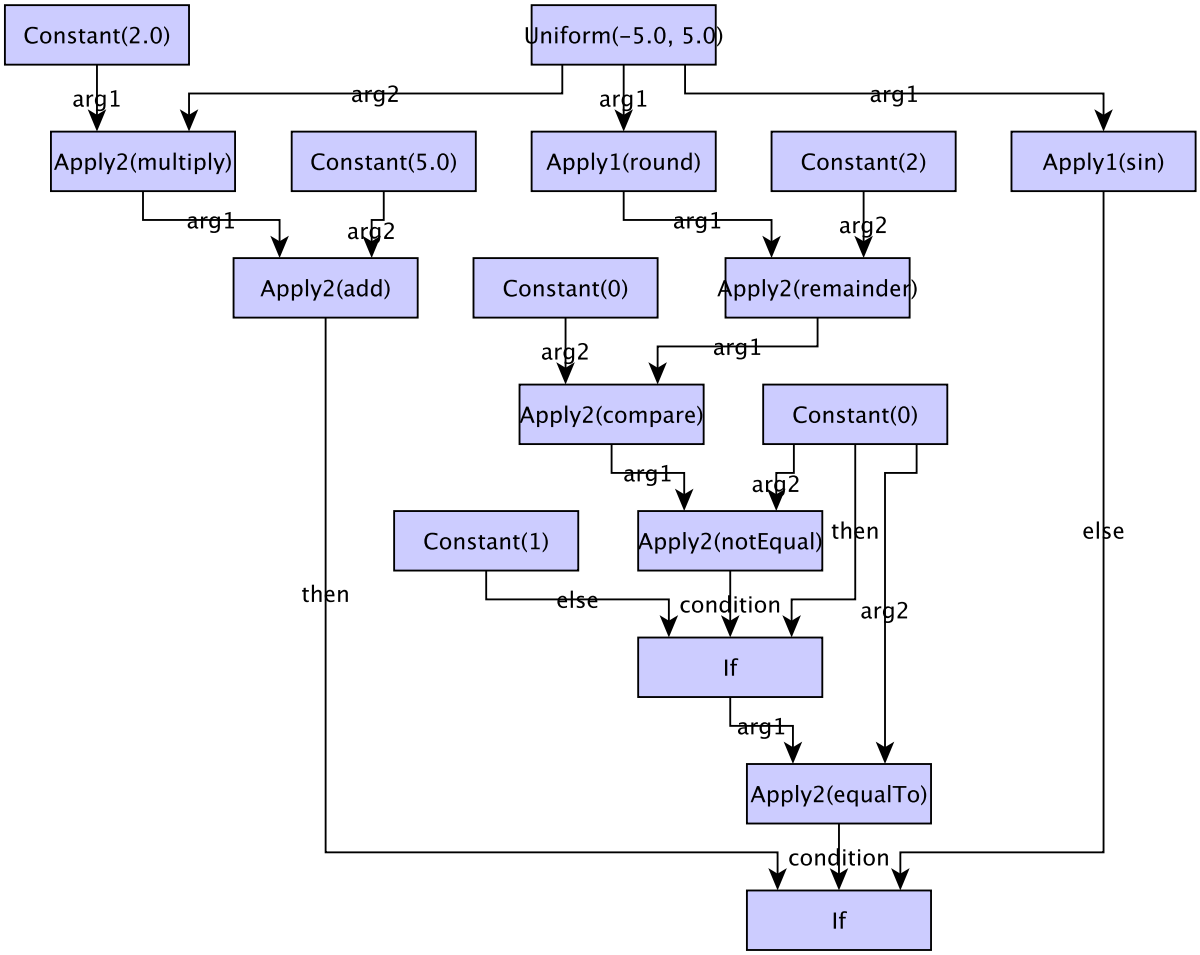


Figure 3: Figaro model representing the function in Figure 1

yield more efficient solutions to a variety of probability queries. For instance, using these Gaussian process models, we can compute:

- $P(Y|X = 3) = \mathcal{N}(-0.87, 0.046)$
- $P(Y|X = 4.8) = \mathcal{N}(14.60, 5.35)$
- $P(X \in [-0.4, 0.4]|Y = 0) \propto 1.06$
- $P(X \in [-0.4, 0.4]|Y = 1) \propto 0.007$
- $P(X \in [-0.4, 0.4]|Y = 5) \propto 0$

