

Probability Type Inference for Flexible Approximate Programming

by

Brett Boston

Supervised by Dan Grossman

A senior thesis submitted in partial fulfillment of
the requirements for the degree of

Bachelor of Science
With Departmental Honors

Computer Science & Engineering

University of Washington

June 2015

Presentation of work given on 05/15/2015

Thesis and presentation approved by



Date 06/04/2015

Abstract

With approximate computing, programs can save energy or increase performance by allowing occasional mistakes. However, writing approximate programs at a granular level is challenging. We propose a programming language that allows the programmer to place probabilistic reliability bounds on key variables in a program. Our language, DECAF, then performs type inference using an SMT solver to fill in the reliabilities for other approximate variables and operators. Optional dynamic tracking allows the programmer to use approximation in situations where DECAF cannot statically reason about reliabilities. Together, these features empower the programmer to leverage the use approximation without the tremendous effort of manually annotating an entire program.

DECAF is evaluated with existing approximate computing benchmarks to demonstrate the low annotation overhead of our system. We aim to inform hardware design by allowing the user to input hardware parameters to the system to compare the effects of various hardware configurations. We find that architectures with more than two degrees of operator precision can offer significant advantages over simpler two-level architectures. Additionally, we find that solving type constraints with access to hardware parameters improves efficiency over assuming a continuous model and rounding to supported precision levels at runtime.

Contents

1	Introduction	5
2	Motivation	5
3	Probabilistic Types	5
3.1	Subtyping	5
3.2	Operators	6
3.3	Control Flow	6
3.4	Dependence	6
3.5	Arrays	7
4	Probability Type Inference	7
4.1	Constraint Generation	7
4.2	Objective Function	9
4.3	Extracting Values From Z3	9
4.4	Method Specialization	10
4.4.1	Limiting Method Checks	11
4.5	Name Mangling	12
4.6	Conservative Independent Substitution for Dependent Values	12
5	Dynamic Tracking	13
6	Programming Model	13
6.1	Warnings	13
7	Hardware Model	13
7.1	Hardware Simulation	13
7.2	Discrete Precision Levels	14
8	Formalism	14
8.1	Syntax	14
8.2	Typing	15
8.2.1	Operator Typing	15
8.2.2	Other Expressions	15
8.2.3	Qualifiers and Subtyping	15
8.2.4	Statement Typing	16
8.3	Operational Semantics	16
8.3.1	Expression Semantics	16
8.3.2	Statement Semantics	17
8.4	Theorems	18
8.4.1	Soundness	18
8.4.2	Erasure of Probability Bookkeeping	20
9	Implementation	21
9.1	Managing Solver Time	21
9.2	“Separate” Compilation	21
10	Evaluation	22
10.1	Benchmarks	22
10.2	Solving Versus Rounding Discrete Precision Levels	22
10.3	Granularity of Discrete Levels	23
10.4	Compilation Time	23

11 Future Work	24
11.1 Error Messages	24
11.2 Modularity	24
12 Conclusion	24
13 Acknowledgements	24

1 Introduction

Approximate computing is the notion that not every operation in a program must be completely error-free. Allowing this relaxation may result in energy savings or performance increases [5, 10].

Approximation in computing is nothing new, showing up often where significant benefits can be seen from an imperceptible reduction in quality. One example of this is lossy audio compression. The MPEG/audio compression algorithm employs a “virtual ear” to determine what humans can and cannot hear in an audio file. Sounds that are determined to be inaudible or indistinguishable from other sounds are then thrown out. This process significantly reduces file sizes below what is possible with lossless compression [6].

However, this style of approximation requires intimate knowledge of the problem space as well as how to achieve the desired level of approximation. Given that architectures exist that can execute arithmetic instructions approximately [4], we designed a programming language to facilitate approximate programming using these instructions.

Our system is built on top of EnerJ [7] which allows type annotations on variables to denote that the value in the variable may be approximated. To indicate that a variable is to be approximated the programmer annotates the variable with `@Approx`. An unqualified variable is given the default qualifier (`@Precise`) indicating that a value may not be approximated. Precise values may flow into approximate variables, but the opposite is forbidden. Approximate values may be statically cast into precise values using the `endorse` keyword. For the purposes of this work we focus only on EnerJ’s ability to approximate operations, ignoring approximate storage. A simple EnerJ program can be seen below:

```
@Approx int a = 1;
@Approx int b = 2;
@Approx int c = a + b; // + is approximate
@Precise int p; // @Precise is unnecessary
p = c; // Illegal
p = endorse(c); // Cast c to a @Precise int
```

2 Motivation

EnerJ’s annotations allow for binary specification of values as either approximate or not, but architectures have been suggested that give the programmer access to multiple levels of approximation [9]. At an instruction level, each operator has a probability of returning the correct answer. With this type of system the programmer needs fine control over every operation in a program. However, this style of programming quickly becomes unusable as it is very difficult to come up with appropriate probabilities for every single operation in a program. We would like to give the programmer control when they want it, and leave the rest to the compiler.

To facilitate approximate programming in this context we propose DECAF (**DECAF**, an **E**nergy-aware **C**ompiler to make **A**pproximation **F**lexible). DECAF frames this problem as one of type inference and uses an SMT solver to infer continuous probability types.

3 Probabilistic Types

DECAF introduces the parameterized `@Approx(n)` qualifier. At any point in the execution, the probability that the value is *correct* is at least n where *correct* is defined as the value being the same as it would be during fully precise execution. DECAF offers no guarantees about an incorrect value, it may be anything.

