Programming with Angelic Nondeterminism

Shaon Barman, Nicholas Tung, Joel Galenson

Angelic programming

Motivation

Why hasn't Moore's Law revolutionized programming? In model checking, cycles fuel bug discovery, improving code quality, but programmers still write programs with their bare hands. Can we give them a coding assistant?

Key inspiration

We learn and design algorithms by studying examples, before understanding or writing pseudocode. Can an oracle generate these examples for us?

Angelic nondeterminism

We developed a language that allows the programmer to ask an oracle for demonstrations of an algorithm's execution before the algorithm is developed.

The oracle provides values that the programmer does not yet know how to compute. The programmer then generalizes these executions into an algorithm.

The oracle is an angelically nondeterministic choice operator (it looks into the future of the execution).

Our current implementation uses a parallel backtracking solver to find correct executions. Correctness is defined in the program by assertions.

Example angelic program

Imagine we want to know whether to reverse a list with a forward or backward traversal. We first ask an oracle for a demonstration.

Angelic Program

```
def reverse(list) {
    while (choose(true,false)) {
        choose(Node).next = choose(Node)
    }
    reversedList = choose(Node)
    assert reversedList is reversal of list
    return reversedList
}
```

Demonstration

This execution right (Figure 1) is neither forward nor backward because the oracle is too unconstrained, so we need to revise the angelic program to walk the list in order. This can by accessing the list using an iterator.

Synthesis constructs in Scala

We embedded synthesis constructs in Scala, and implement them using the SKETCH project's CEGIS solver (which in turn uses SAT solvers).

Why Scala?

Compared with SKETCH, Scala is an general purpose language, with less non-standard syntax. Scala is expressive, and has an advanced type system. It also uses the JVM and interoperates with Java code.

Sketching constructs

- Data structure constructs (using type annotations),

```
class MyClass(a : Int @ Range(-3 to 3), b : Boolean)
val v1 : MyClass = !!       // v1 = new MyClass()
                            // v1.a = choose( [-3, 3] )
                            // v1.b = choose(true, false)
```

- For performance and clarity of implementation, most solving logic uses compiler transformations, lowering Scala to the existing SKETCH language.

Current progress

- Scala to SKETCH uses graph rewriting to do most lowering operations (rewriting class methods as functions, converting program statements to C-like semantics).
  - A Scala compiler plugin emits GXL (an XML-based graph format). Since the program is now a graph instead of a tree, symbols are no longer special entities; they are only graph nodes.
  - GrGen, a fast algebraic graph transformation framework, is used. GrGen has an expressive language for productions ("rewrite rules").
  - SKETCH AST nodes then need to be constructed; this is done using a mini-language, which generates appropriate Java code.
- "Hello world" programs (just recently) working.

