Probability Type Inference for Flexible Approximate Programming

Brett Boston Adrian Sampson Dan Grossman Luis Ceze
University of Washington, USA
{bboston7,asampson,djg,luisceze}@cs.washington.edu

Abstract

In approximate computing, programs gain efficiency by allowing occasional errors. Controlling the probabilistic effects of this approximation remains a key challenge. We propose a new approach where programmers use a type system to communicate high-level constraints on the degree of approximation. A combination of type inference, code specialization, and optional dynamic tracking makes the system expressive and convenient. The *core type system* captures the probability that each operation exhibits an error and bounds the probability that each expression deviates from its correct value. Solver-aided *type inference* lets the programmer specify the correctness probability on only some variables—program outputs, for example—and automatically fills in other types to meet these specifications. An *optional dynamic type* helps cope with complex run-time behavior where static approaches are insufficient. Together, these features interact to yield a high degree of programmer control while offering a strong soundness guarantee.

We use existing approximate-computing benchmarks to show how our language, DECAF, maintains a low annotation burden. Our constraint-based approach can encode hardware details, such as finite degrees of reliability, so we also use DECAF to examine implications for approximate hardware design. We find that multi-level architectures can offer advantages over simpler two-level machines and that solver-aided optimization improves efficiency.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Data types and structures

Keywords approximate computing, type inference

OOPSLA'15, October 25–30, 2015, Pittsburgh, PA, USA
ACM. 978-1-4503-3689-5/15/10...$15.00
http://dx.doi.org/10.1145/2814270.2814301

1. Introduction

Approximate computing seeks to exploit the accuracy–efficiency trade-offs latent in computer systems. Applications exploit unreliable hardware to save time and energy over traditional, fully reliable execution. But while programmers may know which outputs can withstand occasional errors, it is tedious and error-prone to compose individual approximate operations to achieve the desired result. Fine-grained reliability choices can have subtle and far-reaching implications for the efficiency and reliability of a whole computation. Programmers need a way to easily maximize the efficiency of fine-grained operations while controlling the impact of unreliability on overall accuracy properties.

Previous languages for approximate computing have demonstrated that programmers can apply efficient-but-unreliable hardware components using a high-level language [2, 3] and that a type system can ensure that approximation never corrupts essential program state [21]. But safety properties, such as freedom from pointer corruption, are only part of approximate computing's programmability challenge: more nuanced *quality* properties dictate how accurate an output must be or how likely a value is to deviate from its reliable equivalent [2, 23].

We propose DECAF (**D**ECAF, an **E**nergy-aware **C**ompiler to make **A**pproximation **F**lexible), a type-based approach to controlling quality in approximate programs. DECAF's goal is to let programmers specify important quality constraints while leaving the details to the compiler. Its design explores five critical research questions in approximate programming:

How can programmers effectively use complex hardware with many available degrees of approximation? Current languages for approximate programming assume that approximation will be an all-or-nothing affair [2, 16, 21]. But recent work has suggested that more sophisticated architectures, supporting multiple levels of reliability, are a better match for application demands [26]. DECAF is a language abstraction that shields the programmer from reasoning about individual operators to compose reliable software. Its probability type system constrains the likelihood that any expression in the relaxed program differs from its equivalent in a reliable execution.

How can automated tuning interact with programmer control? Compiler assistance can help reduce the annotation burden of approximate programming [10, 16, 20]. But fully automated approaches impede programmers from bringing intuition to bear when fine-grained control is more appropriate. DECAF's solver-aided type inference adds flexibility: programmers add accuracy requirements where they are most crucial and omit them where they can be implied. Programmers in early development phases can opt to rely more heavily on inference, while later-stage optimization work can exert total control over any type in the program.

When static reasoning is insufficient, how can a program safely opt into to dynamic tracking? Purely static systems for reasoning about approximation can be overly conservative when control flow is dynamic [2] while dynamic monitoring incurs run-time overhead [18]. DECAF's optional dynamic typing interoperates with its static system to limit overheads to code where static constraints are insufficient. We prove a soundness theorem that shows that DECAF's hybrid system of static types, dynamic tracking, and run-time checks conservatively bounds the chance of errors.

How should compilers re-use approximate code in contexts with different accuracy demands? An approximate program can invoke a single function in some contexts that permit more approximation and others with stricter reliability requirements. A fixed degree of "aggressiveness" for a function's approximation can therefore be conservative. DECAF's type inference can automatically synthesize specialized versions of approximate functions at multiple levels of reliability.

What do language-level constraints imply for the design of approximate hardware? Approximate hardware designs remain in the research stage. As designs mature, architectures will need to choose approximation parameters that fit a wide range of approximate software. We use DECAF's architecture-aware tuning to examine the implications of programs' language-level constraints on approximate hardware. Our evaluation finds that using a solver to optimize for a hardware configuration can lead to significant efficiency gains over a hardware-oblivious approach to assigning probabilities. We also demonstrate that multi-level architectures can better exploit the efficiency potential in approximate programs than simpler two-level machines, and we suggest a specific range of probability levels that a general-purpose approximate ISA should support.

DECAF consists of a static type system that encodes an expression's probability of correctness, a type inference and code specialization mechanism based on an SMT solver, and an optional dynamic type. We begin with an overview of DECAF and its goals before detailing each component in turn. At the end of the paper, we formalize a core language to prove its soundness, and we report on our implementation and empirical findings.

2. Language Overview

The goal of DECAF is to enforce quality constraints on programs that execute on approximate hardware. Some proposals for approximate hardware, and our focus in this work, provide "relaxed" operations that have a high probability of yielding a correct output but a nonzero chance of producing arbitrarily wrong data [9]. Architectures that allow even a very small probability of error can conserve a large fraction of operation energy [13, 27]. Recently, Venkataramani et al. [26] suggested that hardware with multiple reliability levels—i.e., multiple probabilities of correctness—could provide better efficiency by adapting to the specific demands of approximate software. However, these fine-grained probabilistic operations compose in subtle ways to impact the correctness of coarser-grained outputs.

Consider, for example, a Euclidean distance computation from a clustering algorithm:

```
float distance(float[] v1, float[] v2) {
  float total = 0.0;
  for (int i = 0; i < v1.length; ++i) {
    float d = v1[i] - v2[i];
    total += d * d;
  }
  return sqrt(total);
}
```

This distance function has been shown to be resilient to approximation in clustering algorithms [8]. To manually approximate the function, a programmer would need to select the reliability of each arithmetic operator and determine the overall reliability of the output.

In DECAF, the programmer can instead specify only the reliability of the output—here, the return value. For other values, where the "right" reliability levels are less obvious, the programmer can leave the probability inferred. The programmer decides only which variables may tolerate *some* degree of approximation and which must remain fully reliable. The programmer may write, for example, `@Approx(0.9) float` for the return type to specify that the computed value should have at least a 90% probability of being correct. The intermediate value `d` can be given the unparameterized type `@Approx float` to have its reliability inferred, and the loop induction variable `i` can be left reliable to avoid compromising control flow. The programmer never needs to annotate the operators $-$, $*$, and $+$; these reliabilities are inferred. More simply, the programmer places annotations where she can make sense of them and relies on inference where she cannot. Sections 3 and 4 describe the type system and inference.

DECAF also adapts reused code for different reliability levels. The `sqrt` function in the code above, for example, may be used in several contexts with varying reliability demands. To adapt the `sqrt` function to the reliability contexts in `distance` and other code, DECAF's type inference creates a limited number of *clones* of `sqrt` based on the (possibly inferred) types of the function's arguments and result. The operations in each clone are specialized to provide the opti-

$$
\begin{aligned}
s &::= T\ v := e \mid v := e \mid s\ ;\ s \mid \textbf{if}\ e\ s\ s \mid \textbf{while}\ e\ s \mid \textbf{skip} \\
e &::= c \mid v \mid e \oplus_p e \mid \mathsf{endorse}(p, e) \mid \mathsf{check}(p, e) \mid \mathsf{track}(p, e) \\
\oplus &::= + \mid - \mid \times \mid \div \\
T &::= q\ \tau \\
q &::= \texttt{@Approx}(p) \mid \texttt{@Dyn} \\
\tau &::= \mathsf{int} \mid \mathsf{float} \\
v &\in \text{variables},\ c \in \text{constants},\ p \in [0.0, 1.0]
\end{aligned}
$$

(a) Core language.

$$
\begin{aligned}
e &::= \cdots \mid e \oplus e \mid \mathsf{check}(e) \\
q &::= \cdots \mid \texttt{@Approx}
\end{aligned}
$$

(b) With type inference.

Figure 1: Syntax of the DECAF language. The inferred forms (b) allow omission of the explicit probabilities in the core language (a).

mal efficiency for its quality demands. Section 4.2 describes how DECAF specializes functions.

Finally, DECAF provides optional dynamic tracking to cope with code that is difficult or impossible to analyze statically. In our Euclidean-distance example, the for loop has a data-dependent trip count, so a sound static analysis would need to conservatively assume it executes an unbounded number of times. Multiplying an operator's accuracy probability approaches zero in the limit, so any conservative estimate, as in Rely [2], must assign the total variable the probability 0.0—no guarantees. DECAF's @Dyn type qualifier adds dynamic analysis for these situations. By giving the type @Dyn float to total, the programmer requests limited dynamic reliability tracking—the compiler adds code to the loop to compute an upper bound on the reliability loss at run time. The programmer then requests a dynamic check, and a transition back to static tracking, with an explicit check() cast that throws an exception if the run-time probability falls below an inferred static threshold. Section 5 describes DECAF's dynamic type and run-time checks.

By combining all of these features, one possible approximate implementation of distance in DECAF reads:

```
@Approx(0.9) float distance(float[] v1, float[] v2) {
  @Dyn float total = 0.0;
  for (int i = 0; i < v1.length; ++i) {
    @Approx float d = v1[i] - v2[i];
    total += d * d;
  }
  return sqrt(check(total));
}
```

3. Probability Type System

The core concept in DECAF is an expression's *probability of correctness:* the goal is to specify and control the likelihood that, in any given execution, a value equals the corresponding value in an error-free execution. This section describes DECAF's basic type system, in which each type and operation is explicitly qualified to encode its correctness probability. Later sections add inference, functions and function cloning, and optional dynamic tracking.

Figure 1 depicts the syntax for a simplified version of DECAF. A type qualifier q indicates the probability that an expression is correct: for example, the type @Approx(0.9) int denotes an integer that is correct in least 90% of executions. The basic language also provides approximate operators, denoted $\oplus_p$ where p is the chance that the operation produces a correct answer *given correct inputs*. (We assume that any operator given an incorrect input produces an incorrect output, although this assumption can be conservative—for example, when multiplying an incorrect value by zero.)

The language generalizes EnerJ [21], where types are either completely precise or completely approximate (providing no guarantees). In DECAF, there is no distinct "precise" qualifier—instead, the @Precise annotation is syntactic sugar for @Approx(1.0). EnerJ's @Approx is equivalent to DECAF's @Approx(0.0). In our implementation, as in EnerJ, the precise qualifier—@Approx(1.0)—is the default, so programmers can incrementally annotate reliable code to safely enable approximation.

