Probability Type Inference for Flexible Approximate Programming

Brett Boston[1] Adrian Sampson[2] Dan Grossman[3] Luis Ceze[3]

[1]MIT [2]Cornell [3]University of Washington

OOPSLA 2015

Approximate Computing

- Not every operation in a program has to be correct
- Possible to save energy and execution time

Motivation

- Programmer indicates what operations may be approximated and to what degree
- Hardware allows specification of each arithmetic operator
- Hard to choose right level of approximation for each arithmetic operation in a program

```
int a = 1 +_{0.95} 4
int b = 4 +_{0.99} a
...
int z = y +_{0.9} v
```

Approach

- We'd like to allow the programmer to bound reliability and let the compiler figure out operator correctness
- Our approach frames problem as one of type inference and uses an SMT solver to solve it

Outline

- Basic Approximate Programming Language
- Probability Type Inference
- Hardware Model

All-or-Nothing Approximation

- Type annotations on variables
- Built our system on EnerJ

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

- Endorse is unsound

All-or-Nothing Approximation

What if we want more than all-or-nothing approximation?

- We need something more descriptive
- We have hardware that can support more than two degrees of reliability

Enter: The Paramaterized `@Approx` Annotation

`@Approx(n)`

At any point in the execution, the probability that the value is correct is at least n.

Correct The value is the same as it would be during fully precise execution.

@Approx(n) Rules: Subtyping

```
@Approx(0.9) int a = 1; // legal
@Approx(0.9) int b = a; // legal
@Approx(0.5) int c = a; // legal
@Approx(0.95) int d = a; // illegal
```

Let $\prec$ denote a subtyping relationship between qualified types $q\ \tau$, then

$$\frac{x \geq y}{\texttt{@Approx(x)}\tau \prec \texttt{@Approx(y)}\tau}$$

@Approx(n) Rules: Binary Operators

```
@Approx(0.9) int x = 1;
@Approx(0.9) int y = 2;
@Approx(0.81) int a = x + y;
@Approx(0.7) int b = x + y;
```

- $x + y$ is correct with probability at least 0.81, given $+$ is precise
  - Follows from product rule; $\mathrm{P}(A \cap B) \geq \mathrm{P}(A) \times \mathrm{P}(B)$
- Values approximated through imprecise binary operations

Language Details

- Conservatively treat all values as independent.
- Control flow only allowed on precise values.

Outline

- ~~Basic Approximate Programming Language~~
- Probability Type Inference
- Hardware Model

Problem: Annotation Burden

I have a complex function I'd like to annotate, and I know how precise I'd like the inputs and outputs to be. How should I go about annotating the innards?

Problem: Annotation Burden

I have a complex function I'd like to annotate, and I know how precise I'd like the inputs and outputs to be. How should I go about annotating the innards?

Answer: Guess, get errors, repeat.

Problem: Annotation Burden

I have a complex function I'd like to annotate, and I know how precise I'd like the inputs and outputs to be. How should I go about annotating the innards?

~~Answer: Guess, get errors, repeat.~~
Answer: Type inference!

Solution: Type Inference

Inputs

@Approx(n) explicit annotations

Innards

@Approx inferred annotations

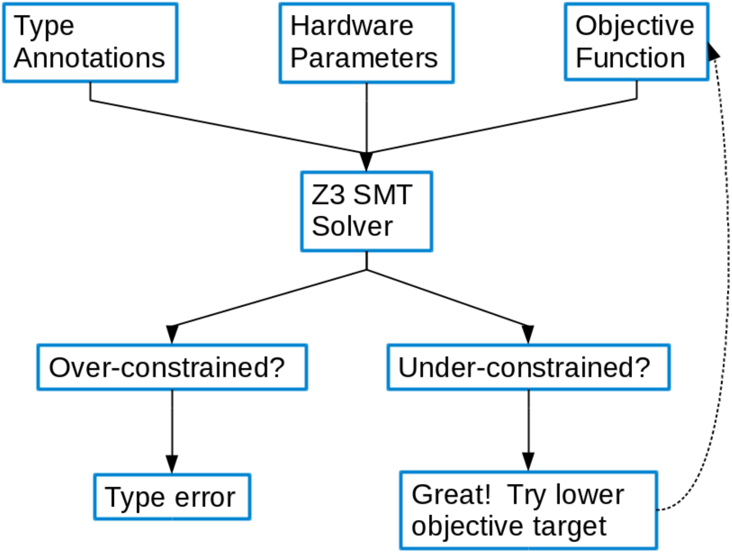
Outputs

@Approx(n) explicit annotations

- Indicate probability
  should be inferred by leaving off parameter.
- Reliability of binary operators
  implicitly inferred such that the expression's reliability minimized as much as possible while still satisfying reliability guarantees.
- Developers use @Approx(n)
  where they have quality requirements.

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

Inferring Probability Types



Inferring Probability Types Example

```
@Approx int x = 1;
@Approx int y = 2;
@Approx(0.81) int z = x + y;
```

```
(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)))
```

- One solution to this is $x = y = 0.9, op1 = 1.0$
- Z3 arbitrarily selects $x = \frac{15}{16}, y = \frac{127}{128}, op1 = \frac{7}{8}$
- Optimal result is $x = y = 1.0, op1 = 0.81$

Objective Function

- Average inferred probabilities across a function, targeting a specific average reliability.
- We can approach an optimal result using a linear search.
- Lower target reliability by a constant amount until problem is unsatisfiable, or times out.