3 Loops

Loops present the challenge of possibly infinite program paths. Even if we assume all programs terminate with probability 1, different program executions can instantiate vastly different quantities

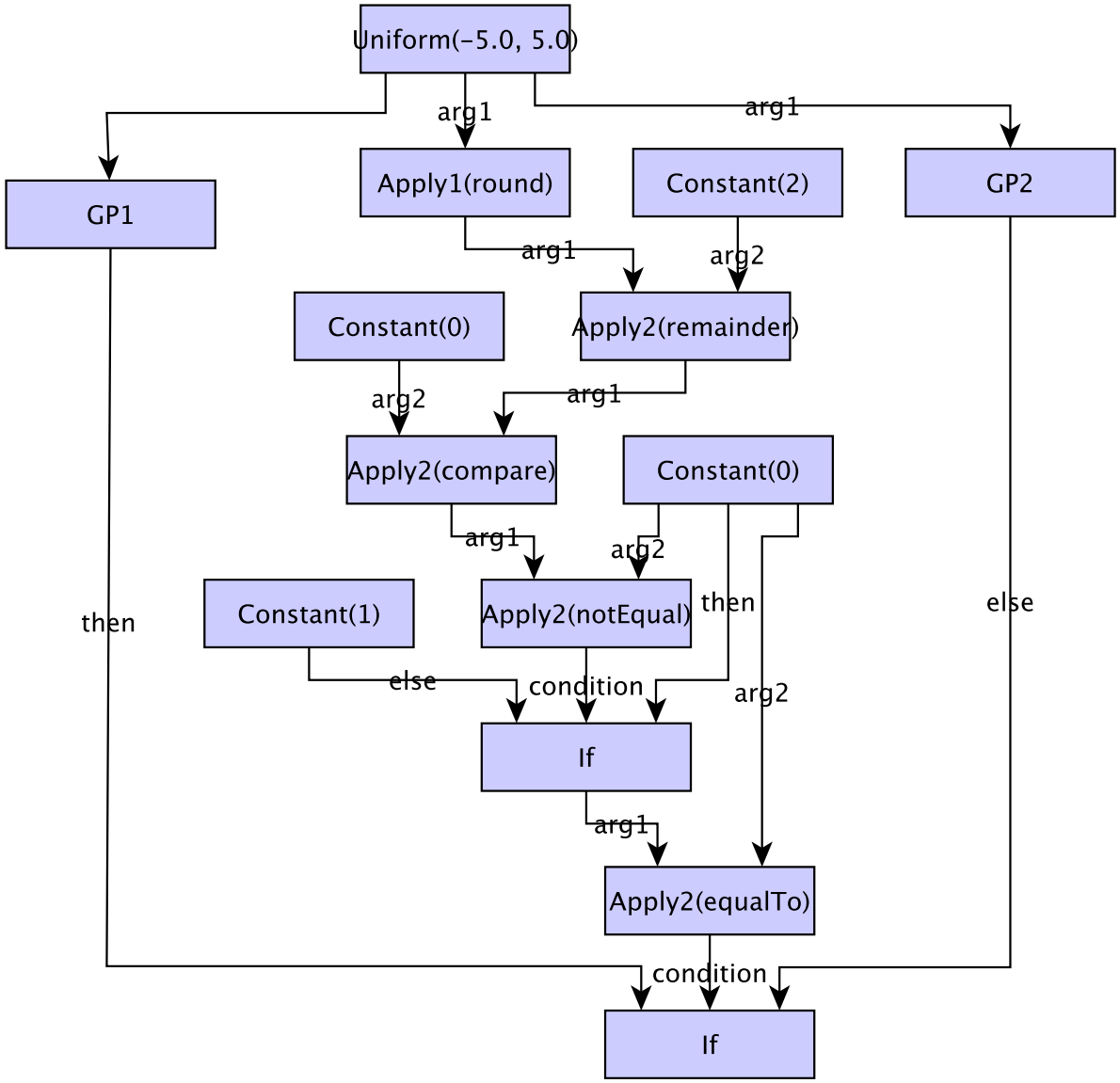


Figure 4: Approximation of the Figaro model in Figure 3. In this case, the *then* and *else* arguments of the final `If` statement are replaced by two Gaussian process models, GP1, and GP2. More commonly, we would be replacing calls to subroutines with Gaussian process models. I have avoided subroutines to maintain the simplicity of the source code.

```
public int loopiters() {
    int count = 0;
    double total = 0;
    while(total < 4) {
        total += Math.random();
        count++;
    }
    return(count);
}
```