3.1 Subtyping

DECAF preserves soundness by permitting data flow from values with high reliability to variables with lower reliability while preventing low-to-high flow:

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

```

y = x; // sound
x = y; // error

```

More concretely, we define a subtyping rule such that a type is a subtype of other types with lower probability:

$$\frac{\text{SUBTYPING} \quad p \geq p'}{\text{@Approx}(p) \tau \prec \text{@Approx}(p') \tau}$$

The full formalism for DECAF will be discussed later in Section 8.

3.2 Operators

All approximation in DECAF comes from arithmetic operators. Every operator in a DECAF program has a probability of succeeding associated with it. In the failure case, DECAF offers no guarantees about the result of the operation. When a binary operator is used, the result of the operation has a probability of correctness equal to the product of the reliabilities of the two operands and the reliability of the operator. This follows from the product rule from probability theory:

$$P(A \cap B) \geq P(A) \cdot P(B)$$

The following example demonstrates the uses of binary operators:

```

1 @Approx(0.9) int x = 1;
2 @Approx(0.9) int y = 2;
3 @Approx(0.81) int a = x + y; \\ Precise +
4 @Approx(0.70) int b = x + y; \\ + may be approximated

```

The sum on line 3 allows for no approximation as the product of the reliabilities of `x` and `y` is already 0.81 so any operator reliability under 1.0 would bring the reliability of the right hand side of the assignment below the constraint on `a`. However, the same sum on line 4 may be approximated as the product of the reliabilities of the values on the right hand side of the assignment is strictly greater than the reliability of the variable it is being assigned into.

3.3 Control Flow

Any branch in control flow must have a test that is precise. This forces the programmer to make a conscious decision to branch on approximate values using an endorsement.

For example, take a program that would like to display a friendly greeting to the user if it is before noon:

```

1 int seconds = getTimeInSeconds();
2 String greeting = "";
3 @Approx(0.9) int hour = seconds / 3600;
4
5 if (endorse(hour) < 12)
6     greeting = "Good Morning!";

```

However, the programmer has decided that this greeting is not always necessary and has chosen to branch on the result of the hour conversion on line 3 by using an endorsement to cast to a precise value.

The `endorse` keyword is an unchecked static cast, and therefore introduces unsoundness. Alternatively, the `check` keyword described in Section 5 is a sound, dynamic cast.

3.4 Dependence

DECAF conservatively treats all values as independent. An example of this can be seen below where the assignment on line 2 fails even though it technically satisfies the annotation:

```

1 @Approx(0.9) int x = ...;
2 @Approx(0.9) int xSquared = x * x; // illegal
3 @Approx(0.81) int xSquared = x * x; // legal

```

While this particular case may be easy to handle, there are many cases where it is tough to know how values depend on each other and it is unclear how to treat dependence in probability calculations as many values may flow into variables. The proof that this transformation is conservative can be found in Section 4.6.

3.5 Arrays

In DECAF annotations on arrays apply to each element independently. That is, the code below will produce an array containing roughly 95 twos:

```
@Approx(0.95) int[] twos = new @Approx(0.95) int[100];

for (int i = 0; i < 100; ++i)
    twos[i] = 1 + 1;
```

4 Probability Type Inference

The system as described so far is very difficult to use. In many cases the programmer knows how reliable to make inputs and outputs, but finding reliabilities for intermediate variables in the program innards is challenging. To make programming with probability types easier we introduced the unparameterized `@Approx` annotation. By leaving off the parameter to the `@Approx(n)` annotation the programmer indicates that they would like `n` to be inferred. DECAF will infer `n` such that the variable’s reliability satisfies the reliability of the variables it flows into as well as the variables that flow into it. This new annotation greatly eases the annotation burden of using DECAF.

An example of this can be seen in the code approximating the area of a triangle:

```
@Approx float base = ...;
@Approx float height = ...;
@Approx(0.9) float area = base * height / 2;
```

While the programmer would like the final result to be correct at least 90% of the time, they do not care about the reliability of `base` or `height` so long as they satisfy the constraint placed on `area`.

At a high level the inference process works as shown in Figure 1. The programmer’s annotations, information about hardware, and an objective function that is an abstraction of energy usage is fed into Microsoft’s Z3 SMT solver [3]. If this set of constraints is over-constrained then the program contains a type error. That is, there exists no set of reliabilities for inferred annotations that satisfy the explicit `@Approx(n)` constraints. If the system is under-constrained then we have a correct solution for the inferred annotations. At this point DECAF will attempt to lower the objective target, thus lowering energy usage. This process continues until the objective target cannot be lowered further.

4.1 Constraint Generation

To facilitate inference over continuous types, we made use of Microsoft’s Z3 SMT solver. Translation from type qualifiers to constraints works as follows:

Explicit `@Approx(n)` Annotations Variables with the `@Approx(n)` annotation are bound to be equal to `n`.

Inferred `@Approx` Annotations Variables with the `@Approx` annotation are bound to be between 0.0 and 1.0 inclusive.

Operators Operators are treated like variables with inferred `@Approx` annotations.

Assignment The assignment `x = y` generates a constraint that asserts that the reliability of `x` is less than or equal to the reliability of `y`.

An example of this translation can be seen in Figure 2. Here, the DECAF code on the left is translated to the set of constraints on the right. `x`, `y`, and `op1` are all bound between 0.0 and 1.0 to be inferred. `z` has an explicitly declared reliability and is therefore bound to be equal to 0.81 and less than or equal to `x * y * op1`.