Information flow and subtyping. For soundness, DECAF's type system permits data flow from high probabilities to lower probabilities but prevents low-to-high flow:

```
@Approx(0.9) int x = ...;
@Approx(0.8) int y = ...;
y = x; // sound
x = y; // error
```

Specifically, we define a subtyping rule so that a type is a subtype of other types with lower probability:

$$\frac{p \geq p'}{\texttt{@Approx}(p)\ \tau \prec \texttt{@Approx}(p')\ \tau}$$

We control implicit flows by enforcing that only fully reliable types, of the form @Approx(1.0) τ, may appear in conditions in if and while statements. (Appendix A gives the full type type system.)

Endorsement expressions provide an unsound escape hatch from DECAF's information flow rules. If an expression e has a type $q\ \tau$, then endorse(0.8, e) has the type @Approx(0.8) τ regardless of the original qualifier q.

Approximate operations. DECAF's operators reflect approximate hardware operations with probabilistic reliability. Specifically, we model hardware where an approximate arithmetic operation produces a perfectly accurate output

most of the time but, with a small but non-negligible probability, can produce arbitrarily wrong output instead. We also assume that these failure events are statistically independent: an error in one operation does not imply anything about the chance of error in another. While this chance-of-failure model does not capture situations where the *magnitude* of error is relevant, it reflects a broad class of current hardware proposals where the probabilty of error is the important parameter: for example, DRAM refresh relaxation [15], probabilistic resistive memories [22], and arithmetic instructions in the Truffle [9] and QUORA [26] architectures.

To implement this model, DECAF provides primitive arithmetic operations parameterized by a correctness probability. For example, the expression $\mathsf{x} +_{0.9} \mathsf{y}$ produces the sum of x and y at least 90% of the time but may return garbage otherwise. The annotation on an operator in DECAF is a lower bound on the correctness probability for the instruction that implements it. For example, if the hardware provides an approximate add instruction with a correctness probability of 0.99, then it suffices to implement $+_{0.9}$ in DECAF. Similarly, a reliable add instruction suffices to implement an approximate addition operator with any probability (although it saves no energy).

The correctness probability for an operation $\mathsf{x} +_{0.9} \mathsf{y}$ is at least the product of the probabilities that x is correct, y is correct, and the addition behaves correctly (i.e., 0.9). To see this, let $\Pr[e]$ denote the probability that the expression e is correct and $\Pr[\oplus_p]$ be the probability that an operator behaves correctly. Then the joint probability for a binary operation's correctness is:

$$\begin{aligned}\Pr[\mathsf{x} \oplus_p \mathsf{y}] &= \Pr[\mathsf{x}, \mathsf{y}, \oplus_p] \\ &= \Pr[\mathsf{x}] \cdot \Pr[\mathsf{y} \mid \mathsf{x}] \cdot \Pr[\oplus_p \mid \mathsf{x}, \mathsf{y}]\end{aligned}$$

The operator's correctness is independent of its inputs, so $\Pr[\oplus_p \mid \mathsf{x}, \mathsf{y}]$ is p. Additionally, operator correctness is independent of all other operators in the program. Therefore, for any operators $\oplus_{p_1}, \oplus_{p_2}, \Pr[\oplus_{p_1} \mid \oplus_{p_2}] = p_1$. The conditional probability $\Pr[\mathsf{y} \mid \mathsf{x}]$ is at least $\Pr[\mathsf{y}]$. This bound is tight when the operands are independent but conservative when they share some provenance, as in $\mathsf{x} + \mathsf{x}$. So we can bound the overall probability:

$$\Pr[\mathsf{x} \oplus_p \mathsf{y}] \geq \Pr[\mathsf{x}] \cdot \Pr[\mathsf{y}] \cdot p$$

DECAF's formal type system captures this reasoning in its rule defining the result type qualifier for operators:

$$\frac{\Gamma \vdash e_1 : \texttt{@Approx}(p_1)\ \tau_1 \qquad \Gamma \vdash e_2 : \texttt{@Approx}(p_2)\ \tau_2 \qquad \tau_3 = \text{optype}(\tau_1, \tau_2) \qquad p' = p_1 \cdot p_2 \cdot p_{\text{op}}}{\Gamma \vdash e_1 \oplus_{p_{\text{op}}} e_2 : \texttt{@Approx}(p')\ \tau_3}$$

where optype defines the unqualified types. Appendix A lists the full set of rules.

This basic type system soundly constrains the correctness probability for every expression. The next two sections describe extensions that improve its expressiveness.

4. Inferring Probability Types

We introduce type inference to address the verbosity of the basic system. Without inference, DECAF requires a reliability level annotation on every variable and every operation in the program. We want to allow the programmer to add reliability annotations only at outputs where requirements are intuitive. In the Euclidean distance example above, we want to uphold a 90% correctness guarantee on the returned value without requiring explicit probabilities on each $+$, $*$, and float. If a programmer wants to experiment with different overall output reliabilities for the distance function, she should not need to manually adjust the individual operators and the sqrt call to meet a new requirement. Instead, the programmer should only express important output correctness requirements while letting the compiler infer the details.

4.1 Inferred Types and Operations

We extend DECAF to make probability annotations optional on both types and operations. The wildcard type qualifier is written @Approx without a parameter. Similarly, $\oplus$ without a probability denotes an inferred operator.

DECAF uses a constraint-based type inference approach to determine operation reliabilities and unspecified types. While constraint-based type inference is nothing new, our type system poses a distinct challenge in that its types are *continuous*. We use an SMT solver to find real-valued type assignments given constraints in the form of inequalities.

As an example, consider a program with three unknown reliabilities: two variables and one operator.

```
@Approx int a, b; ...;
@Approx(0.8) int c = a + b;
```

The program generates a trivial equality constraint for the annotated variable c, a subtyping inequality for the assignment, and a product constraint for the binary operator:

$$p_c = 0.8 \qquad p_c \leq p_{\text{expr}} \qquad p_{\text{expr}} = p_a \cdot p_b \cdot p_{\text{op}}$$

Here, p_{op} denotes the reliability of the addition itself and p_{expr} is the reliability of the expression $\mathsf{a} + \mathsf{b}$. Solving the system yields a valuation for p_{op}, the operator's reliability.

DECAF's constraint systems are typically underconstrained. In our example, the valuation $p_{\text{a}} = p_{\text{b}} = 1$, $p_{\text{op}} = 0.8$ satisfies the system, but other valuations are also possible. We want to find a solution that maximizes energy savings. Energy consumption is a dynamic property, but we can optimize a proxy: specifically, we minimize the total reliability over all operations in the program while respecting the explicitly annotated types. We encode this proxy as an objective function and emit it along with the constraints. We leave other approaches to formulating objective functions, such as profiling or static heuristics, to future work.

DECAF generates the constraints for a program and invokes the Z3 SMT solver [6] to solve them and to minimize the objective function. The compiled binary, including reliability values for each operator, may be run on a hardware simulator to observe energy usage.

4.2 Function Specialization

DECAF's inference system is interprocedural: parameters and return values can have inferred approximate types. In the Euclidean distance code above, for example, the square root function can be declared with wildcard types:

```
@Approx float sqrt(@Approx float arg) { ... }
```

A straightforward approach would infer a single type for sqrt compatible with all of its call sites. But this can be wasteful: if sqrt is invoked both from highly reliable code and from code with looser requirements, a "one-size-fits-all" type assignment for sqrt will be unnecessarily conservative for the more approximate context. Conversely, specializing a version of sqrt for every call site could lead to an exponential explosion in code size.

Instead, we use constraint solving to specialize functions a constant number of times according to calling contexts. The approach resembles traditional procedure cloning [4] but exploits DECAF's SMT formulation to automatically identify the best set of specializations. The programmer enables specialization by giving at least one parameter type or the return type of a function the inferred @Approx qualifier. Each call site to a specializable function can then bind to one of the versions of the callee. The DECAF compiler generates constraints to convey that every call must invoke exactly one specialized version.

For example, in this context for a call to sqrt:

```
@Approx(0.9) float a = ...;
@Approx(0.8) float r = sqrt(a);
```

The compiler generates constraints resembling:

$$p_a = 0.9 \qquad p_r = 0.8 \qquad p_r \leq p_{\text{call}}$$

$$(p_{\text{call}} \leq p_{\text{ret1}} \wedge p_{\text{arg1}} \leq p_a) \vee (p_{\text{call}} \leq p_{\text{ret2}} \wedge p_{\text{arg2}} \leq p_a)$$

Here, p_{ret1} and p_{ret2} denote the reliability of sqrt's return value in each of two versions of the function while p_{arg1} and p_{arg2} denote the argument. This disjunction constrains the invocation to be compatible with at least one of the versions.

The compiler also generates constraint variables—not shown above—that contain the index of the version "selected" for each call site. When inferring types for sqrt itself, the compiler generates copies of the constraints for the body of the function corresponding to each potential specialized version. Each constraint system binds to a different set of variables for the arguments and return value.

DECAF's optimization procedure produces specialization sets that minimize the overall objective function. The compiler generates code for each function version and adjusts each call to invoke the selected version.

Like unbounded function inlining, unbounded specialization can lead to a combinatorial explosion in code size. To avoid this, DECAF constrains each function to at most k versions, a compile-time parameter. It also ensures that all specialized function versions are *live*—bound to at least one call site—to prevent the solver from "optimizing" the program by producing dead function variants and reducing their operator probabilities to zero.

The compiler also detects recursive calls that lead to cyclic dependencies and emits an error. Recursion requires that parameter and return types be specified explicitly.

5. Optional Dynamic Tracking

A static approach to constraining reliability avoids run-time surprises but becomes an obstacle when control flow is unbounded. Case-by-case solutions for specific forms of control flow could address some limitations of static tracking but cannot address all dynamic behavior. Instead, we opt for a general dynamic mechanism.

Inspired by languages with gradual and optional typing [25], we provide optional run-time reliability tracking via a dynamic type. The data-dependent loop in Section 2's Euclidean distance function is one example where dynamic tracking fits. Another important pattern where static approaches fall short is convergent algorithms, such as simulated annealing, that iteratively refine a result:

```
@Approx float result = ...;
while (fitness(result) > epsilon)
  result = refine(result);
```

In this example, the result variable flows into itself. A conservative static approach, such as our type inference, would need to infer the type @Approx(0.0) float for result. Fundamentally, since the loop's trip count is data-dependent, purely static solutions are unlikely to determine an appropriate reliability level for result. Previous work has acknowledged this limitation by abandoning guarantees for any code involved in dynamically bounded loops [2].

To cope with these situations, we add optional dynamic typing via a @Dyn type qualifier. The compiler augments operations involving @Dyn-qualified types with bookkeeping code to compute the probability parameter for each result. Every dynamically tracked value has an associated *dynamic correctness probability field* that is managed transparently by the compiler. This dynamic tracking follows the typing rules analogous to those for static checking. For example, an expression x $+_{0.9}$ y where both operands have type @Dyn float produces a new @Dyn float; at run time, the bookkeeping code computes the dynamic correctness as the product of x's dynamic probability value, y's probability, and the operator's probability, 0.9.