(a) Source code

$$P(\text{count} \leq 1) = P(\mathcal{U}(0,1) \geq 4) = 0$$
$$\dots$$
$$P(\text{count} \leq 4) = P(\mathcal{U}(0,4) \geq 4) = 0$$
$$P(\text{count} \leq 5) = P(\mathcal{U}(0,5) \geq 4) = 0.2$$
$$P(\text{count} \leq 6) = P(\mathcal{U}(0,6) \geq 4) = 0.33$$
$$P(\text{count} \leq 7) = P(\mathcal{U}(0,7) \geq 4) = 0.42$$
$$P(\text{count} \leq 8) = P(\mathcal{U}(0,8) \geq 4) = 0.5$$
$$\dots$$
$$P(\text{count} \leq 100) = P(\mathcal{U}(0,100) \geq 4) = 0.96$$

(b) Analysis of the loop exit condition

Figure 5: A Java program with an uncertain loop termination condition

of random variables, depending on the number of iterations taken through loops. To motivate this problem, consider the probabilistic Java program in Figure 5a.

Different executions of this program can sample from a different number of random variables. A statistical model that explicitly models every possible execution of the program would be infinite.

I see three strategies for handling loops.

Strategy 1 Treat loops as black-box functions. A loop reads some variables from its containing scope and writes some variables to that scope upon completion. We can model these input-output relationships with a graphical model, multi-output Gaussian process, or a neural network that admits a conditional density. In the case of the function shown in Figure 5a, the inputs are static, so a marginal density estimator for `count` is sufficient. For instance, we could simulate the loop for 100 iterations and use those samples of `count` to estimate the parameters of a geometric distribution, such as that shown in Figure 6. This model is reasonably accurate, but only because the geometric model is well-suited to this problem. In general, automatically selecting an appropriate model would be challenging. However, one advantage of this approach is that it completely avoids the complexities of reasoning about the control-flow structure of loops.

Strategy 2 Model loops as recursive `Chain`s. Under the lazy structured factored inference (SFI) framework, loops would be unrolled at inference time to a depth that is within a user-specified error tolerance for the query at hand. One such model is shown in Figure 7. By representing loops as recursive function, this approach has the advantage of unifying subroutine processing with loop processing. Further, the lazy SFI techniques used as part of this strategy will soon exist in the Figaro core, and advancements in lazy SFI will be beneficial for translations of this style.

Strategy 3 Unroll loops. At the time of model construction, unroll loops to a pre-specified depth or a depth that covers some sufficient percentage of program executions, and then mark other executions as "uncertain" or use a probabilistic model to summarize the remainder of the executions. For instance, we could estimate the quantities in Figure 5b at the time of model construction, and unroll to a level that guarantees a certain degree of coverage. Suppose we were willing to accept a rough approximation, and unrolled the loop to depth 10, covering 60% of the

