Scalable Synthesis with Symbolic Syntax Graphs

Rohin Shah
UC Berkeley
rohinmshah@berkeley.edu

Sumith Kulal
IIT Bombay
sumith@cse.iitb.ac.in

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

General-purpose program synthesizers face a tradeoff between having a rich vocabulary for output programs and the time taken to discover a solution. One performance bottleneck is the construction of a space of possible output programs that is both expressive and easy to search. In this paper we achieve both richness and scalability using a new algorithm for constructing symbolic syntax graphs out of easily specified components to represent the space of output programs. Our algorithm ensures that any program in the space is type-safe and only mutates values that are explicitly marked as mutable. It also shares structure where possible and encodes programs in a straight-line format instead of the typical bushy-tree format, which gives an inductive bias towards realistic programs. These optimizations shrink the size of the space of programs, leading to more efficient synthesis, without sacrificing expressiveness. We demonstrate the utility of our algorithm by implementing Syncro, a system for synthesis of incremental operations. We evaluate our algorithm on a suite of benchmarks and show that it performs significantly better than prior work.

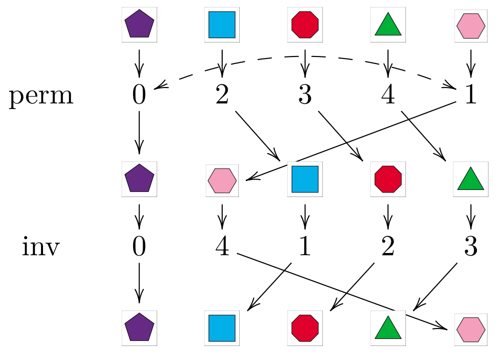
1 Introduction

Program synthesis is the task of automatically finding a program that meets a high level specification. Researchers have approached the problem by identifying a class of programs, formalizing it with a domain-specific language, and creating an efficient synthesis algorithm for that domain [30]. In *component-based synthesis*, this is done by providing a set of components that the program can be built out of [16, 9, 10].

In order to implement a synthesis algorithm for a new domain, researchers can take one of several approaches. In order of decreasing effort, they can build a new algorithm from scratch, specialize a generic meta-algorithm with domain-specific operators [27], or leverage an existing framework that provides a general-purpose synthesis algorithm [29, 19]. However, when leveraging an existing framework, scalability constraints make it necessary to build the space of programs with a carefully constructed grammar, resulting in long development times.

We provide a novel algorithm for constructing symbolic syntax graphs, which represent the program space compactly and can then be efficiently searched using an SMT solver. The algorithm can be specialized to a domain by specifying constraints on the way each component can be used, most notably through types. The algorithm works with both functional and imperative code. Normally with imperative code, the programmer must specify many *frame conditions* [23] that prevent changes to particular variables in order to prevent the synthesizer from returning pathological solutions. We introduce a *mutability analysis* that instead asks the user to *whitelist* particular variables as modifiable. The algorithm also encodes the program space in a format that leads to better performance from the SMT solver by sharing structure where possible. Finally, we structure the symbolic syntax graph as a sequence of temporary variable definitions, rather than as an expression tree, in order to more finely control the space

Submitted to:
SYNT 2018



(a) If our input permutation is $[0, 2, 3, 4, 1]$, then its inverse is $[0, 4, 1, 2, 3]$, as we can see by considering their action on a list of five shapes. We now want to modify `perm` by transposing the elements at indices 0 and 4. How should we modify `inv`?

```
(define perm (vector 0 2 3 4 1))
(define inverse (vector 0 4 1 2 3))

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

(transpose! 0 4)
(fix-inverse! 0 4)

;; Given perm, inverse, and transpose!, we
;; want a procedure fix-inverse! that will
;; update inverse appropriately without
;; recomputing it from scratch.
```