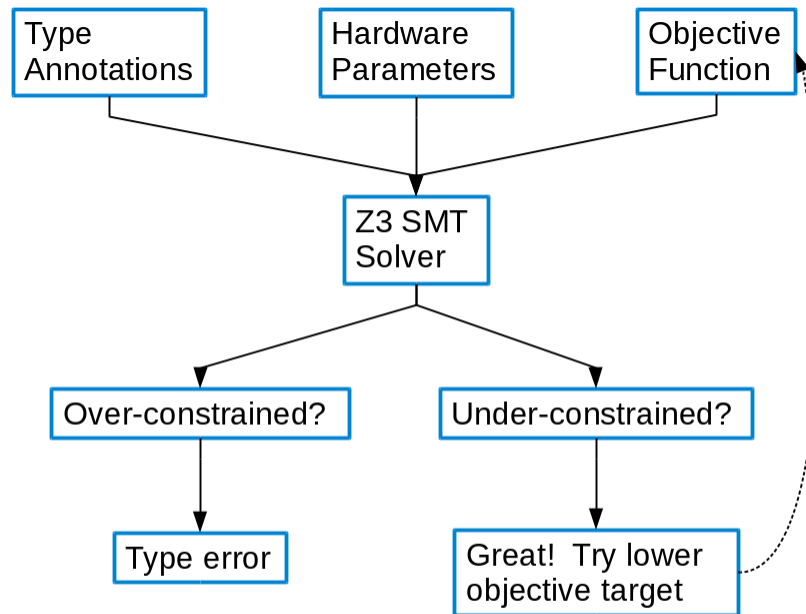


Figure 1: Inference process

<pre> @Approx int x = 1; @Approx int y = 2; @Approx(0.81) int z = x + y; </pre>	<pre> (declare-const op1 Real) (assert (>= op1 0.0)) (assert (<= op1 1.0)) (declare-const x Real) (assert (>= x 0.0)) (assert (<= x 1.0)) (declare-const y Real) (assert (>= y 0.0)) (assert (<= y 1.0)) (declare-const z Real) (assert (= z 0.81)) (assert (<= z (* x y op1))) </pre>
---	--

Figure 2: Sample DECAF to Z3 constraint translation

4.2 Objective Function

While the process in Section 4.1 will produce a solution that satisfies the constraints, there may be infinitely many solutions. Take Figure 2 for example. One naive solution with no approximation is $x = y = 0.9$, $op1 = 1.0$. By default, Z3 likes $x = \frac{15}{16}$, $y = \frac{127}{128}$, $op1 = \frac{7}{8}$. However, the optimal solution for maximizing approximation is $x = y = 1.0$, $op1 = 0.81$.

To drive Z3 towards solutions that maximize approximation we employ an objective function. This function averages the inferred probabilities across a function, targeting a specific average precision. We then approach an optimal result using a linear search, lowering target average precision by a constant amount until the problem is unsatisfiable or times out.

For example, take the following statement:

```
@Approx(0.81) int z = a + b + c;
```

The constraints generated when minimizing the objective target for this statement are:

```
(declare-const obj-target Real)
(assert (= obj-target (/ (+ op1 op2) 2)))

(assert (<= obj-target 1.0))
(check-sat)
sat

(push)
(assert (<= obj-target 0.99))
(check-sat)
sat

...

(push)
(assert (<= obj-target p))
(check-sat)
unsat
(pop)
```

First, the target is set to be equal to the average of the two operators. The `(push)` and `(pop)` commands tell Z3 to push/pop its internal stack of constraints. Thus, after over-constraining the objective target we can get back to a near-optimal satisfiable set of constraints by popping the previous constraints off the stack, seen here after asserting the objective target to be less than or equal to some probability p .

4.3 Extracting Values From Z3

After a successful `check-sat`, the values Z3 used to determine satisfiability may be extracted using the `get-value` command. This command is used after successfully checking methods to record the inferred reliabilities of variables and operators.

The result of `get-value` can come in two forms. First, Z3 may return an easily parsable decimal:

```
(get-value (x))
((x 1.0))
```

Alternatively, Z3 may return a fraction that must then be translated into a float:

```
(get-value (ret))
((x (/ 13273.0 16384.0))
```

This is due to the fact that Z3 uses precise real numbers internally over floating point numbers.

```

1 void example() {
2     @Approx(0.9) float area1 = triArea(1, 2);
3     @Approx(0.95) float area2 = triArea(1, 3);
4 }
5
6 @Approx float triArea(@Approx float b, @Approx float h) {
7     @Approx float c = b * h / 2;
8     return c;
9 }

```

Figure 3: Inference may be applied in method signatures as seen in this approximate triangle area calculator.

4.4 Method Specialization

In addition to being used on local variables the inferred `@Approx` annotation may be used in method signatures. When invoking a method with an inferred `@Approx` annotation in its signature a specialized version of the method will be generated to match the call site. This process is similar to function generation for C++ templates.

For example, take the implementation of an approximate triangle area calculator in Figure 3. Due to the different return precisions, two different versions of `triArea` will be synthesized to satisfy the invocations on lines 2 and 3.

DECAF’s method specialization is interprocedural. When an invocation that must be specialized is encountered the type checker jumps into the invocation and rechecks it, effectively inlining the function. As a result, cycles in the call structure consisting solely of specialized methods is not allowed as the compiler cannot reason about them. While jumping into invocations an internal stack of the call structure is maintained to detect cycles and issue errors when found.