Every dynamic type @Dyn τ is a supertype of its static counterparts @Approx τ and @Approx(p) τ. When a statically typed value flows into a dynamic variable, as in:

```
@Approx(0.9) x = ...;
@Dyn y = x;
```

The compiler initializes the run-time probability field for the variable y with x's static reliability, 0.9.

Flows in the opposite direction—from dynamic to static—require an explicit dynamic cast called a *checked endorse-*

ment. For an expression e of type @Dyn τ, the programmer writes check(p, e) to generate code that checks that the value's dynamic probability is at least p and produce a static type @Approx(p) τ. If the check succeeds, the static type is sound. If it fails, the checked endorsement raises an exception. The program can handle these exceptions to take corrective action or fall back to reliable re-execution.

This dynamic tracking strategy ensures that run-time quality exceptions are predictable. In a program without (unchecked) endorsements, exceptions are raised deterministically: the program either always raises an exception or never raises one for a given input. This is because control flow is fully reliable and the dynamic probability tracking depends only on statically-determined operator probabilities, not the dynamic outcomes of approximate operations.

In our experience, @Dyn is only necessary when an approximate variable forms a loop-carried dependency. Section 8 gives more details on the placement and overhead of the @Dyn qualifier.

Interaction with inference. Like explicitly parameterized types, inferred static types can interact bidirectionally with the @Dyn-qualified types. When a value with an inferred type flows into a dynamic type, as in:

```
@Approx x = ...;
@Dyn y = x;
```

The assignment into y generates no constraints on the type of x; any inferred type can transition to dynamic tracking. (The compiler emits a warning when no other code constrains x, a situation that can also arise in the presence of endorsements. See the next section.)

Inference can also apply when transitioning from dynamic to static tracking with a checked endorsement. DECAF provides a check(e) variant that omits the explicit probability threshold and infers it. This inferred parameter is useful when other constraints apply, as in the last line of the Euclidean distance example above:

```
return sqrt(check(total));
```

The result of the sqrt call needs to meet the programmer's @Approx(0.9) float constraint on the return type, but the correctness probability required on total to satisfy this demand is not obvious—it depends on the implementation of sqrt. The compiler can infer the right check threshold, freeing the programmer from manual tuning.

Operators with @Dyn-typed operations cannot be inferred. Instead, operations on dynamic values are reliable by default; the programmer can explicitly annotate intermediate operations to get approximate operators.

6. Using the Language

This section details two practical considerations in DECAF beyond the core mechanisms of inference, specialization, and dynamic tracking.

Constraint warnings. In any type inference system, programmers can encounter unintended consequences when constraints interact in unexpected ways. To guard against two common categories of mistakes, the DECAF compiler emits warnings when a program's constraint system ether *allows* a probability variable to be 0.0 or *forces* a probability to 1.0. Each situation warrants developer attention.

An inferred probability of 0.0 indicates that a variable is unconstrained—no chain of dependencies connects the value to an explicit annotation. Unconstrained types can indicate dead code, but they can also signal legitimate uses that require additional annotation. If an inferred variable flows only into endorsements and @Dyn variables, and never into explicitly annotated types, it will have no constraints. Without additional annotation, the compiler will use the most aggressive approximation level available. The programmer can add explicit probabilities to constrain these cases.

Conversely, an inferred probability of 1.0—i.e., no approximation at all—can indicate a variable that flows into itself, as in the iterative refinement example in the previous section or the total accumulation variable in the earlier Euclidean distance example. This self-flow pattern also arises when updating a variable as in x = x + 1 where x is an inferred @Approx int. In these latter situations, a simple solution is to introduce a new variable for the updated value (approximating a static single assignment transformation). More complex situations require a @Dyn type.

Hardware profiles. While DECAF's types and inference are formulated using a continuous range of probabilities, realistic approximate hardware is likely to support only a small number of discrete reliability levels [9, 26]. The optimal number of levels remains an open question, so different machines will likely provide different sets of operation probabilities. A straightforward and portable approach to exploiting this hardware is to round each operation's probability up to the nearest hardware-provided level at deployment time. When there is no sufficiently accurate approximation level, a reliable operation can be soundly substituted.

We also implement and evaluate an alternative approach that exploits the hardware profile of the intended deployment platform at compile time. The compiler can use such an *a priori* hardware specification to constrain each variable to one of the available levels. The SMT solver can potentially find a better valuation of operator probabilities than with post-hoc rounding. (This advantage is analogous to integer linear programming, where linear programming relaxation followed by rounding typically yields a suboptimal but more efficient solution.) In our evaluation, we study the effects of finite-level hardware with respect to a continuous ideal and measure the advantage of *a priori* hardware profiles.

7. Formalism

A key feature in DECAF is its conservative quality guarantee. In the absence of unchecked endorsements, a DECAF

program's probability types are *sound:* an expression's static type gives a lower bound on the actual run-time probability that its value is correct. The soundness guarantee applies even to programs that combine static and dynamic tracking. To make this guarantee concrete, we formalize a core of DECAF and prove its soundness.

The formal language represents a version of DECAF where all types have been inferred. Namely, the core language consists of the syntax in Figure 1a. It excludes the inferred expressions and types in Figure 1b but includes approximate operators, dynamic tracking, and endorsements. (While we define the semantics for both kinds of endorsements for completeness, we will prove a property for programs having only *checked* endorsements. Unchecked endorsements are an unsound escape hatch.)

The core language also includes one expression that is unnecessary in the programmer-facing version of DECAF: $\mathsf{track}(p,\ e)$. This expression is a cast from any static type $\texttt{@}\mathsf{Approx}(p')\,\tau$ to its dynamically-tracked equivalent, $\texttt{@}\mathsf{Dyn}\,\tau$. At run time, it initializes the dynamic probability field for the expression. In the full language, the compiler can insert this coercion transparently, as with implicit int-to-float coercion in Java or C.

This section gives an overview of the formalism's type system, operational semantics, and main soundness theorem. Appendix A gives the full details and proofs.

Types. There are two judgments in DECAF's type system: one for expressions, $\Gamma \vdash e : T$, and another for statements, $\Gamma \vdash s : \Gamma'$, which builds up the static context Γ'.

One important rule gives the static type for operators, which multiplies the probabilities for both operands with the operator's probability:

$$\frac{\Gamma \vdash e_1 : \texttt{@}\mathsf{Approx}(p_1)\ \tau_1 \qquad \Gamma \vdash e_2 : \texttt{@}\mathsf{Approx}(p_2)\ \tau_2 \qquad \tau_3 = \text{optype}(\tau_1, \tau_2) \qquad p' = p_1 \cdot p_2 \cdot p_{\text{op}}}{\Gamma \vdash e_1 \oplus_{p_{\text{op}}} e_2 : \texttt{@}\mathsf{Approx}(p')\ \tau_3}$$

Here, optype is a helper judgment defining operators' unqualified types.

Operational semantics. We present DECAF's run-time behavior using operational semantics: small-step for statements and large-step for expression evaluation. Both sets of semantics are nondeterministic: the operators in DECAF can produce either a correct result number, c, or a special error value, denoted $\square$.

To track the probability that a value is correct (that is, not $\square$), the judgments maintain a probability map S for all defined variables. There is a second probability map, D, that reflects the compiler-maintained dynamic probability fields for @Dyn-typed variables. Unlike D, the bookkeeping map S is an artifact for defining our soundness criterion—it does not appear anywhere in our implementation.

The expression judgment $H; D; S; e \Downarrow_p V$ indicates that the expression e evaluates to the value V and is correct with probability p. We also use a second judgment, $H; D; S; e \Downarrow_p V, p_d$, to denote dynamically-tracked expression evaluation, where p_d is the computed shadow probability field. As an example, the rules for variable lookup retrieve the "true" probability from the S map and the dynamically-tracked probability field from D:

$$\textsc{var}\ \frac{v \notin D}{H; D; S; v \Downarrow_{S(v)} H(v)} \qquad \textsc{var-dyn}\ \frac{v \in D}{H; D; S; v \Downarrow_{S(v)} H(v), D(v)}$$

The statement step judgment is $H; D; S; s \longrightarrow H'; D'; S'; s'$. The rule for mutation is representative:

$$\frac{H; D; S; e \Downarrow_p V}{H; D; S; v := e \longrightarrow H, v \mapsto V; D; S, v \mapsto p; \textbf{skip}}$$

It updates both the heap H and the bookkeeping map S. A similar rule uses the dynamically-tracked expression judgment and also updates D.

Soundness. To express our soundness property, we define a *well-formedness* criterion that states that a dynamic probability field map D and a static context Γ together form lower bounds on the "true" probabilities in S. We write this property as $\vdash D, S : \Gamma$.

Definition 1 (Well-Formed)**.** $\vdash D, S : \Gamma$ *iff for all* $v \in \Gamma$*,*

- *If* $\Gamma(v) = \texttt{@}\mathsf{Approx}(p)\ \tau$*, then* $p \leq S(v)$ *or* $v \notin S$*.*
- *If* $\Gamma(v) = \texttt{@}\mathsf{Dyn}\ \tau$*, then* $D(v) \leq S(v)$ *or* $v \notin S$*.*

The language's soundness theorem states that D and S remain well-formed through the small-step statement evaluation semantics.

Theorem 1 (Soundness)**.** *For all programs* s *with no* endorse *expressions, for all* $n \in \mathbb{N}$ *where* $\cdot; \cdot; \cdot; s \longrightarrow^n H; D; S; s'$*, if* $\cdot \vdash s : \Gamma$*, then* $\vdash D, S : \Gamma$*.*

See Appendix A for the full proof of the theorem. The appendix also states an erasure theorem to show that S does not affect the actual operation of the program: its only purpose is to define soundness for the language.

8. Evaluation

We implemented DECAF and evaluated it using a variety of approximation-tolerant applications. The goals of this evaluation were twofold: to gain experience with DECAF's language features; and to apply it as a testbed to examine the implications of software constraints for hardware research.

8.1 Implementation

We implemented a type checker, inference system, and runtime for DECAF as an extension to Java. The implementation extends the simpler EnerJ type system [21] and is similarly based on Java 8's extensible type annotations [7]. The compiler uses AST instrumentation and a runtime library to implement dynamic tracking for the @Dyn qualifier. For Java arrays, the implementation uses conservative object-granularity type checking and dynamic tracking.

Application	Description	Build Time	LOC	@Approx	@Approx(p)	@Dyn	Approx	Dyn
fft	Fourier transform	2 sec	747	37	11	23	7%	55%
imagefill	Bar code recognition	14 min	344	76	20	0	45%	<1%
lu	LU decomposition	1 min	775	63	9	12	24%	<1%
mc	Monte Carlo approximation	2 min	562	67	8	6	21%	<1%
raytracer	3D image reading	1 min	511	38	4	2	12%	44%
smm	Sparse matrix multiply	1 min	601	37	4	4	28%	28%
sor	Successive over-relaxation	19 min	589	43	3	3	63%	<1%
zxing	Bar code recognition	16 min	13180	220	98	4	31%	<1%

Table 1: Benchmarks used in the evaluation. The middle set of columns show the static density of DECAF annotations in the Java source code. The final two columns show the dynamic proportion of operations in the program that were approximate (as opposed to implicitly reliable) and dynamically tracked (both approximate and reliable operations can be dynamically tracked).

