Tuning Approximate Computations with Constraint-Based Type Inference

Brett Boston Adrian Sampson Dan Grossman Luis Ceze
University of Washington

Abstract

Unreliable hardware can lead to great gains in energy efficiency, but it can be difficult to reason about how unreliable each operation in a computation may feasibly be. To make approximate computing viable, we need tools that can help programmers derive precision–energy trade-offs for individual fine-grained operations while reasoning about the collective impact on the result quality. We formulate the problem of precision tuning as type inference over a system of types parameterized on their accuracy. Our type inference system generates numerical constraints and uses an SMT solver to produce parameters for unspecified types. Programmers can choose to provide explicit types where they make sense and depend on inference where the appropriate accuracy parameter is unclear. Remaining research challenges are discussed.

1. Introduction

Approximate computing seeks to exploit the accuracy–efficiency trade-offs of computer systems and software. Applications that can harness unreliable hardware to do useful work can save time, energy, and complexity over traditional, fully precise execution. But a persistent challenge remains in relating the accuracy of individual operations in a program to the program's overall reliability. Programmers need a way to easily control fine-grained accuracy–efficiency trade-offs that pervade approximate programs while understanding how those individual decisions can compromise important accuracy properties.

This paper proposes a lightweight type system that captures the probability of correctness for every expression in the program, from individual approximate operations to computation outputs. The type system checks that the program upholds programmer-written reliability properties. We propose a system resembling constraint-based type inference that alleviates the need for many annotations. Inference automatically tunes the types to save as much energy as possible while meeting programmer-specified accuracy requirements by choosing the "best" solution to underconstrained systems.

Paired with our inference technique, the proposed type system gives programmers the flexibility to add accuracy requirements where they make sense and omit them where they are less obvious. Expert programmers can optimize approximate code by exerting total control over every type in the program; casual programmers in earlier development phases can opt to rely more heavily on inference. The project is still at an early stage and there are substantial research questions to be resolved, but this inference approach shows the potential for making approximate programming easier throughout the software development lifecycle.

2. Objective

We refine the notion of *precision types* from EnerJ [10] to capture the probability that an expression is correct. In EnerJ's original formulation, every expression has a type that is either *precise*, permitting no approximation, or *approximate*, permitting arbitrary errors. The decision about the degree of precision—*how* approximate an expression is allowed to be—is relegated to another system such as a profile-based tuner. The task of choosing precision is further complicated when each operation may have a distinct precision, as in the Quora ISA [11]. This paper addresses the problem of expressing and tuning these per-operation precisions.

We extend EnerJ's type system to explicitly represent precision parameters in the types. Each approximate type indicates the probability that, in any given execution, the value equals the corresponding value in an error-free execution; if this is satisfied, we would say that the value is *correct*. For example, `@Approx(0.9) int` denotes an integer that is correct at least 90% of the time. This accuracy parameter resembles the *reliability* property proposed by Carbin et al. [2].

Consider this simple example using parameterized `@Approx` type qualifiers:

```
@Approx(0.8) int square(@Approx(0.9) int x) {
    @Approx(0.8) int xSquared = x * x;
    return xSquared;
}
```

The multiplication expression `x * x` is correct when both of the subexpressions are correct (assuming that the multiplication itself is precise). Specifically, the expression is correct with probability at least $0.9 \times 0.9 = 0.81$. The assignment

into `xSquared` typechecks because this probability exceeds the required 0.8 for the local variable.

Intuitively, it is illegal to assign a value with a low guarantee of correctness to a variable with higher precision, but the opposite direction must be legal. Put informally, let $\prec$ denote a subtyping relationship among qualified types $q\ \tau$. Then the rule for precision-qualified types is:

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

For backwards compatibility with EnerJ, `@Approx` is syntactic sugar for `@Approx(0.0)` (no guarantees) and `@Precise` is equivalent to `@Approx(1.0)` (no errors).

