Automated Incrementalization through Synthesis

Rohin Shah
rohinmshah@berkeley.edu
University of California, Berkeley

Rastislav Bodík
bodik@cs.washington.edu
University of Washington

1 Introduction

Incremental computation allows the value of a complex program to be updated efficiently upon small changes to the inputs of the program, leading to asymptotic speedups which are crucial for reasonable performance in many domains. However, an incremental program is typically significantly more complicated than its non-incremental counterpart, and so it is desirable to automatically generate an incremental program from a non-incremental one.

In many domains, we require fast incremental programs without the runtime overhead of dynamic techniques. For example, many approximate inference algorithms in machine learning change a small portion of the state, and have to incrementally update many data structures in response. These algorithms are computation intensive, making runtime overhead unacceptable, and so incremental versions are designed by hand, which is tedious and complicated. To automatically generate the incremental versions, we use *program synthesis*, in which we search through a space of potential incremental programs looking for one that is correct.

Program synthesis can handle arbitrary input programs as long as it understands the required primitives, and generates efficient programs with no runtime overhead. In addition, it is not sensitive to the specific way in which a programmer may have written the input program, since it models only the output value of the program and ignores the structure and syntax of the program itself. It can consider invariants and discover implicit assumptions in order to generate better incremental programs. The key challenge with synthesis is scaling it to to large programs, but we have made progress by using deductive techniques to prune the search space.

2 Example

Consider the example of finding a permutation's inverse, where the original permutation may be changed by composing it with a transposition (a permutation that swaps two elements and leaves all other elements alone). We represent a permutation as an array that maps indexes of the input to indexes of the output. To compose the permutation with a transposition (i, j), we simply swap the elements stored at indices i and j, leading to the specification in Figure 1a.

Given this specification, we want to find a fast incremental program that is *from-scratch consistent*, that is, running the incremental program produces a state which is equivalent to the state that we would get by recomputing `inverse` from scratch. More formally, we want to solve:

$$\exists \Delta f \ \forall I, \Delta I : \quad O + \Delta f(I, \Delta I, O) = f(I + \Delta I) \qquad (1)$$

```
(define int (Integer-type))
(define-symbolic LEN int)
(define (permutation? lst)
  (equal? (sort lst <) (range LEN)))

(define-mutable perm (Vector-type LEN int)
  #:invariant (permutation? (vector->list perm))
  #:deltas
  [(define (transpose! [i int] [j int])
     (define tmp (vector-ref perm i))
     (vector-set! perm i (vector-ref perm j))
     (vector-set! perm j tmp))])

(define-incr inverse (Vector-type LEN int)
  (let ([r (make-vector LEN 0)])
    (for ([i (range LEN)])
      (vector-set! r (vector-ref perm i) i))
    r))
```

(a) Specification for the permutation problem.

```
(define (transpose! i j)
  (define tmp (vector-ref perm i))
  (vector-set! perm i (vector-ref perm j))
  (vector-set! perm j tmp))
  (vector-set! inverse (vector-ref perm i) i)
  (vector-set! inverse (vector-ref perm j) j))
```

(b) Solution to Figure 1a. The highlighted code is the code generated by synthesis. Note that the code is as good as handwritten code – it consists of two loads and two stores and does not maintain any state at runtime.

Figure 1. Example problem: Incrementally updating the inverse permutation computation when the original permutation is composed with a transposition (i, j) (which swaps the elements at i and j).

where I is the input data structure (`perm`), $O = f(I)$ is the output data structure (`inverse`), ΔI is the small change to the input I (`transpose!`), and Δf is the desired update function.

Note that the solution to the permutation example (Figure 1b) is only correct given a specific invariant of the input and the particular structure of the input delta. The invariant is that `perm` is actually a permutation, that is, it is a bijective function $[0, \text{LEN}) \longrightarrow [0, \text{LEN})$. If we had to also consider cases where this does not hold, the generated solution would be incorrect. In addition, the first `vector-set!` in `transpose!`

breaks the invariant, and so to reason properly we must consider both statements together rather than one after the other. These aspects make it impossible for existing static techniques to incrementalize this program.