The compiler generates constraints for the Z3 SMT solver [6] to check satisfiability, emit warnings, and tune inferred operator probabilities. The constraint systems exercise Z3's complete solver for nonlinear real arithmetic. To stay within the reach of this complete solver, we avoided generating any integer-valued constraints, which can quickly cause Z3's heuristics to reject the query.

Z3 does not directly support optimization problems, so we use a straightforward search strategy to minimize the objective function. The linear search executes queries repeatedly while reducing the bound on the objective until the solver reports unsatisfiability or times out (after 1 minute in our experiments). The optimization strategy's dependence on real-time behavior means that the optimal solutions are somewhat nondeterministic. Also, more complex constraint systems can time out earlier and lead to poorer optimization results—meaning that adding constraints meant to improve the solution can paradoxically worsen it. In practice, we observe this adverse effect for two benchmarks where hardware constraints cause an explosion in solver time (see below).

We optimize programs according to a static proxy for a program's overall efficiency (see Section 4). Our evaluation tests this objective's effectiveness as a static proxy for dynamic behavior by measuring dynamic executions.

8.2 Experimental Setup

We consider an approximate processor architecture where arithmetic operations may have a probability of failure, mirroring recent work in hardware design [9, 13, 26, 27]. Because architecture research on approximate computing is at an early stage, we do not model a specific CPU design: there is no consensus in the literature on which reliability parameters are best or how error probabilities translate into energy savings. Instead, we design our evaluation to advance the discussion by exploring the constraints imposed by language-level quality demands. We explore error levels in a range commensurate with current proposals—correctness probabilities 99% and higher—to inform the specific parameters that hardware should support. Researchers can use this platform-agnostic data to evaluate architecture designs.

We implemented a simulation infrastructure that emulates such a machine with tunable instruction reliability. The simulator is based on the freely available implementation used by Sampson et al. [21], which uses a source-to-source translation of Java code to invoke a run-time library that injects errors and collects execution statistics. To facilitate simulation, three pieces of data are exported at compile time and imported when the runtime is launched. Every operator used in an approximate expression is exported with its reliability. When an operator is encountered, the simulator looks up its reliability or assumes reliable execution if the operator is not defined. To facilitate @Dyn expression tracking, the compiler exports each variable's reliability and the runtime imports this data to initialize dynamic reliability fields. Finally, the run-time uses a mapping from invocations to function variants to look up the reliabilities specialized functions.

Performance statistics were collected on a 4-core, 2.9 GHz Intel Xeon machine with 2-way SMT and 8 GB RAM running Linux. We used OpenJDK 1.7.0's HotSpot VM and version Z3 version 4.3.1.

8.3 Benchmarks and Annotation

We evaluate a set of Java benchmarks from the EnerJ benchmark suite [21]. Table 1 lists the applications and statistics about their source code and annotations.

The benchmarks' original EnerJ annotations distinguish approximation-tolerant variables (marked with @Approx) from reliable variables (the default). To adapt the programs for DECAF, we left most of these type annotations as the inferred @Approx annotation. On the output of each benchmark and on a few salient boundaries between components, we placed concrete @Approx(p) restrictions. These outputs have a variety of types, including single values, arrays of pixels, and strings. Informed by compiler warnings, we used @Dyn for variables involved in loop-carried dependencies where static tracking is insufficient along with check() casts to transition back to static types. Finally, we parameterized

some @Approx annotations to add constraints where they were lacking—i.e., when inferred values flow into endorsements or @Dyn variables exclusively.

For each application, we applied the @Approx(0.9) qualifier to the overall output of the computation. This and other explicit probability thresholds dictate the required reliability for the program's operations, which we measure in this evaluation. We believe these constraints to be representative of practical deployments, but deployment scenarios with looser or tighter output quality constraints will lead to correspondingly different operator probability requirements.

When starting from unannotated code, we find that it is typically possible to follow type errors to find the set of declarations that need @Approx annotations. Programmers only need to apply domain knowledge at module boundaries to pick specific probabilities in an interface. In our benchmarks, which focus on small computational kernels known to be approximate, annotations can be dense. We expect that, in larger applications, approximation will be confined to performance-critical modules: the majority of a larger application's code will be uninteresting for approximation, so DECAF's annotations will be proportionally sparser.

8.4 Results

We use these benchmarks to study the implications of our benchmarks on the design of approximate hardware. Key findings (detailed below) include:

- By tuning a application to match a specific hardware profile, a compiler can achieve better efficiency than with hardware-oblivious optimization. Hardware-targeted optimization improves efficiency even on a simple two-level approximate architecture.
- Most benchmarks benefit from multiple operator probabilities. Processors should provide at least two levels for approximate operations to maximize efficiency.
- Operator correctness probabilities between $1.0 - 10^{-2}$ and $1.0 - 10^{-8}$ are most broadly useful. Probabilities outside this range are less general.

These conclusions reflect our benchmarks and their annotations. The raw data is available for use in future research: `http://sampa.cs.washington.edu/decaf`

8.5 Sensitivity to Hardware Reliability Levels

An ideal approximate machine would allow arbitrarily fine-grained reliability tuning to exactly match the demands of every operation in any application. Realistically, however, an architecture will likely need to provide a fixed set of probability levels. The number of levels will likely be small to permit efficient instruction encoding. We use DECAF to evaluate the impact of this restriction by simulating different hardware configurations alongside the ideal case.

We simulate architectural configurations with two to eight levels of reliability. A two-level machine has one reliable

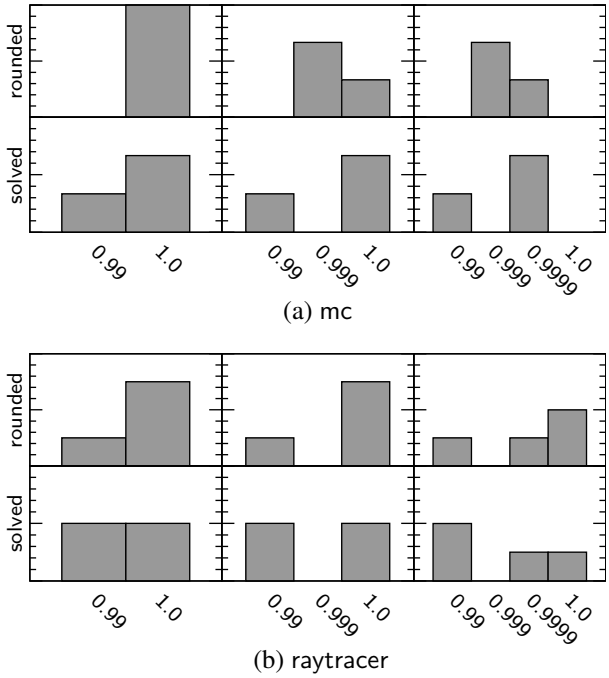

Figure 2: Sensitivity to hardware restrictions for two benchmarks. The horizontal axes show the probability levels; the vertical axes reflect the fraction of approximate operations in an execution assigned to each level. The *rounded* executions were assigned to levels after unrestricted solving; the *solved* executions used the hardware profile during type inference.

operation mode ($p = 1.0$) and one approximate mode, for which we choose $p = 0.99$. This configuration resembles the Truffle microarchitecture, which provides only one approximate mode [9]. We evaluate multi-level configurations that each add a probability level with one more "nine": $p = 0.999$, $p = 0.9999$, and so on, approaching fully reliable operation. Architecture proposals suggest that even low probabilities of error can yield energy savings [11, 12, 14].

Solving vs. rounding levels. To run a DECAF-compiled program on realistic approximate hardware, two strategies are possible for selecting the probability level for each operation. A simplistic approach rounds the inferred probabilities up to the nearest level. The compiler can potentially do better by using the SMT solver to apply constraints during type inference if the architecture is known ahead of time.

Figure 2 compares the two approaches, denoted *solving* and *rounding*, for two of our benchmarks on two-, three-, and four-level machines. Hardware-constrained solving shifts the distribution toward lower probabilities in each of the three machines. When mc runs on a three-level machine, for example, the simple rounding strategy rarely uses the lowest $p = 0.99$ reliability level; if we instead inform the solver that this level is available, the benchmark can use it for a third of its approximate operations. A similar effect arises for raytracer, for which the solver assigns the lowest reli-

ability level to about half of the operations executed while rounding makes the majority of operations fully reliable.

These differences suggest that optimizing an approximate program for a specific hardware configuration can enable significant energy savings, *even for simple approximate machines with only two probability levels*. DECAF's solver-based tuning approach enables this kind of optimization.

While solving for hardware constraints can lead to better efficiency at run time, it can also be more expensive at compile time. The SMT queries for most benchmarks took only a few minutes, but two outliers—sor and zxing—took much longer when level constraints were enabled. For sor, solving succeeded for machine configurations up to four levels but exceeded a 30-minute timeout for larger level counts; zxing timed out even in the two-level configuration. In the remainder of this evaluation, we use the more sophisticated solving scheme, except for these cases where solving times out and we fall back to the cheaper rounding strategy.

Probability granularity. More hardware probability levels can enable greater efficiency gains by better matching applications' probability requirements. Figure 3 depicts the allocation of programs' operations to reliability levels for a range of hardware configurations from 2 to 8 levels. In this graphic, white and lighter shades indicate more reliable execution and correspondingly lower efficiency gains; darker shades indicate more opportunity for energy savings.

Five of the eight benchmarks use multiple operator probability levels below 1.0 when optimized for hardware that offers this flexibility. This suggests that multi-level approximate hardware designs like QUORA [26] can unlock more efficiency gains in these benchmarks than simpler single-probability machines like Truffle [9]. The exceptions are imagefill, lu, and smm, where a single probability level seems to suffice for the majority of operations.

Most of our benchmarks exhibit diminishing returns after a certain number of levels. For example, mc increases its amount of approximation up to four levels but does not benefit from higher level counts. Similarly, imagefill's benefits do not increase after six levels. In contrast, raytracer and zxing see improvements for configurations up to eight levels.

In an extreme case, smm falls back to reliable execution for nearly all of its operations in every configuration we simulated except for the eight-level machine. This suggests that a two-level machine would suffice for this benchmark, provided the single approximate operation probability were high enough. On the other hand, specializing a two-level architecture to this outlier would limit potential efficiency gains for other applications.

Increasing reliability levels do not strictly lead to efficiency benefits in DECAF's solver-based approach. For sor, the added constraints for more granular hardware levels lead to a more complex SMT solver query and eventually timeouts. After four levels, the solver failed to optimize the benchmark and we fell back to the naïve rounding strategy, which leads to lower efficiency gains. These timeouts are partially due to DECAF's straightforward encoding of program and hardware constraints; future work on optimizing DECAF's constraint systems for efficient solving could make larger level counts more tractable.