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

;; The solution that we want to synthesize
;; for fix-inverse!.
```

(b) Rosette code for permutation problem and the solution.

Figure 1: The permutation inverse problem.

of programs and to bias the synthesizer towards more realistic programs. The resulting system can be used both as a fast general-purpose expression synthesis tool, as well as a framework upon which domain-specific systems can be built.

As a case study, we implement Syncro (Synthesis of Incremental Operations), which can synthesize incremental updates Δf given a from-scratch program f. Syncro is able to quickly synthesize interesting programs from large search spaces, validating our approach.

Specifically, we make the following contributions:

1. A novel algorithm for constructing symbolic syntax graphs that automatically enforces type safety and some frame conditions.
2. Optimizations that improve synthesis time compared to a simple recursive grammar.
3. A case study where we implement Syncro, a system for synthesizing incremental operations on data structures.
4. A comparison of our algorithm to existing expression synthesizers that finds that our algorithm can synthesize programs faster with less programmer effort.

2 Overview

Consider the task shown in Figure 1. We have two permutations, represented as vectors (arrays) of numbers, that are inverses of each other. We now transpose two elements in the first permutation, and we want to update the second permutation so that it continues to be the inverse of the first permutation. What code should we write?

This is a typical program synthesis problem that we could solve with a generic synthesis framework, such as Sketch [19] or Rosette [29]. In Rosette, we would have to:

- Define the space of programs to search over.
- Specify a correctness condition.

Given these pieces, Rosette will translate the semantics of the program and the correctness condition to SMT formulas through symbolic execution. These formulas are then solved by an off-the-shelf solver.

However, writing these pieces can be quite a lot of work for the programmer. Typically, to define the space of programs, the programmer would write down a context-free grammar over a symbolic language [5], which would generate a *symbolic syntax graph* (SSG) that encodes the space of abstract syntax trees (ASTs) drawn from the grammar with a bound on the depth of the tree. The grammar must be designed carefully in order to get synthesis to scale.

To reduce the effort that the programmer must put in, we would like to automate the process of building the program space. We take the approach of *component-based* synthesis, where the user specifies a set of components to use during synthesis. For example, in the permutation example, we may have the `vector-set!` and `vector-ref` components. We can then write a program space generator takes in a type and produces a symbolic syntax graph (SSG) such that any program encoded by the SSG would produce a value of the given type when evaluated. This can be implemented by first introducing a choice over which component to use at the current node, and then recursively producing SSGs for the children with types consistent with the chosen component. In addition, as described in Section 4, we can apply several optimizations, such as subgraph sharing and common subexpression elimination, that reduce the size of the search space or produce better encodings for the SMT solver. The resulting program space outperforms handwritten grammars on large benchmarks.

Another issue is that when specifying the correctness condition, it is very easy to forget to provide a relevant frame condition. For example, for our permutation example, the correctness condition could be that `perm` and `inverse` must be inverses of each other. However, we must also assert a frame condition that says that `perm` must remain the same – otherwise the synthesizer may return the program that undoes the transposition, rather than updating `inverse`.

Instead of asking for frame conditions, we ask the programmer to specify which variables are allowed to be mutated. For permutation, we specify that `inverse` can be mutated, but `permutation` cannot. We use a novel *mutability analysis* to ensure that every program in the search space only modifies values that originate from variables marked as mutable. This leads to a performance improvement, since it reduces the size of the search space.

Once the symbolic syntax graph is constructed, it is combined with the correctness condition, converted to an SMT formula, and solved, returning the solution shown in Figure 1.

3 Background: Symbolic Syntax Graphs

A symbolic syntax graph (SSG) is a data structure that allows us to represent a large space of programs, such that we can efficiently use symbolic execution to simulate the results of running each of the programs in the space. It is a generalization of an abstract syntax tree (AST) in which in addition to concrete nodes (such as `if` and `+`), there can be *choice* nodes that choose between their children based on a symbolic condition. It is a directed acyclic graph instead of a tree because nodes can share children for efficiency.

In order to use SSGs for program synthesis, we must be able to efficiently *interpret* them to check the correctness conditions. We focus on an implementation of SSGs in Rosette, which uses *symbolic execution* to interpret SSGs.

Rosette [29] provides support for angelic execution, verification and synthesis. Users write programs as if they only had to work on concrete inputs, and Rosette will automatically lift the programs to work on symbolic inputs as well. In our context, we write an interpreter for our language $\mathcal{L}$ in the standard way, as though it would only have to work on ASTs. We then construct a symbolic syntax graph S using Rosette's built in functions for constructing symbolic values, and pass it in to the interpreter. Rosette will then use symbolic evaluation to run the interpreter on S, producing a symbolic value representing the output of the computation that can then be used in a correctness condition.