To generate the appropriate constraints for inference across methods, constraints are emitted as previously described, but extra constraints are emitted at the end of a function call to bind parameters and return values. Formal parameters must have reliabilities less than or equal to actual parameters and the returned expression’s reliability must be at least the reliability in the method’s return type. An example of this can be seen below where the DECAF code:

```

void caller() {
    @Approx int a = 1;
    @Approx int b = 2;
    @Approx(0.9) int c = add(1,2);
}

@Approx int add(@Approx int x, @Approx int y) {
    return x + y;
}

```

results in the following constraints being emitted where `method-add` is the return value of `add`:

```

; Initialize caller’s local variables
(declare-const a Real)
(assert (>= a 0.0))
(assert (<= a 1.0))
(declare-const b Real)
(assert (>= b 0.0))
(assert (<= b 1.0))
(declare-const c Real)
(assert (= c 0.9))

; Jump into add

```

```

; Initialize add's formal parameters
(declare-const x Real)
(assert (>= x 0.0))
(assert (<= x 1.0))
(declare-const y Real)
(assert (>= y 0.0))
(assert (<= y 1.0))

; Initialize operator
(declare-const op1 Real)
(assert (>= op1 0.0))
(assert (<= op1 1.0))

; Initialize add's return value
(declare-const method-add Real)
(assert (>= method-add 0.0))
(assert (<= method-add 1.0))

; Assign return value
(= method-add (* x y op1))

; Bind formal parameters and actual parameters
(assert (<= x a))
(assert (<= y b))

; Bind return value to c in caller
(assert (<= c method-add))

```

4.4.1 Limiting Method Checks

Inlining every single invocation can quickly increase code size as well as compile time, so the programmer may provide an optional parameter at compile time to place an upper bound on the total number of times any given method may be specialized. DECAF will then try to find a set of specializations that satisfy all invocations even when there are more invocations than specializations.

To find this set of specializations DECAF emits a set of constraints that emulates a two-dimensional array where each column represents a method invocation and each row represents a generated version. Each element may be a zero or one where a one signifies that an invocation is bound to a version. Correctness requires exactly one one in every column. That is, every invocation must be bound to exactly one version. To enforce that methods are maximally specialized we require that every row contain at least one one. That is, every generated version must have at least one invocation associated with it. Without this provision, Z3 will “cheat” by minimizing the number of versions bound to invocations so it can drive the operator precision in the unbound methods to zero for a better objective score.

An example array representing five invocations of a function capped at three method generations is below:

		Invocation				
		0	1	2	3	4
Version	0	1	0	0	1	0
	1	0	1	0	0	1
	2	0	0	1	0	0

Z3 technically has its own built in arrays but using them disables the real solver engine, which is why we chose to emulate arrays with specially named variables instead.

4.5 Name Mangling

With our interprocedural analyses it is very possible to have variable name collisions across functions. To mitigate this, variable names are mangled in a deterministic way when declared to Z3. It is important that this process is repeatable for the instrumentation step described in Section 7.1.

Variable names are mangled by appending the sum of a global counter multiplied by 1000000 and the number of variables with the same name in the call stack of inlined functions. For example, in Figure 3 from earlier `c` from both calls is mangled as `c-10000000` and `c-22000000` for each call. If `example` contained a variable named `c`, the two mangled names would have ones in the ones place.

Method return values are more complicated, following the name scheme:

```
method-<classname>-<methodname>-<caller line number>--<varnum>
```

where `varnum` is follows the number scheme for variables. If limited method checks are enabled formal parameters are concatenated with the method return name.

Lastly, operators must be mangled:

```
op-method-<classname>-<top methodname>-<caller line number>-<method #>--<id>-<varnum>
```

Top `methodname` represents the topmost method that methods are being inlined into. `method #` is a unique number assigned to each invocation. This is necessary to bind the operator to the correct invocation. `id` is a unique identifier assigned by the Java compiler to each operator. `varnum` is the same counter used for methods and variables. Not all of this information is totally necessary for inferring operators, but the names are built up from sources for which the information is necessary. This reuse simplifies the code base.

4.6 Conservative Independent Substitution for Dependent Values

Imagine we have the following scenario:

```
@Approx(n) int A, X;  
@Approx int B = A + X;  
@Approx int C = A + B;
```

`A` and `X` have declared probabilities but are independent of one another while `B` and `C` have inferred probabilities. The probability of correctness of `B` depends on `A` and `X`. With this in mind, how do we calculate the probability of correctness of `C`? From joint probability theory we know that this is simply

$$P(C) = P(A) \times P(B|A)$$

However, we don't always know $P(B|A)$ so it would be nice substitute $P(B)$. In order for this to work the substitution must be conservative, which is to say

$$P(C) \leq P(A) \times P(B)$$

To demonstrate that this is a conservative substitution we prove that

$$P(B) \leq P(B|A)$$

Our proof begins with the theory of total probability which states that

$$P(B) = P(A) \times P(B|A) + P(\neg A) \times P(B|\neg A) \tag{1}$$

Since it is impossible for `B` to be correct given that `A` is incorrect at the time of assignment we know that $P(B|\neg A) = 0$. Thus we can simplify as follows

$$P(B) = P(A) \times P(B|A) \tag{2}$$

Since $P(A)$ is a probability and by definition between 0 and 1 inclusive, dropping it from the right hand side will either increase its value or keep it the same. This action cannot decrease the value of the right hand side. So, we have

$$P(B) \leq P(B|A) \tag{3}$$

Therefore it is conservative to substitute $P(B)$ for $P(B|A)$.

5 Dynamic Tracking

Reasoning about error statically means that DECAF thus far cannot handle situations where a variable is modified as a function of itself. That is, the following running sum is illegal because the `sum` variable feeds into itself decreasing its reliability each time:

```
@Approx(0.99) int[] nums = getNums();
@Approx(0.9) int sum = 0;
for (int num : nums)
    sum += num;
```

To facilitate this case DECAF offers a `@Dyn` annotation that causes a value's reliability to be tracked dynamically at runtime:

```
@Approx(0.99) int[] nums = getNums();
@Dyn int sum = 0;
for (int num : nums)
    sum += num;
@Approx(0.9) int approxSum = check(sum, 0.9);
```

In this adaptation of the previous example, the running sum will have its reliability tracked at runtime. To cast from a `@Dyn` type to a static `@Approx` type the `check` keyword is used. This keyword will perform a runtime check to ensure that `sum`'s reliability is greater than or equal to 0.9. If this check fails an unchecked `PrecisionException` is thrown.