Objective Function Example

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

```
(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)
```

Method Specialization

- Methods specialized for each invocation
- Interprocedural
- Detects cycles in call structure
- Programmer can bound number of times a method will be specialized

```
void example() {
    @Approx(0.9) area1 = triArea(1, 2);
    @Approx(0.95) area2 = triArea(1, 3);
}

@Approx float triArea(@Approx float b,
                      @Approx float h) {
    @Approx float c;
    c = b * h / 2;
    return c;
}
```

@Dyn Types

- @Dyn types track reliability at runtime.
- Dynamic cast back to @Approx(n) with checked_endorse.

```
@Approx int[] nums = ...;
@Dyn int sumD = 0;
for (@Approx int num : nums)
    sumD += num;
@Approx int sum = checked_endorse(sumD, 0.9);
```

Outline

- ~~Basic Approximate Programming Language~~
- ~~Probability Type Inference~~
- Hardware Model

Hardware Simulation

- Z3 exports operator reliability for each operator
- Instrumentation pass replaces operators with calls to custom function in simulator
- Simulator performs approximate operations and records statistics

Discrete Reliability Levels

- Realistically, most hardware will not have continuous operation reliability knobs.
- Allow programmer to specify discrete levels at compile time, or run time

Discrete Reliability Level Constraints

$ enerjc prog.java -Alevels=0.9,0.99,1.0

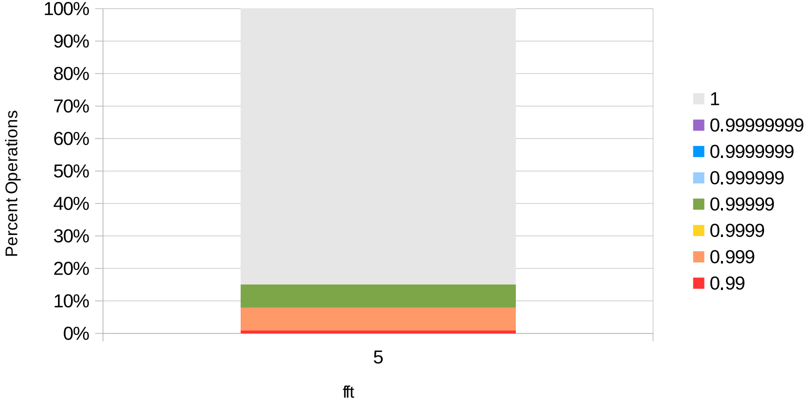
```
(declare-const op Real)
(assert (or (= op 0.9)
            (= op 0.99)
            (= op 1.0)))
```

Benchmarks

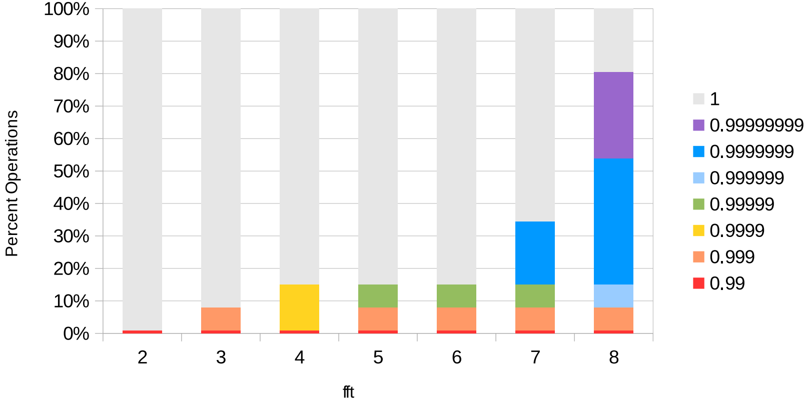
- EnerJ benchmarks
- Constrained inputs and outputs
- 344 LOC - 13180 LOC
- Few annotations needed to achieve approximation
- Overall outputs constrained to `@Approx(0.9)`.

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%

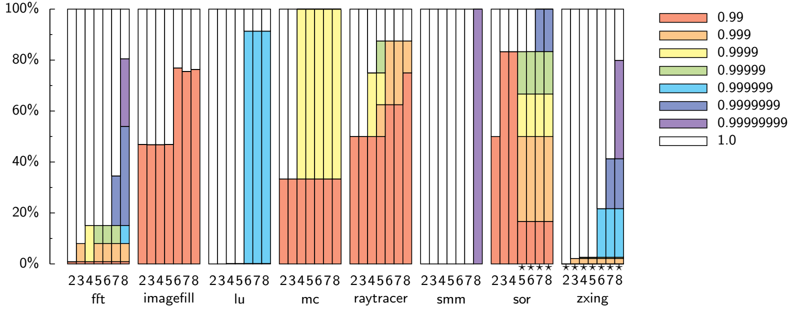
Operator Probabilities for n Discrete Levels



Operator Probabilities for n Discrete Levels



Operator Probabilities for n Discrete Levels



Also in the Paper

- Formal semantics
- Compiler warnings
- Results on solving versus rounding with discrete reliability levels
- Available at `http://sampa.cs.washington.edu/decaf`

Future Work

Modularity Methods are effectively "inlined" for the purpose of type-checking. We could solve this by storing reliability of the return value in terms of the reliability of function arguments.

- Similar to Rely's system. [Carbin et al. 2013]

Error messages Errors in inference tell you only that an error exists *somewhere* in the method.

Summary

- Language abstraction over flexible approximate hardware supporting multiple degrees of approximation.
- Low annotation burden leveraging type inference.
- Hope to inform hardware community.