Comparison to ideal. An ideal approximate architecture that features arbitrary probability levels could offer more flexibility at the extremes of the probability range. To evaluate the importance of higher and lower levels, we simulated an ideal continuous machine. Figure 4 shows the fraction of approximate operations in executions of each benchmark that used probabilities below the range of our realistic machines (below 99% probability) and above the range (above $p = 1.0 - 10^{-8}$). The figure also shows the operations that executed with probability exactly 1.0 even on this continuous architecture, indicating that they were constrained by program requirements rather than hardware limitations.

For all but one application, most operations lie in the range of probabilities offered by our discrete machine simulations. Only three benchmarks show a significant number of operations with probabilities below 99%, and one outlier, imagefill, uses these low-probability operations for nearly all of its approximable computations. The only benchmark that significantly uses higher-probability operations is zxing, where about 20% of the operations executed had a probability greater than $1.0 - 10^{-8}$. Among our benchmarks, the $0.99 \leq p \leq 0.99999999$ probability range suffices to capture most of the flexibility offered by an ideal machine.

Example energy model. This evaluation measures error probabilities with the goal of allowing hardware designers to plug in energy models. (We intend to make the raw data available in full after publication.) To give a sense of the potential savings, we apply a simple energy model based on EnerJ [21]: a correctness probability of 0.99 yields 30% energy savings over a precise operation, $p = 10^{-4}$ saves 20%, $p = 10^{-6}$ saves 10%, and other levels are interpolated. More optimistic hardware proposals exist (e.g., Venkataramani et al. [26]), but EnerJ's conservative CPU-based model serves as a useful point of comparison. On an eight-level machine, the total operation energy saved is:

Benchmark	Rounded	Solved
fft	<1%	<1%
imagefill	7%	22%
lu	<1%	9%
mc	5%	23%
raytracer	1%	20%
smm	2%	2%
sor	12%	—
zxing	1%	—

The table shows the modeled energy reduction for both the hardware-oblivious rounding strategy and the platform-specific solving strategy (except where the solver timed out).

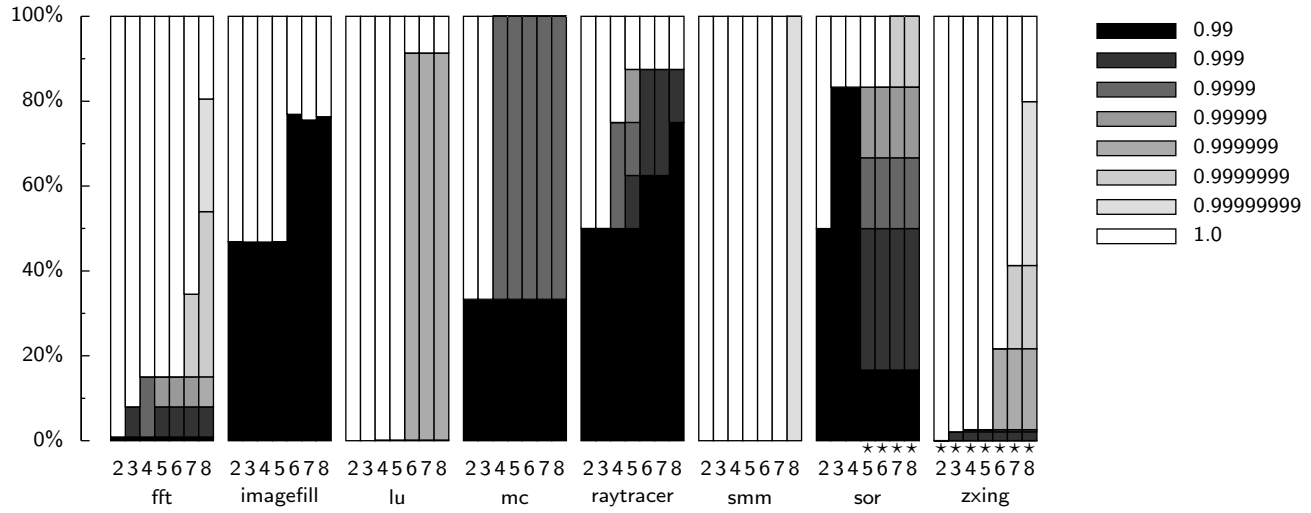

Figure 3: Operator probability breakdowns. Each bar shows a hardware configuration with a different number of levels. Darker shades indicate lower probabilities and correspondingly higher potential energy savings. Bars marked $\star$ used the cheaper rounding strategy instead of solving to determine levels.

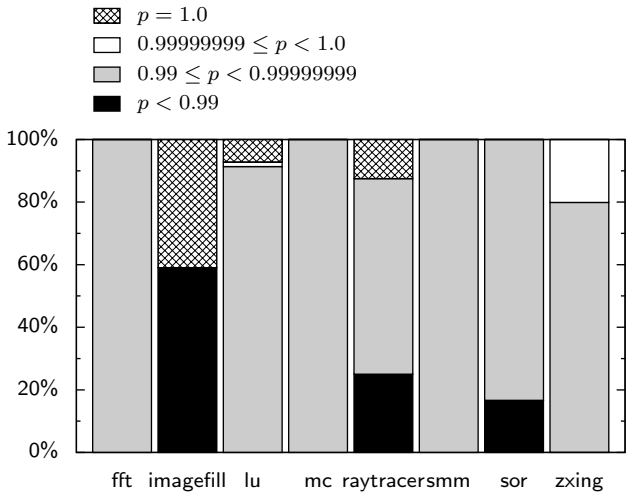

Figure 4: Approximate operation probabilities on an ideal continuous machine. Gray boxes show the probability range accommodated by our simulated discrete-level machines, white boxes represent higher-reliability operations, and black boxes are lower-reliability operations. Hatched boxes indicate approximate operations that are forced to execute reliably by program constraints, even on ideal hardware.

The results reflect the above finding that solving yields better savings than rounding after the fact.

8.6 Interprocedural Inference and Specialization

In all of our benchmarks, we used the inferred @Approx qualifier on function parameters and return types to let the compiler propagate constraints interprocedurally. This let us write simpler annotations that directly encoded our desired output correctness constraints and avoid artificially aligning them with function boundaries. In some benchmarks—namely, lu and zxing—multiple call sites to these inferred functions allowed the compiler to specialize variants and improve efficiency.

In lu, for example, specialization was critical to making the benchmark take advantage of approximate hardware. That benchmark uses a utility function that copies approximate arrays. The factorization routine has three calls to the copying function, and each of the intermediate arrays involved have varying impact on the output of the benchmark. When we limit the program to $k = 1$ function variants—disabling function specialization—all three of these intermediates are constrained to have identical correctness probability, and all three must be as reliable as the least tolerant among them. As a result, the benchmark as a whole exhibits very little approximate execution: more than 99% of its approximate operations are executed reliably ($p = 1.0$). By allowing $k = 2$ function specializations, however, the compiler reduces the fraction to 8%, and $k = 3$ specializations further reduce it to 7%. A similar pattern arises in the zxing benchmark, where utility functions on its central data structure—a bit-matrix class used to hold black-and-white pixel values—are invoked from different contexts throughout the program.

8.7 Dynamic Tracking

The @Dyn type qualifier lets programmers request dynamic probability tracking, in exchange for run-time overhead, when DECAF's static tracking is too conservative. Table 1 shows the number of types we annotated with @Dyn in each benchmark. Dynamic tracking was necessary at least once

in every benchmark except one (imagefill). Most commonly, @Dyn applied in loops that accumulate approximate values. For example, zxing has a loop that searches for image patterns that suggest the presence of parts of a QR code. The actual detection of each pattern can be statically tracked, but the loop also accumulates the total size of the image patterns. Since the loop's trip count depends on the input image, dynamic tracking was necessary for precision: no nonzero static bound on the size variable's probability would suffice.

Table 1 also shows the fraction of operations in an execution of each benchmark that required dynamic tracking. In five of our eight benchmarks, less than 1% of the operations in the program need to be dynamically tracked, suggesting that energy overhead would be minimal. In the remaining three benchmarks, a significant portion of the application's approximate and reliable operations required dynamic tracking. (Recall that operations on @Dyn-typed variables are reliable by default but still require propagation of probability information.) In the worst case, fft uses dynamic tracking for 55% of the operations in its execution.

In a simple implementation, each dynamically tracked operation expands out to two operations, so the percentage of dynamically tracked operations is equivalent to the overhead incurred. An optimizing compiler, however, can likely coalesce and strength-reduce the multiplications-by-constants that make up tracking code. In fft, for example, an inner loop reads two array elements, updates them each with a series of four approximate operations, and writes them back. A standard constant-propagation optimization could coalesce the four tracking operations to a single update. In other cases, such as zxing's pattern-search loop described above, the correctness probability loss is directly proportional to the loop trip count. Standard loop optimizations could hoist these updates out of the loop and further reduce overhead.

8.8 Tool Performance

Table 1 lists the running time of the inference system for each benchmark. The total time includes time spent on the initial system-satisfiability query, the optimization query series, parsing and analyzing the Java source code, and checking for DECAF constraint warnings. Most of the time is spent in optimization, so it can be faster to produce a satisfying but suboptimal type assignment. The optimization queries have a timeout of one minute, so the final SMT query in the series can take at most this long; for several benchmarks, the solver returns *unsatisfiable* before this timeout is reached. The compiler typically runs in about 1–20 minutes. One outlier is fft, whose constraint system is fast to solve because of the benchmark's reliance on dynamic tracking.

These measurements are for a continuous configuration of the system rather than a more expensive level-constrained version. Solver times for hardware-constrained inference are comparable, except for the two benchmarks mentioned above that time out: sor and zxing.

9. Related Work

Programming systems for approximate computing have recently become an area of interest for programming languages research. This work's primary distinctions are its focus on types for annotation flexibility, exploitation of recent multi-level architectures, and integration of static and dynamic tracking.

Some approaches to approximate programming, including Relax [5] and EnerJ [21], focus primarily on *safety* properties: preventing bugs where approximation contaminates pointers or causes wild control flow. These proposals have focused on deciding which parts of a computation may be subject to error. Our work addresses the related but distinct issue of *quality*, which concerns the degree of error allowable in a program.

DECAF is most closely related to other systems for inferring operator reliabilities to meet programmer-specified correctness bounds. Chisel [16] uses an integer linear programming formulation to choose which operations in a function should be made approximate to meet a bound on the function's return value. Similarly, ExpAX [10] uses a dataflow analysis combined with a genetic algorithm to determine which operations in a program to approximate based on an overall quality bound for the entire application. DECAF's type-based approach is new in four important ways: (1) Where prior work assumes a single level of hardware approximation, DECAF is the first language we are aware of that targets approximate architectures with multiple probabilities. We show empirically that multi-level architectures can offer applications better efficiency for the same output reliability than simpler single-level approximation. (2) DECAF shows how to augment static guarantees with run-time monitoring. (3) DECAF can specialize functions according to quality demands in calling contexts. (4) Probability type qualifiers admit flexible annotation that scales with programmer effort.