6 Programming Model

The programming model for DECAF is to annotate program inputs and outputs with explicit `@Approx(n)` qualifiers and to use inferred `@Approx` annotations on the innards. This model leverages programmer knowledge about the problem space while freeing them from the burden of figuring out the proper constraints to reach an end goal. Inputs may be approximate to model reads from a noisy sensor while outputs are chosen based on the level of approximation the programmer would accept.

6.1 Warnings

DECAF also introduces warnings that provide insight into the inference system, which can be confusing for large programs.

A warning fires if a variable *could* have a reliability of 0.0 while still satisfying type constraints. This indicates that the variable never flows into any constrained variable. In this case DECAF offers no guarantees about the variable and the compiler could theoretically optimize it out. The warning often indicates programmer error or dead code.

Another warning fires if a variable *must* have a reliability of 1.0. This indicates that a variable marked for approximation was constrained such that none is possible. To eliminate the warning values flowing into this variable must be relaxed. This is not always an error, but this warning helps the developer who is seeing less approximation than expected due to over constraining portions of their code.

7 Hardware Model

7.1 Hardware Simulation

After inference is completed, DECAF exports operator precision for each operator in the program. This data is then read in by the instrumentation pass. Instrumentation walks the abstract syntax tree in the same fashion as the inference compiler pass. This allows mangled variable names to be regenerated in the same way as described in Section 4.5.

As approximate binary operators are found, they are replaced with function calls in the simulator. These functions will perform the operation, failing randomly according to the reliability of the operator. The precision levels of each operation used at runtime are recorded and exported for benchmarking purposes.

Additionally, `@Dyn` values are wrapped in objects with an extra field representing their dynamic reliability. These fields are initialized to 1.0 and modified through assignment by replacing the assignment operator with a function call.

7.2 Discrete Precision Levels

Although DECAF works with continuous probability types, most hardware will realistically not have continuous knobs for operator precision. In reality hardware will likely support a set of discrete reliabilities, similar to the QUORA approximate architecture [9]. To accommodate this, DECAF allows the programmer to specify discrete levels at compile time, or runtime.

If provided at runtime, operator reliabilities will simply be rounded up to the next supported precision level.

However, if discrete levels are supplied at runtime they will be taken into account during constraint generation. Rather than binding operators to between 0.0 and 1.0, they will be bound to one of the available precision levels. For example, on a system supporting the levels 0.9, 0.99, and 1.0 the following constraints would be generated:

```
(declare-const op Real)
(assert (or (= op 0.9)
            (= op 0.99)
            (= op 1.0)))
```

Supplying discrete levels at compile time offers significantly more approximation than rounding at runtime. This is likely because Z3 does not try to push some operators below supported levels at the expense of other operators.

8 Formalism

Here we formalize DECAF with the intention of proving a soundness theorem that captures the probability type system's accuracy guarantee.

8.1 Syntax

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

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

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.

8.2 Typing

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

8.2.1 Operator Typing

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

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

$$\text{optype}(\tau, \tau) = \tau$$

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

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

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 (the compiler inserts `track` casts to introduce dynamic tracking when necessary).

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

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

$$\frac{\text{OP-DYN-TYPES} \quad \Gamma \vdash e_1 : @\text{Dyn} \tau_1 \quad \Gamma \vdash e_2 : @\text{Dyn} \tau_2 \quad \tau_3 = \text{optype}(\tau_1, \tau_2)}{\Gamma \vdash e_1 \oplus_p e_2 : @\text{Dyn} \tau_3}$$

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 3.2 gives the probabilistic intuition behind this rule.

8.2.2 Other Expressions

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

$$\frac{\text{CONST-INT-TYPES} \quad c \text{ is an integer}}{\Gamma \vdash c : @\text{Approx}(1.0) \text{int}}$$

$$\frac{\text{CONST-FLOAT-TYPES} \quad c \text{ is not an integer}}{\Gamma \vdash c : @\text{Approx}(1.0) \text{float}}$$

$$\frac{\text{VAR-TYPES} \quad T = \Gamma(v)}{\Gamma \vdash v : T}$$

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 section, will exclude the latter from the language.) Similarly, `track` casts produce a dynamically-tracked type given a statically-tracked counterpart.

$$\frac{\text{ENDORSE-TYPES} \quad \Gamma \vdash e : q \tau}{\Gamma \vdash \text{endorse}(p, e) : @\text{Approx}(p) \tau}$$

$$\frac{\text{CHECK-TYPES} \quad \Gamma \vdash e : @\text{Dyn} \tau}{\Gamma \vdash \text{check}(p, e) : @\text{Approx}(p) \tau}$$

$$\frac{\text{TRACK-TYPES} \quad \Gamma \vdash e : @\text{Approx}(p') \tau \quad p \leq p'}{\Gamma \vdash \text{track}(p, e) : @\text{Dyn} \tau}$$

8.2.3 Qualifiers and Subtyping

A simple subtyping relation, introduced in Section 3.2, makes high-probability types subtypes of their low-probability counterparts.

$$\boxed{T_1 \prec T_2}$$

$$\frac{\text{SUBTYPING} \quad p \geq p'}{@\text{Approx}(p) \tau \prec @\text{Approx}(p') \tau}$$

Subtyping uses a standard subsumption rule.

$$\frac{\text{SUBSUMPTION} \quad T_1 \prec T_2 \quad \Gamma \vdash e : T_1}{\Gamma \vdash e : T_2}$$

8.2.4 Statement Typing

Our typing judgment for statements builds up the context Γ .