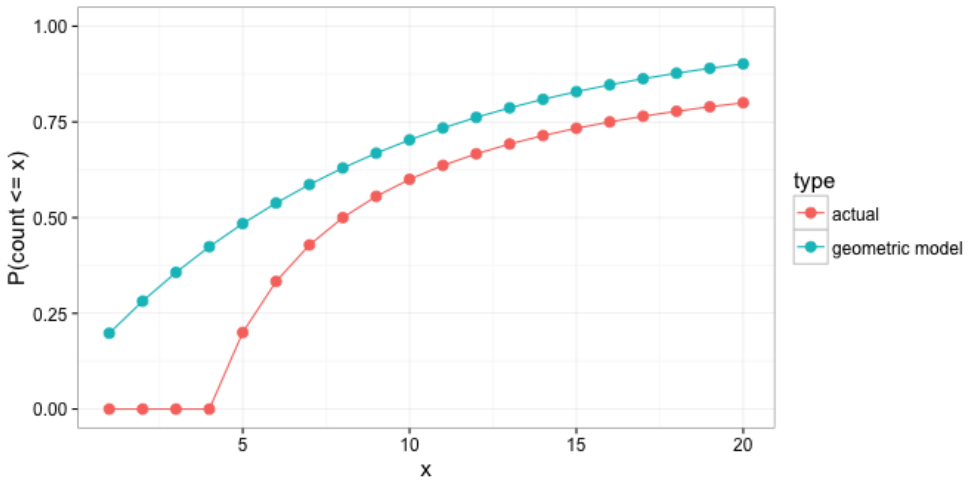


Figure 6: An approximate model of `count`, using a geometric distribution with maximum likelihood estimate $p = 0.1$ obtained by simulating `loopiters` 100 times.

paths through the loop. Then, we fit a model to the uncovered executions. A geometric model can still be applied, but now the model is fit to the distribution of final `count` conditioned on non-termination of the loop for the first 10 iterations. This would produce a Figaro model such as the one in Figure 8. Models of this form make more of the loop structure explicit than the previous two techniques. This may be useful for certain types of approximations, but can result in large programs if the coverage probability does not converge quickly. Even in this simple example, an expansion of 4000 iterations is necessary to ensure 99.9% coverage.

```
def loopFun(count:Int, total:Double):Element[(Int, Double)] = {
  if(total >= 4) {
    ^^(Constant(count), Constant(total))
  } else {
    val newCount = Apply[Int, Int, Int](
        Constant(count), Constant(1), _ + _)
    val thisSample = Uniform(0, 1)
    val newTotal = Apply[Double, Double, Double](
      Constant(total), thisSample, _ + _)
    Chain(newCount, newTotal, loopFun)
  }
}

val loopChain = Chain(Constant(0), Constant(0.0), loopFun)

// infer the expected number of iterations and total
val alg = VariableElimination(loopChain)
alg.run()
println("P(count <= 8): " +
  alg.computeProbability(loopChain, (c:(Int, Double)) => c._1 <= 8))
println("Expected number of iterations: " +
  alg.computeExpectation(loopChain, (c:(Int, Double)) => c._1))
println("Expected total: " +
  alg.computeExpectation(loopChain, (c:(Int, Double)) => c._2))
```

Figure 7: Recursive chain-based Figaro Model for the `loopiters` function. Yields $\hat{P}(\text{count} \leq 8) = 0.6$, an expected number of iterations of about 8.5, and an expected total of about 4.3.

```
val (count10, total10, cond10) = (0 until 10)
    .foldLeft[(Element[Int], Element[Double], Element[Boolean])](
    (Constant(0), Constant(0d), Constant(0d < 4d)))(
    (running, i) => {
    val (count, total, cond) = running

    val curTest = Apply[Double, Double, Boolean](
        total, Constant(4d), _ < _)
    val newCond = Apply[Boolean, Boolean, Boolean](
        cond, curTest, _ && _)

    val newCount = If(newCond, Apply[Int, Int, Int](
        count, Constant(1), _ + _), count)
    val newTotal = If(newCond, Apply[Double, Double, Double](
        total, Uniform(0, 1), _ + _), total)

    (newCount, newTotal, newCond)
  })

val finalCount = If(
    cond10, // are we still in the loop?
    Apply[Int, Int, Int](
        Geometric(0.3886), Constant(10), _ + _), // continuation
           model
    count10 // path covered
    )

val alg = Importance(1000, finalCount)
alg.start()
println(alg.computeDistribution(finalCount).toList)
for(i <- 0 until 20) {
  println("P(count <= " + i + "): " +
    alg.computeProbability(finalCount, (c: Int) => c <= i))
}
println("Expected number of iterations: " +
  alg.computeExpectation(finalCount, (c:Int) => c))
```

Figure 8: Figaro model of `loopiters` constructed by bounding coverage probability and unrolling loops. Yields $\hat{P}(\text{count} \leq 8) = 0.48$, and an expected number of iterations of about 9.