Probabilistic assertions [23] express statistical program properties: for example, output quality requirements in approximate programs. The verifier supports arbitrary accuracy metrics, but it cannot produce conservative bounds or infer partially-specified approximations. Similarly, Bornholt et al.'s Uncertain<T> [1] uses a library-based approach to compose probability distributions at run time, a process that resembles DECAF's dynamic probability tracking.

DECAF also relates to systems for optimizing floating-point precision. Precimonious [19] chooses bitwidths to meet an output precision bound, while Schkufza et al. [24] design a superoptimizer that takes an accurate kernel and synthesizes new implementations that approximate the original behavior.

While type inference typically reconstructs types so that any solution leads to the same program semantics, DECAF's inference determines the program's probabilistic behavior. Similarly, Chlorophyll uses type inference to assign data and

computations to processing elements to minimize communication [17]. That work also uses an SMT solver to optimize an objective function.

10. Conclusion

Approximate programming models need tools that help developers decide how much reliability is necessary throughout an algorithm. Especially when probabilistic operation is involved, manual reasoning about individual operator reliabilities can be tedious and error-prone. On the other hand, fully automatic approaches are also problematic: since approximation has broad implications for software correctness, programmers sometimes need fine-grained control over its effects. An opaque auto-tuner sacrifices programmer visibility and control. DECAF's solver-aided type inference offers an intermediate solution that lets programmers write constraints only where they are most relevant. Combined with code specialization and dynamic tracking, DECAF gives programmers flexible control over the efficiency–accuracy trade-offs offered by approximate hardware.

Acknowledgments

This work was supported in part by C-FAR, one of the six SRC STARnet Centers, sponsored by MARCO and DARPA, by NSF grant #1216611, by a Qualcomm Innovation Fellowship, and by gifts from Microsoft.

References

[1] J. Bornholt, T. Mytkowicz, and K. S. McKinley. Uncertain<T>: A first-order type for uncertain data. In *ASPLOS*, 2014.

[2] M. Carbin, S. Misailovic, and M. Rinard. Verifying quantitative reliability of programs that execute on unreliable hardware. In *OOPSLA*, 2013.

[3] M. Cohen, H. S. Zhu, E. E. Senem, and Y. D. Liu. Energy types. In *OOPSLA*, 2012.

[4] K. D. Cooper, M. W. Hall, and K. Kennedy. A methodology for procedure cloning. *Comput. Lang.*, 19(2):105–117, Apr. 1993.

[5] M. de Kruijf, S. Nomura, and K. Sankaralingam. Relax: An architectural framework for software recovery of hardware faults. In *ISCA*, 2010.

[6] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS/ETAPS*, 2008.

[7] M. D. Ernst. Type Annotations specification (JSR 308). `http://types.cs.washington.edu/jsr308/`, September 12, 2008.

[8] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. In *MICRO*, 2012.

[9] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Architecture support for disciplined approximate programming. In *ASPLOS*, 2012.

[10] H. Esmaeilzadeh, K. Ni, and M. Naik. Expectation-oriented framework for automating approximate programming. Technical Report GT-CS-13-07, Georgia Institute of Technology, 2013. URL `http://hdl.handle.net/1853/49755`.

[11] C. Hizli. Energy aware probabilistic arithmetics. Master's thesis, Eindhoven University of Technology, 2013.

[12] A. Kahng, S. Kang, R. Kumar, and J. Sartori. Designing a processor from the ground up to allow voltage/reliability tradeoffs. In *HPCA*, 2010.

[13] U. Karpuzcu, I. Akturk, and N. S. Kim. Accordion: Toward soft near-threshold voltage computing. In *HPCA*, 2014.

[14] Z. M. Kedem, V. J. Mooney, K. K. Muntimadugu, and K. V. Palem. An approach to energy-error tradeoffs in approximate ripple carry adders. In *ISPLED*, 2011.

[15] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn. Flikker: Saving DRAM refresh-power through critical data partitioning. In *ASPLOS*, 2011.

[16] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. C. Rinard. Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels. In *OOPSLA*, 2014.

[17] P. M. Phothilimthana, T. Jelvis, R. Shah, N. Totla, S. Chasins, and R. Bodik. Chlorophyll: Synthesis-aided compiler for low-power spatial architectures. In *PLDI*, 2014.

[18] M. F. Ringenburg, A. Sampson, I. Ackerman, L. Ceze, and D. Grossman. Monitoring and debugging the quality of results in approximate programs. In *ASPLOS*, 2015.

[19] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough. Precimonious: Tuning assistant for floating-point precision. In *Supercomputing*, 2013.

[20] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke. Paraprox: Pattern-based approximation for data parallel applications. In *ASPLOS*, 2014.

[21] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. EnerJ: Approximate data types for safe and general low-power computation. In *PLDI*, 2011.

[22] A. Sampson, J. Nelson, K. Strauss, and L. Ceze. Approximate storage in solid-state memories. In *MICRO*, 2013.

[23] A. Sampson, P. Panchekha, T. Mytkowicz, K. S. McKinley, D. Grossman, and L. Ceze. Expressing and verifying probabilistic assertions. In *PLDI*, 2014.

[24] E. Schkufza, R. Sharma, and A. Aiken. Stochastic optimization of floating-point programs with tunable precision. In *PLDI*, 2014.

[25] S. Tobin-Hochstadt and M. Felleisen. Interlanguage migration: From scripts to programs. In *Dynamic Languages Symposium (DLS)*, 2006.

[26] S. Venkataramani, V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan. Quality programmable vector processors for approximate computing. In *MICRO*, 2013.

[27] M. Weber, M. Putic, H. Zhang, J. Lach, and J. Huang. Balancing adder for error tolerant applications. In *International Symposium on Circuits and Systems (ISCAS)*, 2013.

A. Full Semantics and Proof

This appendix expands on the formalism overview in Section 7. We present the full syntax, static semantics, and dynamic semantics for the core DECAF language. We prove a *soundness* theorem that embodies the probability type system's fundamental accuracy guarantee.

A.1 Syntax

We formalize a core of DECAF without inference. The syntax for statements, expressions, and types is:

$$
\begin{aligned}
s &::= T\ v := e \mid v := e \mid s\ ;\ s \mid \textbf{if}\ e\ s\ s \mid \textbf{while}\ e\ s \mid \textbf{skip} \\
e &::= c \mid v \mid e \oplus_p e \mid \mathsf{endorse}(p, e) \mid \mathsf{check}(p, e) \mid \mathsf{track}(p, e) \\
\oplus &::= + \mid - \mid \times \mid \div \\
T &::= q\ \tau \\
q &::= \mathtt{@Approx}(p) \mid \mathtt{@Dyn} \\
\tau &::= \mathsf{int} \mid \mathsf{float} \\
v &\in \text{variables},\ c \in \text{constants},\ p \in [0.0, 1.0]
\end{aligned}
$$

For the purpose of the static and dynamic semantics, we also define values V, heaps H, dynamic probability maps D, true probability maps S, and static contexts Γ:

$$
\begin{aligned}
V &::= c \mid \square \\
H &::= \cdot \mid H, v \mapsto V \\
D &::= \cdot \mid D, v \mapsto p \\
S &::= \cdot \mid S, v \mapsto p \\
\Gamma &::= \cdot \mid \Gamma, v \mapsto T
\end{aligned}
$$

We define $H(v)$, $D(v)$, $S(v)$, and $\Gamma(v)$ to denote variable lookup in these maps.

A.2 Typing

The type system defines the static semantics for the core language. We first give typing judgments for expressions and then for statements.

A.2.1 Operator Typing

We introduce a helper "function" that determines the unqualified result type of a binary arithmetic operator.

$\boxed{\text{optype}(\tau_1, \tau_2) = \tau_3}$

$$
\text{optype}(\tau, \tau) = \tau \qquad \text{optype}(\mathsf{int}, \mathsf{float}) = \mathsf{float}
$$

$$
\text{optype}(\mathsf{float}, \mathsf{int}) = \mathsf{float}
$$

Now we can give the types of the binary operator expressions themselves. There are two cases: one for statically-typed operators and one for dynamic tracking. The operands may not mix static and dynamic qualifiers (recall that the compiler inserts track casts to introduce dynamic tracking when necessary).

$\boxed{\Gamma \vdash e : T}$

OP-STATIC-TYPES

$$
\frac{\Gamma \vdash e_1 : \mathtt{@Approx}(p_1)\ \tau_1 \qquad \Gamma \vdash e_2 : \mathtt{@Approx}(p_2)\ \tau_2 \qquad \tau_3 = \text{optype}(\tau_1, \tau_2) \qquad p' = p_1 \cdot p_2 \cdot p_{\text{op}}}{\Gamma \vdash e_1 \oplus_{p_{\text{op}}} e_2 : \mathtt{@Approx}(p')\ \tau_3}
$$

OP-DYN-TYPES

$$
\frac{\Gamma \vdash e_1 : \mathtt{@Dyn}\ \tau_1 \qquad \Gamma \vdash e_2 : \mathtt{@Dyn}\ \tau_2 \qquad \tau_3 = \text{optype}(\tau_1, \tau_2)}{\Gamma \vdash e_1 \oplus_p e_2 : \mathtt{@Dyn}\ \tau_3}
$$

In the static case, the output probability is the product of the probabilities for the left-hand operand, right-hand operand, and the operator itself. Section 3 gives the probabilistic intuition behind this rule.

A.2.2 Other Expressions

The rules for constants and variables are straightforward. Literals are given the precise ($p = 1.0$) type.

CONST-INT-TYPES

$$
\frac{c \text{ is an integer}}{\Gamma \vdash c : \mathtt{@Approx}(1.0)\ \mathsf{int}}
$$

CONST-FLOAT-TYPES

$$
\frac{c \text{ is not an integer}}{\Gamma \vdash c : \mathtt{@Approx}(1.0)\ \mathsf{float}}
$$

VAR-TYPES

$$
\frac{T = \Gamma(v)}{\Gamma \vdash v : T}
$$

Endorsements, both checked and unchecked, produce the explicitly requested type. (Note that check is sound but endorse is potentially unsound: our main soundness theorem, at the end of this appendix, will exclude the latter from the language.) Similarly, track casts produce a dynamically-tracked type given a statically-tracked counterpart.

ENDORSE-TYPES

$$
\frac{\Gamma \vdash e : q\ \tau}{\Gamma \vdash \mathsf{endorse}(p, e) : \mathtt{@Approx}(p)\ \tau}
$$

CHECK-TYPES

$$
\frac{\Gamma \vdash e : \mathtt{@Dyn}\ \tau}{\Gamma \vdash \mathsf{check}(p, e) : \mathtt{@Approx}(p)\ \tau}
$$

TRACK-TYPES

$$
\frac{\Gamma \vdash e : \mathtt{@Approx}(p')\ \tau \qquad p \leq p'}{\Gamma \vdash \mathsf{track}(p, e) : \mathtt{@Dyn}\ \tau}
$$

A.2.3 Qualifiers and Subtyping

A simple subtyping relation, introduced in Section 3, makes high-probability types subtypes of their low-probability counterparts.