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

$$\frac{\text{SKIP-TYPES}}{\Gamma \vdash \text{skip} : \Gamma}$$

$$\frac{\text{SEQ-TYPES} \quad \Gamma_1 \vdash s_1 : \Gamma_2 \quad \Gamma_2 \vdash s_2 : \Gamma_3}{\Gamma_1 \vdash s_1; s_2 : \Gamma_3}$$

$$\frac{\text{DECL-TYPES} \quad \Gamma \vdash e : T \quad v \notin \Gamma}{\Gamma \vdash T v := e : \Gamma, v : T}$$

$$\frac{\text{MUTATE-TYPES} \quad \Gamma \vdash e : T \quad \Gamma(v) = T}{\Gamma \vdash v := e : \Gamma}$$

$$\frac{\text{IF-TYPES} \quad \Gamma \vdash e : \text{@Approx}(1.0) \tau \quad \Gamma \vdash s_1 : \Gamma_1 \quad \Gamma \vdash s_2 : \Gamma_2}{\Gamma \vdash \text{if } e \text{ } s_1 \text{ } s_2 : \Gamma}$$

$$\frac{\text{WHILE-TYPES} \quad \Gamma \vdash e : \text{@Approx}(1.0) \tau \quad \Gamma \vdash s : \Gamma'}{\Gamma \vdash \text{while } e \text{ } s : \Gamma}$$

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

8.3 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.

8.3.1 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.

$$\frac{\text{CONST}}{H; D; S; c \Downarrow_{1.0} c}$$

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 .

$$\frac{\text{VAR} \quad v \notin D}{H; D; S; v \Downarrow_{S(v)} H(v)} \quad \frac{\text{VAR-DYN} \quad v \in D}{H; D; S; v \Downarrow_{S(v)} H(v), D(v)}$$

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

$$\frac{\text{ENDORSE} \quad H; D; S; e \Downarrow_p V}{H; D; S; \text{endorse}(p_e, e) \Downarrow_p V}$$

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.)

$$\frac{\text{CHECK} \quad H; D; S; e \Downarrow_p V, p_1 \quad p_1 \geq p_2}{H; D; S; \text{check}(p_2, e) \Downarrow_p V}$$

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.

$$\frac{\text{TRACK} \quad H; D; S; e \Downarrow_p V}{H; D; S; \text{track}(p_d, e) \Downarrow_p V, p_d}$$

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:

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

$$\frac{\text{OP-OPERATOR-INCORRECT} \quad H; D; S; e_1 \Downarrow_{p_1} c_1 \quad H; D; S; e_2 \Downarrow_{p_2} c_2 \quad p = p_1 \cdot p_2 \cdot p_{\text{op}}}{H; D; S; e_1 \oplus_{p_{\text{op}}} e_2 \Downarrow_p \square}$$

$$\frac{\text{OP-OPERANDS-INCORRECT} \quad H; D; S; e_1 \Downarrow_{p_1} \square \text{ or } H; D; S; e_2 \Downarrow_{p_2} \square \quad p = p_1 \cdot p_2 \cdot p_{\text{op}}}{H; D; S; e_1 \oplus_{p_{\text{op}}} e_2 \Downarrow_p \square}$$

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

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

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

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

8.3.2 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 .

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

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

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

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

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

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

Sequencing Sequencing is standard and deterministic.

$$\frac{\text{SEQ-SKIP}}{H; D; S; \mathbf{skip}; s \longrightarrow H; D; S; s}$$

$$\frac{\text{SEQ} \quad H; D; S; s_1 \longrightarrow H'; D'; S'; s'_1}{H; D; S; s_1; s_2 \longrightarrow H'; D'; S'; s'_1; s_2}$$

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

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

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

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

8.4 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.

8.4.1 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 $\text{@Approx}(p)$ 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 $\Gamma(v) = \text{@Approx}(p) \tau$, then $p \leq S(v)$ or $v \notin S$.
- If $\Gamma(v) = \text{@Dyn} \tau$, 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 $T = \textcircled{\text{A}}\text{pprox}(p) \tau$, and $H; D; S; e \Downarrow_{p'} V$, then $p \leq p'$.*
- *If $T = \textcircled{\text{D}}\text{yn} \tau$, 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, $e = e_1 \oplus_{p_{\text{op}}} e_2$ and $T = \textcircled{\text{A}}\text{pprox}(p) \tau$. We also have types for the operands: $\Gamma \vdash e_1 : \textcircled{\text{A}}\text{pprox}(p_1) \tau_1$ and $\Gamma \vdash e_2 : \textcircled{\text{A}}\text{pprox}(p_2) \tau_2$.

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, $e = e_1 \oplus_{p_{\text{op}}} e_2$ and $T = \textcircled{\text{D}}\text{yn} \tau$, and the operand types are $\Gamma \vdash e_1 : \textcircled{\text{D}}\text{yn} \tau_1$ and $\Gamma \vdash e_2 : \textcircled{\text{D}}\text{yn} \tau_2$.

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, $\Gamma \vdash e : \textcircled{\text{A}}\text{pprox}(p) \tau$ 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 $T = \textcircled{\text{A}}\text{pprox}(p) \tau$: 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 $T = \textcircled{\text{D}}\text{yn} \tau$: 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, $e = \text{check}(p, e_c)$.