If $\Gamma \vdash e : q\ \tau$ gives types $q\ \tau$ to expressions e, then we can write the type rule for addition expressions $a +_p b$ where p denotes the probability that the addition is correct:

$$\frac{\Gamma \vdash e_1 : \texttt{@Approx(}p_1\texttt{)} \qquad \Gamma \vdash e_2 : \texttt{@Approx(}p_2\texttt{)}}{\Gamma \vdash e_1 +_p e_2 : \texttt{@Approx(}p_1 \cdot p_2 \cdot p\texttt{)}}$$

This rule reflects the joint probability of statistically independent events. The expression's output is correct if both operands are correct and the operation itself behaves correctly. So the probability that the sum is correct is the product of the probabilities for those three events.

This sketch illustrates three issues with this straightforward extension of EnerJ's type system:

1. Without syntactic support for annotations on operators, how do we assign precision parameters to operations (as in the multiplication in the above code example)?
2. Marking every variable with a probability seems like a high annotation burden. Can we let the programmer omit some parameters?
3. Methods must be overloaded to enable different degrees of approximation if there is significant variability in parameter correctness.

We can address each of these problems with type inference. We extend the above language with a "wildcard" qualifier `@Approx(*)`. Programmers can use this qualifier directly, but it is also used implicitly for intermediate expressions where types are not explicitly written. With these extensions, the simple example above becomes:

```
@Approx(*) int square(@Approx(*) int x) {
    @Approx(*) int xSquared = x * x;
    return xSquared;
}
```

As a more concrete example, consider the approximate fast inverse square root algorithm [8] used for quick lighting calculations in computer graphics. Here, we extend the algorithm to use parameterized `@Approx` annotations. This allows the graphics programmer to aggressively approximate lighting on objects that are far away, while maintaining greater precision on nearby objects that the viewer is likely to notice. By using type inference, the caller of the function does not need to know how to find appropriate values for internal wild cards as they will be inferred from the probability of correctness for n and the return type.

```
@Approx(*) float invSqrt(@Approx(*) float n) {
    @Approx(*) int i;
    @Approx(*) float x2, y;

    x2 = n * 0.5F;
    i = Float.floatToIntBits(endorse(n));
    i = 0x5f3759df - (i >> 1);
    y = Float.intBitsToFloat(endorse(i));
    y = y * (1.5F - (x2 * y * y));
    return y;
}
```

3. Inferring Precision Types

While constraint-based type inference is nothing new, our type system poses a unique challenge in that its types are *continuous*. We use an SMT solver to find real-valued type assignments given constraint in the form of inequalities.

As an example, consider a program with one unknown precision (that of an operator):

```
@Approx(0.9) int a, b;
...
@Approx(0.8) int c = a + b;
```

The program generates trivial constraints for each variable type, a subtyping inequality for the assignment, and a product constraint for the binary operator:

$$\begin{aligned} p_a &= 0.9 \\ p_b &= 0.9 \\ p_c &= 0.8 \\ p_c &\leq p_{\text{expr}} \\ p_{\text{expr}} &= p_a \cdot p_b \cdot p_{\text{op}} \end{aligned}$$

Here, p_{op} denotes the precision of the `+` operator itself and p_{expr} is the precision of the expression `a + b`. Solving the system yields a valuation for p_{op}. If the system is unsatisfiable, then no precision suffices to meet the programmer's demands and an error is issued.

The generated constraint systems are necessarily underconstrained: there may be multiple satisfying type assignments for a program. In our example, $p_{\text{op}} = 0.99$ satisfies the system, but other valuations are also possible. We want to find the valuation that leads to the greatest energy savings. To do this, we produce an *objective function* that is a proxy for energy savings: specifically, we minimize the total precision over all operators in the program.

The current prototype generates constraints that are then fed to the Z3 SMT solver [3]. Due to the fact that Z3 lacks an

optimizer, we minimize the objective function using binary search. This works by emitting an inequality constraint that limits the objective function's value and searching for the smallest limit that makes the system satisfiable.