```
(require rosette/lib/angelic
         rosette/lib/match)

;; Define the interpreter
(define (eval program val)
  (match program
    [(list function x y)
     (function (eval x val) (eval y val))]
    ['x val]
    [other program]))

;; Build a symbolic program
(define fn (choose* + - *))
(define arg1 (choose* 1 2 3 'x))
(define arg2 (choose* 1 2 3 'x))
(define program (list fn arg1 arg2))
```

```
;; Define a symbolic input
(define-symbolic* input integer?)
;; Synthesize the desired program
(define model
  (synthesize
   #:forall (list input)
   #:guarantee
   ;; Correctness condition
   (assert (equal? (eval program input)
                   (+ input input input)))))

;; Evaluates to (* 3 x)
(evaluate program model)
```

Figure 2: Rosette code for synthesis. Our language consists of arithmetic functions of a variable `'x`. We can use constants `1`, `2`, and `3`, and the functions `+`, `-` and `*`. The symbolic syntax graph `program` consists of all programs in the language up to depth 1.

A complete example is shown in Figure 3. We consider a simple language of arithmetic expressions that can have a single variable which must be named `x`. The interpreter is written in the straightforward way. To construct our SSG, we use the Rosette function `choose*` to create choice nodes, and use S-expressions for other internal nodes. The `choose*` function creates a *symbolic union* whose value is determined by fresh symbolic boolean variables created by Rosette. A valuation for these boolean variables induces a valuation for the SSG as a whole. We interpret the SSG over all possible inputs, producing a symbolic value dependent both on `input` and the symbolic boolean variables introduced by `choose*`, and assert a correctness condition on this value. We then ask Rosette to produce a valuation for the boolean variables that satisfies this specification for any value of `input`. The returned valuation of boolean variables induces a valuation for the SSG, which we use to produce our synthesized program.

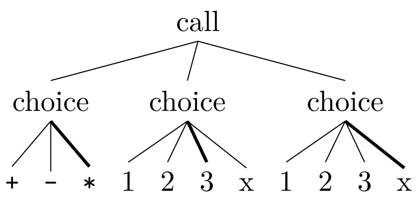


Figure 3: The symbolic syntax graph created by the Rosette program. The bold edges are the edges chosen by the valuation to give the program `(* 3 x)`.

Rosette also has built-in support for *merging* arbitrary Rosette values, including SSGs. When we create a choice node that chooses between multiple SSGs, the choice node is propagated down in order to shrink the resulting graph, as illustrated in Figure 4. This optimization is crucial for efficiency and is performed automatically by Rosette whenever a new choice node is created.

The reader familiar with Rosette may wonder why we are not using `define-synthax` or other synthesis-specific tools in Rosette. While such tools would simplify the presentation of this example, they are not general enough to support the algorithms in Section 4.

4 Constructing Symbolic Syntax Graphs

We now turn to the problem of constructing symbolic syntax graphs automatically. In Figure 3, lines 12-16, we explicitly constructed the symbolic syntax graph of an appropriate depth with the appropriate functions, terminals, and constants, and made sure that these were all combined in a type-safe way. We would now like to generate the SSG automatically without sacrificing performance.

4.1 An Algorithm using Components

When building a context-free grammar for a domain-specific language for program synthesis, it is important to consider both which abstractions are available for use in the grammar, and how the abstractions can be composed with each other. So, we ask the user to provide a set of *components*, which correspond to abstractions or constructs in the domain-specific language. Information about component composition is provided through *types* in most cases, but can also be provided through more general constraints if necessary.

The basic operation of a component is Make-node, which constructs an SSG node given SSGs for the children nodes. In order to produce performant SSGs, we also need to enforce constraints, such as type safety, over the SSGs. So, each component f must implement Get-child-constraints, which given a constraint C, infers the constraints that the children SSGs must satisfy in order for the SSG created by f to satisfy C. Note that the constraints on each child must be independent of each other. The specific form of the constraints will be discussed in future sections.

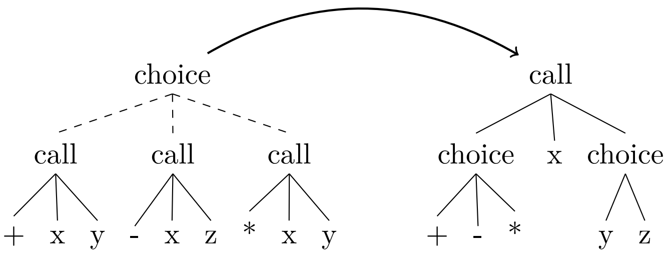