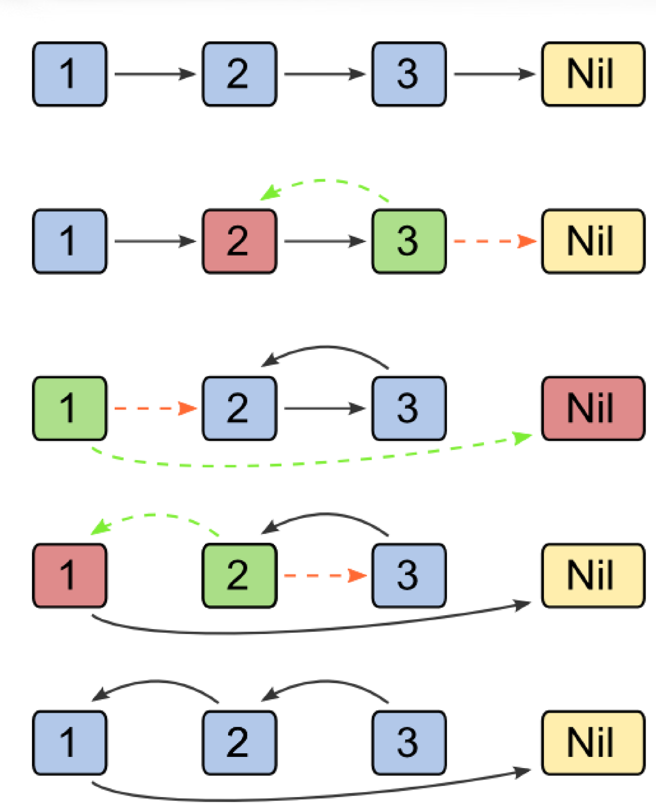
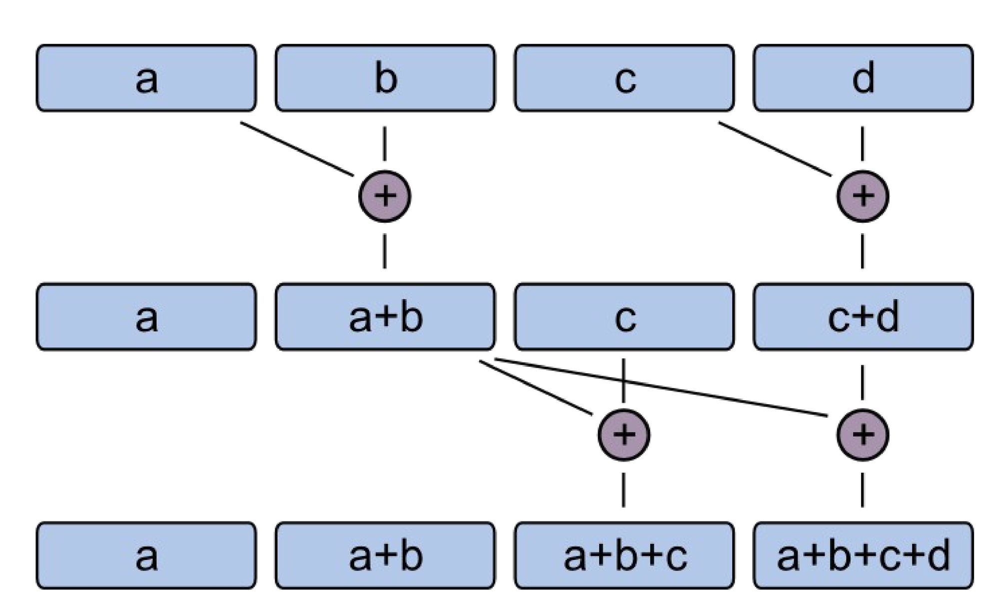


Figure 1
A demonstration of a list reversal

Figure 2
A demonstration of a parallel scan with n=4



Future work

Refinement

- Angelic programs are refined until we create a program free of choice statements. This means implementing angels with subprograms which may themselves be angelic
- A refinement should not allow traces not allowed in the refinee
- We need a way to link an angelic program to the refinee.
  - We do this by allowing those subprograms to return only those values that angels returned in the refinee. This is done with queues.

Angelic entanglement

- Angelic programs can create numerous traces.
  - Difficult to find insight in so many demonstrations.
- We say two choice statements are entangled if they are not independent.
  - Another way of saying this is that two choice statements compensate for each other and cannot change independently
- Entanglement can be used to classify traces, so that only a limited number of representative traces are shown to the programmer.

Implementing a parallel scan

Say we want to implement a parallel scan algorithm in time O(log n).

```
def scan(x : Array[Int])
    y : Array[Int]
    for (step ← 0 until log n)
        // this inner loop can be executed in parallel
        for (r ← 0 until n)
            if (choose(true,false))
                // actual computation
                y[r] = x[r-choose(n)]+x[r-choose(n)]
            else
                y[r] = x[r]
        x = y
    return x
```

An example execution is shown in the figure left The outer for loop limits the demonstrations to only use log n steps.

We can also add a counter (near the "actual computation" comment) in order to limit the number of adds, to ask the oracle if there exists a work efficient algorithm.

Having added this counter constraint, we found that the angels cannot generate a demonstration using log n time steps for the outer loop. We found that the outer loop required 2*log n steps.