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, $e = \text{track}(p_t, e_t)$, $\Gamma \vdash e_t : \text{CApprox}(p'')$, 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 $T = \text{CApprox}(p) \tau$ applies. There is one rule for subtyping, so we have $\Gamma \vdash e : \text{CApprox}(p_s) \tau$ where $p_s \geq p$. By induction, $p_s \leq p'$, so $p \leq p'$. \square

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 $s_1 = \text{skip}$, 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 $T = \text{CApprox}(p) \tau$ or $T = \text{CDyn} \tau$. 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. \square

8.4.2 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, $\text{Dom}(S)$ 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 $\text{Dom}(S_1) = \text{Dom}(S_2)$, 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

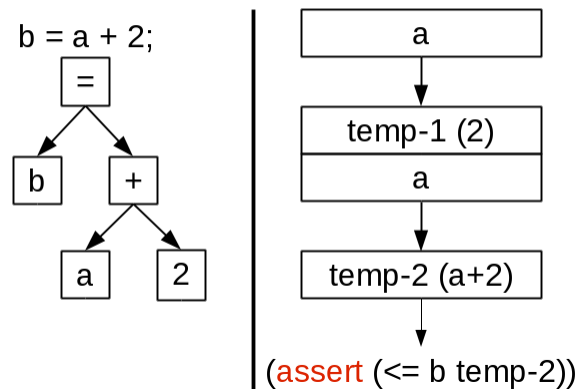
result value V and, in the dynamically-tracked case, the same tracking probability p_d . Finally, structural induction on the small-step statement judgment shows that, in every rule, the expression probability only affects S itself.

9 Implementation

Constraints are generated on a compiler pass while walking the program’s abstract syntax tree. Variables are declared to Z3 as they are visited in the AST.

A stack of variables is maintained internally. Each time a variable is used it is pushed on the stack. Binary operators pop the first two elements off the stack, generate the appropriate constraint, and assign the result to a new temporary variable that is then pushed on the stack. This allows us to chain binary operations. Finally, assignment nodes pop the top of the stack and assign the left hand side to the popped value.

For example, the assignment $b = a + 2$ translates to the AST on the left:



The right hand side shows how the stack changes as the AST is walked, resulting in the final assignment constraint being emitted at the end.

9.1 Managing Solver Time

In most cases, checking satisfiability is a quick operation. However, it can take very long time in other cases. To mitigate this, we offer configurable soft and hard timeouts on any given call to `check-sat`. The programmer may need to play with these numbers a bit. In general we found better results with more time per `check-sat`, but diminishing returns with long timeouts.

When a timeout fires the current `check-sat` is terminated and treated as if the result was unsatisfiable. Because of this default, we do not use timeouts for the purpose of type checking; they are only used for solving objective functions. Thus we never reject valid programs based on the length of solver time. In practice, the type checking `check-sat` calls almost never take a long time whereas minimizing the objective function can take quite a while as the target gets close to the optimal target.

A soft timeout sends a SIGINT to Z3 which (in most cases) causes it to drop the current `check-sat`. However, Z3 may ignore the SIGINT resulting in the hard timeout firing. In this case, Z3 is programmatically killed and restarted. A set of stored constraints are re-issued minus all satisfiability checks and the offending constraints.

9.2 “Separate” Compilation

Due to the interprocedural nature of DECAF’s analyses, we cannot support separate compilation. That is, DECAF needs to be able to jump into other compilation units often to generate specialized methods.

To mitigate this, we wrote `jcat` [1]. `Jcat` is a special tool for concatenating Java files. It treats the first file as the output file’s public class (Java files may only contain one) and strips the public keyword from all other

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).

classes. It also moves imports to the top the output file. Lastly, jcat can detect attempts to concatenate files from multiple packages, which almost certainly will not produce a valid output file.

10 Evaluation

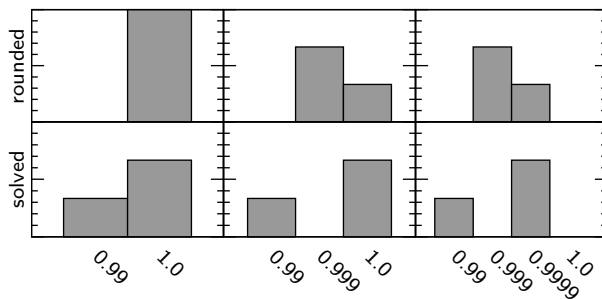
DECAF was evaluated by simulating hardware that supports energy saving approximate operators. Savings are measured through the percentage of operators that were approximated given that they *could* have been approximated.

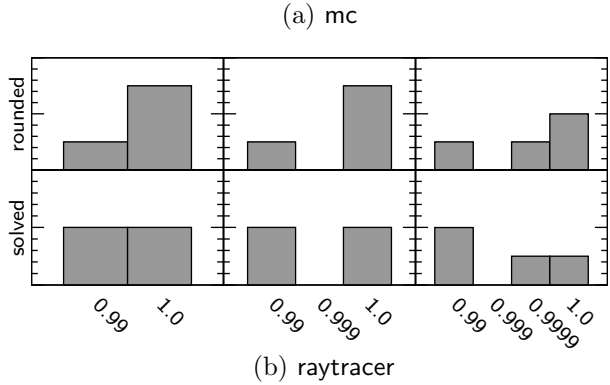
10.1 Benchmarks

Our benchmarks were drawn from a set of benchmarks known to be amenable to approximate execution [7, 8]. These benchmarks came with a set of @Approx annotations from EnerJ that we extended to support new features. The annotation process consisted primarily of placing @Approx(0.9) annotations on program outputs while using inferred @Approx annotations on the innards. @Dyn annotations were used as sparingly as possible. More information about the benchmarks and annotations can be seen in Table 1.