Lastly, the compiled binary, including precision values for each operator, may be run on a simulator to gather information about actual energy usage.

4. Known Issues

This work is still at an early stage, so there are a number of issues that we need to address in realizing the system.

Modularity The system as proposed is interprocedural. Functions must be re-checked for each set of parameter types, which effectively in-lines them for the purpose of type-checking. One approach to a modular analysis is to adopt the interfaces of Carbin et al.'s Rely system [2], which expresses a returned value's accuracy in terms of the argument accuracy.

Optimization efficiency Our approach to optimization—binary search on the objective function—is a hack that lets us treat a solver as an optimizer. A more principled approach would formulate the constraints as an optimization problem with known efficient algorithms—e.g., linear, quadratic, or semidefinite programming. Alternatively, we will consider adopting a different full SMT solver that supports optimization directly, a feature that Z3 itself eventually may add [4].

Storage and time-sensitivity Our type system only allows for approximate execution of operations, but approximate hardware has been proposed that allows errors in *storage* [6, 7]. In approximate storage techniques that allow bits to decay over time, error probabilities are correlated with real time. We can approximate this effect with knowledge of variable lifetime.

Error messages Like any type inference system, ours needs a mechanism for emitting helpful error messages when inference fails—i.e., when a program's constraint system is unsatisfiable. In our SMT-based formulation, this amounts to finding those constraints that are "responsible" for preventing satisfiability. One approach is to solve the MAXSMT problem: to find the largest subset of the constraints that are satisfiable. The remaining constraints correspond to expressions that need the developer's attention.

Control flow divergence Our type system does not reason about programs where errors can cause control flow to differ between precise and approximated executions. A formalism will need to define what a variable's precision means when the variable may be defined in one execution and not in the other. Carbin et al. [1] address this issue in a system for proving relational properties for relaxed programs.

5. Related Work

Rely [2] checks reliability specifications, which resemble our parameterized types, given error probabilities for each operation. The goal of our system is to determine the optimal probabilities rather than to check a particular set of probabilities given *a priori*.

ExpAX [5] uses a data flow analysis combined with a genetic algorithm to determine which operations in a program to approximate without requiring per-variable or per-operation annotation. Our approach solves a complementary problem: it determines the *degree* of precision for operations that are explicitly annotated as approximate.

Precimonious [9] solves a related problem in floating-point programming: how many digits are required in each intermediate number representation to meet an accuracy bound in the output? The two kinds of precision are orthogonal: a full system could combine this analysis of rounding errors with our proposed analysis of probabilistic errors.

6. Conclusion

Approximate programming models need tools that help developers decide how much precision is necessary throughout an algorithm. However, fully automatic approaches are equally problematic: programmers sometimes need fine-grained control and can (wisely) distrust opaque auto-tuners. Type inference offers an intermediate solution. Programmers can collaborate with the tool by providing sparse annotations and depend on inference to fill in the rest.

Acknowledgments

This work was supported in part by C-FAR, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

References

[1] M. Carbin, D. Kim, S. Misailovic, and M. C. Rinard. Reasoning about relaxed programs. In *PLDI*, 2012.

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

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

[4] L. de Moura and G. O. Passmore. (Exact global nonlinear) optimization on demand. ADDCT-2013, CADE-24, 2013. URL `http://www.cl.cam.ac.uk/~gp351/Passmore-Moura-ADDCT-talk-2013.pdf`.

[5] H. Esmaeilzadeh, K. Ni, and M. Naik. Expectation-oriented framework for automating approximate programming. Technical Report GT-CS-13-07, Georgia Institute of Technology.

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

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

[8] C. Lomont. Fast inverse square root. 2003. URL `http://www.lomont.org/Math/Papers/2003/InvSqrt.pdf`.

[9] 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.

[10] 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.

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