$\boxed{T_1 \prec T_2}$

SUBTYPING

$$
\frac{p \geq p'}{\mathtt{@Approx}(p)\ \tau \prec \mathtt{@Approx}(p')\ \tau}
$$

Subtyping uses a standard subsumption rule.

$$\textsc{subsumption}\ \frac{T_1 \prec T_2 \qquad \Gamma \vdash e : T_1}{\Gamma \vdash e : T_2}$$

A.2.4 Statement Typing

Our typing judgment for statements builds up the context Γ.

$\boxed{\Gamma_1 \vdash s : \Gamma_2}$

$$\textsc{skip-types}\ \frac{}{\Gamma \vdash \mathsf{skip} : \Gamma} \qquad \textsc{seq-types}\ \frac{\Gamma_1 \vdash s_1 : \Gamma_2 \qquad \Gamma_2 \vdash s_2 : \Gamma_3}{\Gamma_1 \vdash s_1; s_2 : \Gamma_3}$$

$$\textsc{decl-types}\ \frac{\Gamma \vdash e : T \qquad v \notin \Gamma}{\Gamma \vdash T\ v := e : \Gamma, v : T} \qquad \textsc{mutate-types}\ \frac{\Gamma \vdash e : T \qquad \Gamma(v) = T}{\Gamma \vdash v := e : \Gamma}$$

$$\textsc{if-types}\ \frac{\Gamma \vdash e : \texttt{@Approx}(1.0)\ \tau \qquad \Gamma \vdash s_1 : \Gamma_1 \qquad \Gamma \vdash s_2 : \Gamma_2}{\Gamma \vdash \textbf{if}\ e\ s_1\ s_2 : \Gamma}$$

$$\textsc{while-types}\ \frac{\Gamma \vdash e : \texttt{@Approx}(1.0)\ \tau \qquad \Gamma \vdash s : \Gamma'}{\Gamma \vdash \textbf{while}\ e\ s : \Gamma}$$

The conditions in if and while statements are required to have the precise type ($p = 1.0$).

A.3 Operational Semantics

We use a large-step operational semantics for expressions and small-step semantics for statements. Both are nondeterministic: values produced by approximate operators can produce either an error value $\square$ or a concrete number.

A.3.1 Expression Semantics

There are two judgments for expressions: one for statically typed expressions and one where dynamic tracking is used. The former, $H; D; S; e \Downarrow_p V$, indicates that the expression e produces a value V, which is either a constant c or the error value $\square$, and p is the probability that $V \neq \square$. The latter judgment, $H; D; S; e \Downarrow_p V, p_d$, models dynamically-tracked expression evaluation. In addition to a value V, it also produces a computed probability value p_d reflecting the compiler's conservative bound on the reliability of e's value. That is, p is the "true" probability that $V \neq \square$ whereas p_d is the dynamically computed conservative bound for p.

In these judgments, H is the heap mapping variables to values and D is the dynamic probability map for @Dyn-typed variables maintained by the compiler. The S probability map is used for our type soundness proof: it maintains the actual probability that a variable is correct.

Constants Literals are always tracked statically.

$$\textsc{const}\ \frac{}{H; D; S; c \Downarrow_{1.0} c}$$

Variables Variable lookup is dynamically tracked when the variable is present in the tracking map D. The probability $S(v)$ is the chance that the variable does not hold $\square$.

$$\textsc{var}\ \frac{v \notin D}{H; D; S; v \Downarrow_{S(v)} H(v)} \qquad \textsc{var-dyn}\ \frac{v \in D}{H; D; S; v \Downarrow_{S(v)} H(v), D(v)}$$

Endorsements Unchecked (unsound) endorsements only apply to statically-tracked values and do not affect the correctness probability.

$$\textsc{endorse}\ \frac{H; D; S; e \Downarrow_p V}{H; D; S; \mathsf{endorse}(p_e, e) \Downarrow_p V}$$

Checked Endorsements Checked endorsements apply to dynamically-tracked values and produce statically-tracked values. The tracked probability must meet or exceed the check's required probability; otherwise, evaluation gets stuck. (Our implementation throws an exception.)

$$\textsc{check}\ \frac{H; D; S; e \Downarrow_p V, p_1 \qquad p_1 \geq p_2}{H; D; S; \mathsf{check}(p_2, e) \Downarrow_p V}$$

Tracking The static-to-dynamic cast expression allows statically-typed values to be combined with dynamically-tracked ones. The tracked probability field for the value is initialized to match the explicit probability in the expression.

$$\textsc{track}\ \frac{H; D; S; e \Downarrow_p V}{H; D; S; \mathsf{track}(p_d, e) \Downarrow_p V, p_d}$$

Operators Binary operators can be either statically tracked or dynamically tracked. In each case, either operand can be the error value or a constant. When either operand is $\square$, the result is $\square$. When both operands are non-errors, the operator itself can (nondeterministically) produce either $\square$ or a correct result. The correctness probability, however, is the same for all three rules: intuitively, the probability itself is deterministic even though the semantics overall are nondeterministic.

In these rules, $c_1 \oplus c_2$ without a probability subscript denotes the appropriate binary operation on integer or floating-point values. The statically-tracked cases are:

$$\textsc{op}\ \frac{H; D; S; e_1 \Downarrow_{p_1} c_1 \qquad H; D; S; e_2 \Downarrow_{p_2} c_2 \qquad p = p_1 \cdot p_2 \cdot p_{\text{op}}}{H; D; S; e_1 \oplus_{p_{\text{op}}} e_2 \Downarrow_p c_1 \oplus c_2}$$

$$\textsc{op-operator-incorrect}\ \frac{H; D; S; e_1 \Downarrow_{p_1} c_1 \qquad H; D; S; e_2 \Downarrow_{p_2} c_2 \qquad p = p_1 \cdot p_2 \cdot p_{\text{op}}}{H; D; S; e_1 \oplus_{p_{\text{op}}} e_2 \Downarrow_p \square}$$

$$\textsc{op-operands-incorrect}\ \frac{H; D; S; e_1 \Downarrow_{p_1} \square \text{ or } H; D; S; e_2 \Downarrow_{p_2} \square \qquad p = p_1 \cdot p_2 \cdot p_{\text{op}}}{H; D; S; e_1 \oplus_{p_{\text{op}}} e_2 \Downarrow_p \square}$$

The dynamic-tracking rules are similar, with the additional propagation of the conservative probability field.

OP-DYN

$$\frac{H; D; S; e_1 \Downarrow_{p_1} c_1, p_{d1} \quad H; D; S; e_2 \Downarrow_{p_2} c_2, p_{d2} \quad p = p_1 \cdot p_2 \cdot p_{\text{op}}}{H; D; S; e_1 \oplus_{p_{\text{op}}} e_2 \Downarrow_p c_1 \oplus c_2,\ p_{d1} \cdot p_{d2} \cdot p_{\text{op}}}$$

OP-DYN-OPERATOR-INCORRECT

$$\frac{H; D; S; e_1 \Downarrow_{p_1} c_1, p_{d1} \quad H; D; S; e_2 \Downarrow_{p_2} c_2, p_{d2} \quad p = p_1 \cdot p_2 \cdot p_{\text{op}}}{H; D; S; e_1 \oplus_{p_{\text{op}}} e_2 \Downarrow_p \square,\ p_{d1} \cdot p_{d2} \cdot p_{\text{op}}}$$

OP-DYN-OPERANDS-INCORRECT

$$\frac{H; D; S; e_1 \Downarrow_{p_1} \square, p_{d1} \text{ or } H; D; S; e_2 \Downarrow_{p_2} \square, p_{d2} \quad p = p_1 \cdot p_2 \cdot p_{\text{op}}}{H; D; S; e_1 \oplus_{p_{\text{op}}} e_2 \Downarrow_p \square,\ p_{d1} \cdot p_{d2} \cdot p_{\text{op}}}$$

A.3.2 Statement Semantics

The small-step judgment for statements is $H; D; S; s \longrightarrow H'; D'; S'; s'$.

Assignment The rules for assignment (initializing a fresh variable) take advantage of nondeterminism in the evaluation of expressions to nondeterministically update the heap with either a constant or the error value, $\square$.

$\boxed{H; D; s \longrightarrow H'; D'; s'}$

ASSIGN

$$\frac{H; D; S; e \Downarrow_p V}{H; D; S; \texttt{@Approx}(p')\ \tau\ v := e \longrightarrow H, v \mapsto V; D; S, v \mapsto p; \mathbf{skip}}$$

ASSIGN-DYN

$$\frac{H; D; S; e \Downarrow_p V, p_d}{H; D; S; \texttt{@Dyn}\ \tau\ v := e \longrightarrow H, v \mapsto V; D, v \mapsto p_d; S, v \mapsto p; \mathbf{skip}}$$

Mutation works like assignment, but existing variables are overwritten in the heap.

MUTATE

$$\frac{H; D; S; e \Downarrow_p V}{H; D; S; v := e \longrightarrow H, v \mapsto V; D; S, v \mapsto p; \mathbf{skip}}$$

MUTATE-DYN

$$\frac{H; D; e \Downarrow_p V, p_d}{H; D; v := e \longrightarrow H, v \mapsto V; D, v \mapsto p_d; S, v \mapsto p; \mathbf{skip}}$$

Sequencing Sequencing is standard and deterministic.

SEQ-SKIP

$$\frac{}{H; D; S; \mathbf{skip}; s \longrightarrow H; D; S; s}$$

SEQ

$$\frac{H; D; S; s_1 \longrightarrow H'; D'; S'; s_1'}{H; D; S; s_1; s_2 \longrightarrow H'; D'; S'; s_1'; s_2}$$

If and While The type system requires conditions in if and while control flow decisions to be deterministic ($p = 1.0$).

IF-TRUE

$$\frac{H; D; S; e \Downarrow_{1.0} c \quad c \neq 0}{H; D; S; \mathbf{if}\ e\ s_1\ s_2 \longrightarrow H; D; S; s_1}$$

IF-FALSE

$$\frac{H; D; S; e \Downarrow_{1.0} c \quad c = 0}{H; D; S; \mathbf{if}\ e\ s_1\ s_2 \longrightarrow H; D; S; s_2}$$

WHILE

$$\frac{}{H; D; S; \mathbf{while}\ e\ s \longrightarrow H; D; S; \mathbf{if}\ e\ (s; \mathbf{while}\ e\ s)\ \mathbf{skip}}$$

A.4 Theorems

The purpose of the formalism is to express a soundness theorem that shows that DECAF's probability types act as lower bounds on programs' run-time probabilities. We also sketch the proof of a theorem stating that the bookkeeping probability map, S, is eraseable: it is used only for the purpose of our soundness theorem and does not affect the heap.

A.4.1 Soundness

The soundness theorem for the language states that the probability types are lower bounds on the run-time correctness probabilities. Specifically, both the static types @Approx(p) and the dynamically tracked probabilities in D are lower bounds for the corresponding probabilities in S.