10.2 Solving Versus Rounding Discrete Precision Levels

Although hardware with continuous precision knobs would be ideal, real world hardware will likely support discrete levels of precision. DECAF allows the programmer to supply a set of supported discrete levels at either compile time, or runtime. The effect of this choice can be seen below:

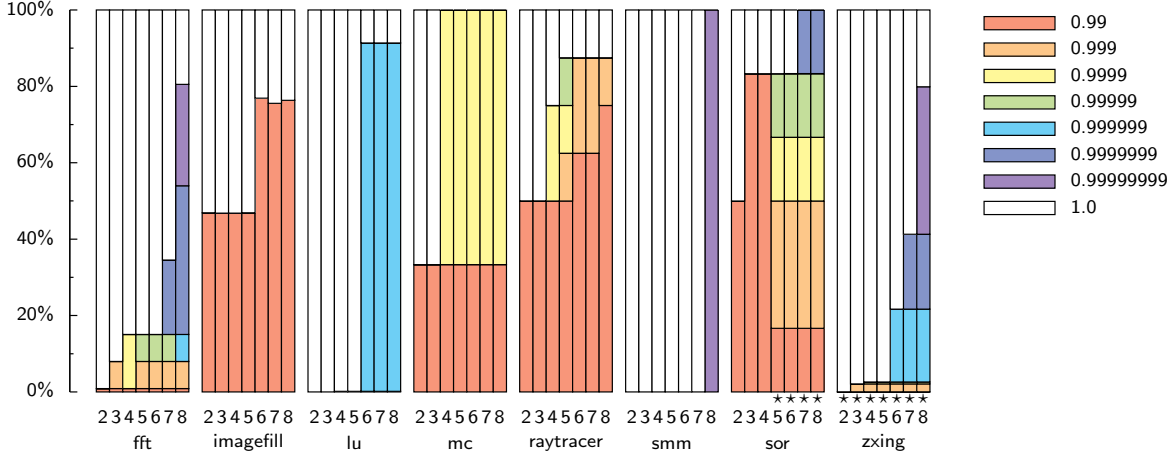




In these charts, horizontal axes show the available reliability levels while the vertical axes show the percent of approximable operations assigned to each level. *Solved* indicates that discrete levels were supplied at compile time while *rounded* indicates that these levels were supplied at runtime. This shows providing known discrete precision levels at compile time offers significant benefits over supplying them at runtime. Not only are more approximable operators approximated, but operators approximated using the rounding approach may be approximated further under the solving approach.

10.3 Granularity of Discrete Levels

We also wanted to offer some insight into how the total number of available discrete levels affects approximation:



In this graph, the horizontal axes shows the number of discrete precision levels used. The bars are grouped by benchmarks. The vertical axes shows the percentage of approximable operations that executed approximately at runtime. Colors denote which precision level operators executed at. Lastly, a star indicates that precision levels were provided at runtime and rounded rather than solved at compile time because the latter took too long to solve.

We hope this information can inform hardware designers on how many discrete levels can be added before seeing diminishing returns to find the balance between cost and flexibility.

10.4 Compilation Time

Compilation time with DECAF can be unpredictable, as seen in Table 1. For the measurements in this table the solving timeout was set to one minute. In general, compilation time increases significantly with larger programs. Additionally, compile time increases when providing discrete operator precision levels to be solved.

The most significant way to reduce compile time is to lower the timeout length as the majority of compilation time is spent optimizing the objective target. It seems possible to significantly reduce timeouts with small impacts on inference, but this has not been empirically measured. Additionally, solver time may be mitigated by reducing the constraints on variables with explicitly declared reliabilities.

11 Future Work

11.1 Error Messages

Error messages in inference are unsatisfying and tough to fix. Due to the interprocedural nature of inference, our error messages only indicate that an error exists *somewhere* in a function. To aid the programmer, the set of constraints for that function are then dumped to standard error. This leaves the programmer to scan through potentially thousands of lines of constraints to find a set that contradict each other.

Z3 offers an “unsatisfiable core” to get a set of constraints that are inherently unsatisfiable. Perhaps we could use this to provide more concise error messages with enough information to figure out why the set of variables are over constrained. This improvement would require significant effort due to the fact that many variables within the solver are temporary variables and operators with little recorded connection to the programmer’s code.

11.2 Modularity

The lack of modularity in DECAF results in full recompilations for small changes to files. It also means that DECAF libraries cannot be distributed without providing access to the source code of the library. This problem may be fixable by storing precision of function return values in terms of function arguments similar to Rely [2].

12 Conclusion

With new approximate hardware supporting multiple precision levels the programmer is presented with fine control over every operation in their program. DECAF aims to reduce the overhead of writing approximate programs by using inference to find optimal operator precisions. DECAF allows the programmer to finely tune their program manually, or let inference do the hard work. Where a no overhead static system breaks down, the programmer has the option to use dynamic tracking to produce values that may be seamlessly reintegrated back into static types.

In addition to aiding programmers, DECAF also helps hardware manufactures. By measuring the effect of different hardware parameters on approximation, we hope to inform the hardware community on how to best build approximate hardware that is both flexible and efficient.

13 Acknowledgements

I would like to thank Adrian Sampson, Dan Grossman, and Luis Ceze for their support and guidance on this research project and life beyond undergrad. Additionally, I would like to thank the rest of the Sampa group for their valuable feedback on practice talks.

References

- [1] B. Boston. jcat. URL <https://github.com/bboston7/jcat>
- [2] M. Carbin, S. Misailovic, and M. Rinard. Verifying quantitative reliability of programs that execute on unreliable hardware. In *OOPSLA*, 2013.
- [3] L. DeMoura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS/ETAPS*, 2008.

- [4] H. Esmailzadeh, A. Sampson, L. Ceze, and D. Burger. Architecture support for disciplined approximate programming. In *ASPLOS*, 2012.
- [5] U. Karpuzcu, I. Akturk, and N. S. Kim. Accordion: Toward soft near-threshold voltage computing. In *HPCA*, 2014.
- [6] D. Pan. A tutorial on MPEG/audio compression. *IEEE Multimedia*, 2(2):60-74, 1995.
- [7] 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.
- [8] A. Sampson, J. Nelson, K. Strauss, and L. Ceze. Approximate storage in solid-state memories. In *MICRO*, 2013.
- [9] S. Venkataramani, V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan. Quality programmable vector processors for approximate computing. In *MICRO*, 2013.
- [10] 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.