We have built a prototype tool that can synthesize correct incremental programs using Rosette [5], which provides a symbolic evaluation engine for a large subset of Racket. The pseudocode for the generated Rosette program for the `perm` problem is shown in Figure 2. We represent every quantified variable in Equation 1 (that is, I, ΔI and Δf) with a symbolic value that encodes the set of all possible concrete values that variable could have. In order to compute the values necessary in Equation 1, we run the corresponding parts of the specification. Since some of the values are symbolic, Rosette will model the semantics of the program symbolically. Finally, we use the computed values to `assert` that the correctness condition holds, and call `synthesize` which delegates the search to an SMT solver.

```
(define perm (sym (Vector-type LEN int)))
(define i (sym int))
(define j (sym int))
(assume (permutation? (vector->list perm)))

(define inverse (compute-inverse))
(transpose! i j) ;; Mutate perm
(assume (permutation? (vector->list perm)))
(define expected-inverse (compute-inverse))

;; Search space of programs that fix inverse
(make-and-run-symbolic-program)
(synthesize
 ;; For every input and delta
 #:forall (list perm i j)
 #:guarantee
 ;; It is as though we recomputed from scratch
 (assert (equal? inverse expected-inverse)))
```

Figure 2. Pseudocode showing the Rosette program generated from the specification in Figure 1a. The highlighted code is taken directly from the specification, where `compute-inverse` refers to the expression used to compute `inverse` from scratch. `sym` is a function that, given a type, produces a symbolic value representing all possible concrete values of that type. `synthesize` is a Rosette built-in that here produces a formula "∃ program: ∀ inputs and input deltas: guarantee correctness condition", and then solves it using an SMT solver.

Since Rosette models what the output value is and does not look at how it is computed, it does not matter how we write our specification. For example, we would get the same result if we rewrote `inverse` in a functional style, or eliminated the temporary variable in `transpose!`.

3 Discussion

Incremental computation is an active field of research, and many automated techniques have been developed. Existing techniques fall into two main areas – rule based static systems and memoizing dynamic systems.

A rule based system has a set of composable rules which are used to incrementalize programs in a particular domain specific language (DSL). Paige applied this technique to SETL [4] and the databases community has applied it to incremental view maintenance, most recently in DBToaster [2]. Though these only work on programs within the DSL, it is unclear how restrictive existing DSLs are.

A memoizing dynamic system builds a computation graph and uses change propagation to recompute outputs, reusing cached intermediate values where possible. Acar laid the foundations in his PhD thesis [1], and more recently Adapton [3] made the technique demand-driven. These systems can handle many programs, but introduce significant runtime overhead through the memoization and computation graph. Often, the program must be written in just the right way in order to benefit from change propagation.

Synthesis avoids these disadvantages, but it does not scale well, since the search space of programs grows exponentially in the size of the output program, and so complex programs can have long compile times. However, the synthesis community has developed techniques to improve scalability, and by adding type-based and mutability-based pruning to our tool we are able to scale to interesting programs.

Unlike existing techniques, our tool cannot discover auxiliary data structures that can help incrementalize the input program. We could simply ask Rosette to simultaneously synthesize an intermediate data structure, but the resulting search space would likely be intractably large. Alternatively, we could search over the set of data structures that store intermediate values produced when executing the input program. Finally, we could try to write "rules" that analyze the input program and generate potential auxiliary data structures, and use synthesis to fill in the rest of the implementation.

References

[1] Umut A. Acar. *Self-adjusting Computation.* PhD thesis, Pittsburgh, PA, USA, 2005. AAI3166271.

[2] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. Dbtoaster: Higher-order delta processing for dynamic, frequently fresh views. *Proc. VLDB Endow.*, 5(10):968–979, June 2012.

[3] Matthew A. Hammer, Khoo Yit Phang, Michael Hicks, and Jeffrey S. Foster. Adapton: Composable, demand-driven incremental computation. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 156–166, New York, NY, USA, 2014. ACM.

[4] R. Paige. *Formal Differentiation: A Program Synthesis Technique.* Computer Science Series. UMI Research Press, 1981.

[5] Emina Torlak and Rastislav Bodík. A lightweight symbolic virtual machine for solver-aided host languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 530–541, New York, NY, USA, 2014. ACM.