To state the soundness theorem, we first define well-formed dynamic states. We write $\vdash D, S : \Gamma$ to denote that the dynamic probability field map D and the actual probability map S are *well-formed* in the static context Γ.

Definition 1 (Well-Formed)**.** $\vdash D, S : \Gamma$ *iff for all* $v \in \Gamma$*,*

- *If* $\Gamma(v) = \texttt{@Approx}(p)\ \tau$*, then* $p \leq S(v)$ *or* $v \notin S$*.*
- *If* $\Gamma(v) = \texttt{@Dyn}\ \tau$*, then* $D(v) \leq S(v)$ *or* $v \notin S$*.*

We can now state and prove the soundness theorem. We first give the main theorem and then two preservation lemmas, one for expressions and one for statements.

Theorem 1 (Soundness)**.** *For all programs* s *with no* endorse *expressions, for all* $n \in \mathbb{N}$ *where* $\cdot; \cdot; \cdot; s \longrightarrow^n H; D; S; s'$*, if* $\cdot \vdash s : \Gamma$*, then* $\vdash D, S : \Gamma$*.*

Proof. Induct on the number of small steps, n. When $n = 0$, both conditions hold trivially since $v \notin \cdot$ for all v.

For the inductive case, we assume that $\cdot; \cdot; \cdot; s \longrightarrow^n H_1; D_1; S_1; s_1$ and $H_1; D_1; S_1; s_1 \longrightarrow H_2; D_2; S_2; s_2$ and that $\vdash D_1, S_1 : \Gamma$. We need to show that $\vdash D_2, S_2 : \Gamma$ also. The Statement Preservation lemma, below, applies and meets this goal. $\square$

The first lemma is a preservation property for expressions. We will use this lemma to prove a corresponding preservation lemma for statements, which in turn applies to prove the main theorem.

Lemma 1 (Expression Preservation)**.** *For all expressions e with no* endorse *expressions where $\Gamma \vdash e : T$ and where $\vdash D, S : \Gamma$,*

- *If $T = \texttt{@Approx}(p)\,\tau$, and $H; D; S; e \Downarrow_{p'} V$, then $p \leq p'$.*
- *If $T = \texttt{@Dyn}\,\tau$, and $H; D; S; e \Downarrow_{p'} V, p$, then $p \leq p'$.*

Proof. Induct on the typing judgment for expressions, $\Gamma \vdash e : T$.

***Case* OP-STATIC-TYPES** Here, $e = e_1 \oplus_{p_{op}} e_2$ and $T = \texttt{@Approx}(p)\,\tau$. We also have types for the operands: $\Gamma \vdash e_1 : \texttt{@Approx}(p_1)\,\tau_1$ and $\Gamma \vdash e_2 : \texttt{@Approx}(p_2)\,\tau_2$.

By inversion on $H; D; S; e \Downarrow_{p'} V$ (in any of the cases OP, OP-OPERATOR-INCORRECT, or OP-OPERANDS-INCORRECT), $p' = p'_1 \cdot p'_2 \cdot p_{\text{op}}$ where $H; D; S; e_1 \Downarrow_{p'_1} V_1$ and $H; D; S; e_2 \Downarrow_{p'_2} V_2$.

By applying the induction hypothesis to e_1 and e_2, we have $p_1 \leq p'_1$ and $p_2 \leq p'_2$. Therefore, $p_1 \cdot p_2 \cdot p_{\text{op}} \leq p'_1 \cdot p'_2 \cdot p_{\text{op}}$ and, by substitution, $p \leq p'$.

***Case* OP-DYN-TYPES** The case for dynamically-tracked expressions is similar. Here, $e = e_1 \oplus_{p_{op}} e_2$ and $T = \texttt{@Dyn}\,\tau$, and the operand types are $\Gamma \vdash e_1 : \texttt{@Dyn}\ \tau_1$ and $\Gamma \vdash e_2 : \texttt{@Dyn}\ \tau_2$.

By inversion on $H; D; S; e \Downarrow_{p'} V, p$ (in any of the cases OP-DYN, OP-DYN-OPERATOR-INCORRECT, or OP-DYN-OPERANDS-INCORRECT), $p' = p'_1 \cdot p'_2 \cdot p_{\text{op}}, p = p_{d1} \cdot p_{d2} \cdot p_{\text{op}}$ where $H; D; S; e_1 \Downarrow_{p'_1} V_1, p_{d1}$ and $H; D; S; e_2 \Downarrow_{p'_2} V_2, p_{d2}$.

By applying the induction hypothesis to e_1 and e_2, we have $p_{d1} \leq p'_1$ and $p_{d2} \leq p'_2$. Therefore, $p_{d1} \cdot p_{d2} \cdot p_{\text{op}} \leq p'_1 \cdot p'_2 \cdot p_{\text{op}}$ and, by substitution, $p \leq p'$.

***Case* CONST-INT-TYPES *and* CONST-FLOAT-TYPES** Here, $\Gamma \vdash e : \texttt{@Approx}(p)\,\tau$ where $\tau \in \{\text{int}, \text{float}\}$ and $p = 1.0$.

By inversion on $H; D; S; e \Downarrow_{p'} V$ we get $p' = 1.0$.

Because $1.0 \leq 1.0$, we have $p \leq p'$.

***Case* VAR-TYPES** Here, $e = v$, $\Gamma \vdash v : T$. Destructing T yields two subcases.

- Case $T = \texttt{@Approx}(p)\,\tau$: By inversion on $H; D; S; e \Downarrow_{p'} V$ we have $p' = S(V)$.
  The definition of well-formedness gives us $p \leq S(V)$.
  By substitution, $p \leq p'$.
- Case $T = \texttt{@Dyn}\,\tau$: By inversion on $H; D; S; e \Downarrow_{p'} V, p$, we have $p' = S(V)$ and $p = D(V)$.
  Well-formedness gives us $D(V) \leq S(V)$.
  By substitution, $p \leq p'$.

***Case* ENDORSE-TYPES** The expression e may not contain endorse expressions so the claim holds vacuously.

***Case* CHECK-TYPES** Here, $e = \mathsf{check}(p, e_c)$.

By inversion on $H; D; S; e \Downarrow_{p'} V$, we have $H; D; S; e_c \Downarrow_{p'} V, p''$, and $p \leq p''$.

By applying the induction hypothesis to $H; D; S; e_c \Downarrow_{p'} V, p''$, we get $p'' \leq p'$.

By transitivity of inequalities, $p \leq p'$.

***Case* TRACK-TYPES** Here, $e = \mathsf{track}(p_t, e_t)$, $\Gamma \vdash e_t : \texttt{@Approx}(p'')$, and $p \leq p''$.

By inversion on $H; D; S; e \Downarrow_{p'} V, p$, we get $H; D; S; e_t \Downarrow_{p'} V$.

By applying the induction hypothesis to $H; D; S; e_t \Downarrow_{p'} V$, we get $p'' \leq p'$.

By transitivity of inequalities, $p \leq p'$.

***Case* SUBSUMPTION** The case where $T = \texttt{@Approx}(p)\ \tau$ applies. There is one rule for subtyping, so we have $\Gamma \vdash e : \texttt{@Approx}(p_s)\ \tau$ where $p_s \geq p$. By induction, $p_s \leq p'$, so $p \leq p'$. □

Finally, we use this preservation lemma for expressions to prove a preservation lemma for statements, completing the main soundness proof.

Lemma 2 (Statement Preservation)**.** *For all programs s with no* endorse *expressions, if $\Gamma \vdash s : \Gamma'$, and $\vdash D, S : \Gamma$, and $H; D; S \longrightarrow H'; D'; S'$, then $\vdash D', S' : \Gamma'$.*

Proof. We induct on the derivation of the statement typing judgment, $\Gamma \vdash s : \Gamma'$.

***Cases* SKIP-TYPES, IF-TYPES, *and* WHILE-TYPES** In these cases, $\Gamma = \Gamma'$, $D = D'$, and $S = S'$, so preservation holds trivially.

***Case* SEQ-TYPES** Here, $s = s_1; s_2$ and the typing judgments for the two component statements are $\Gamma \vdash s_1 : \Gamma_2$ and $\Gamma_2 \vdash s_2 : \Gamma'$. If $s_1 = \textbf{skip}$, then the case is trivial. Otherwise, by inversion on the small step, $H; D; S; s_1 \longrightarrow H'; D'; S'; s'_1$ and, by the induction hypothesis, $\vdash D'_1, S'_1 : \Gamma$.

***Case* DECL-TYPES** The statement s is $Tv := e$ where $\Gamma \vdash e : T$ and $\Gamma' = \Gamma, v : T$. We consider two cases: either $T = \texttt{@Approx}(p)\,\tau$ or $T = \texttt{@Dyn}\,\tau$. In either case, the expression preservation lemma applies.

In the first case, $H; D; S; e \Downarrow_{p'} V$ where $p \leq p'$ via expression preservation and, by inversion, $S' = S, v \mapsto p$ and $D' = D$. Since $S'(v) = p \leq p'$, the well-formedness property $\vdash D, S : \Gamma'$ continues to hold.

In the second case $H; D; S; e \Downarrow_{p'} V, p_d$ where $p_d \leq p'$. By inversion, $S' = S, v \mapsto p$ and $D' = D, v \mapsto p_d$. Since $D'(v) = p_d \leq p'$, we again have $\vdash D, S : \Gamma'$.

***Case* MUTATE-TYPES** The case where s is $v := e$ proceeds similarly to the above case for declarations. □

A.4.2 Erasure of Probability Bookkeeping

We state (and sketch a proof for) an *erasure* property that shows that the "true" probabilities in our semantics, called S, do not affect execution. This property emphasizes that S is bookkeeping for the purpose of stating our soundness result—it corresponds to no run-time data. Intuitively, the theorem states that the steps taken in our dynamic semantics are insensitive to S: that S has no effect on which H', D', or s' can be produced.

In this statement, $\mathrm{Dom}(S)$ denotes the set of variables in the mapping S.

Theorem 2 (Bookkeeping Erasure)**.** *If* $H; D; S_1; s \longrightarrow^n H'; D'; S_1'; s'$*, then for any probability map* S_2 *for which* $\mathrm{Dom}(S_1) = \mathrm{Dom}(S_2)$*, there exists another map* S_2' *such that* $H; D; S_2; s \longrightarrow^n H'; D'; S_2'; s'$*.*

Proof sketch. The intuition for the erasure property is that no rule in the semantics uses $S(v)$ for anything other than producing a probability in the $\Downarrow_p$ judgment, and that those probabilities are only ever stored back into S.

The proof proceeds by inducting on the number of steps, n. The base case ($n = 0$) is trivial; for the inductive case, the goal is to show that a single step preserves H', D', and s' when the left-hand probability map S is replaced. Two lemmas show that replacing S with S' in the expression judgments leads to the same result value V and, in the dynamically-tracked case, the same tracking probability p_d. Finally, structural induction on the small-step statement judgment shows that, in every rule, the expression probability only affects S itself.