Figure 4: An example of merging in Rosette. We initially want to choose between three ASTs. Rather than creating a single choice node at the top, Rosette *merges* the call nodes, and puts choice nodes over the subexpressions where necessary. Note that there must be constraints on the choice node, for example to disallow the program `(+ x z)`. See [29] for details.

Now, given a set of components that implement these two operations, we can construct an SSG as shown in Algorithm 1. For every component f, we recursively produce the SSGs for the children of f, and make an SSG node out of that. We then introduce a choice node over all of these SSGs at the top level and return that.

Let us consider the case where we do not impose any constraints on the SSG (or equivalently, all constraints in Algorithm 1 are the `true` constraint, which is always satisfied). In this case, if we had components corresponding to `+` and `<`, then we would generate an SSG that includes the AST `(+ (< x 3) x)`, which is not type safe. We assume that programs that are incorrect in this way (that is, they are not successfully evaluated by the interpreter, whether due to type errors, syntax errors, or others) raise an error in Rosette. This means that when designing the operations in a domain-specific language, the operations should not fail silently when presented with invalid inputs, but should instead raise an error.

During synthesis, Rosette will discard incorrect programs as soon as they generate an error. So, even with no constraints on the SSG, Algorithm 1 is correct, in that we could perform synthesis with the SSG it returns. The issue is performance – as the size of the SSG increases and the number of possible types increases, the number of ill-typed programs represented by the SSG grows drastically. Rosette then spends most of its time attempting to evaluate these incorrect programs and discarding them when they fail. We must prevent most type unsafe programs from being generated in the first place if we want to have scalable synthesis.

The constraint C in Algorithm 1 allows us to prune away bad programs, as long as we implement the following:

R1. We must ask the user to specify the top-level constraint on the desired program (line 1).
R2. We must be able to check whether a given terminal satisfies a given constraint (line 3).
R3. For every component, we can infer the constraints on the children of the component that ensure that the constraint on the overall SSG holds (line 8).

4.2 Type Constraints

In order to generate only well-typed programs, we add a *desired type* τ as part of the constraint C. Then, CREATE-SSG must return an SSG such that any valuation of that SSG will evaluate to a value of type τ. We start with the Hindley-Milner type system, so that we have a set of monotypes (vectors, lists, integers, booleans, functions, etc.) and parametric polymorphism. We extend the type system with record types, enums, and subtyping of monomorphic types.

According to requirement R1, we must now ask the user to provide a type for the value computed by the desired program. Any variables that we may use during synthesis must also have types provided, in order to satisfy R2. R3 implies that we must be able to perform *top-down* type-based reasoning, that is, given the type for the larger SSG, deduce the types for the smaller child SSGs. In the case where the component is a function such as `+`, we must be able to infer the domain of the function given its range. This is trivial for functions without polymorphism, as the types for the child SSGs are given by the domain of the function.

Algorithm 1 SSG construction algorithm. The constraint C can include type (Section 4.2), syntactic (Section 4.3), and mutability information (Section 4.4).

```
Require: $F$ is a list of components
Require: $V$ is a list of terminals
Require: $C$ is a constraint that the SSG must satisfy
Require: $d$ is the maximum depth of the SSG
 1: function CREATE-SSG($F$, $V$, $C$, $d$)
 2:     ▷ Can use any terminal that satisfies $C$
 3:     $O \leftarrow$ FILTER($V$,$C$)
 4:     if $d = 0$ then
 5:         return CHOOSE($O$)
 6:     for $f \leftarrow F$ do
 7:         $S \leftarrow []$
 8:         $C_{subs} \leftarrow$ GET-CHILD-CONSTRAINTS($f$,$C$)
 9:         if $C_{subs} \neq$ FAILURE then
10:             for $c_{sub} \leftarrow C_{subs}$ do
11:                 $S \leftarrow S +$ CREATE-SSG($F$,$V$,$c_{sub}$,$d-1$)
12:             if FAILURE $\notin S$ then
13:                 $O \leftarrow O +$ MAKE-NODE($f$,$S$)
14:     return CHOOSE($O$)
15: function CHOOSE($L$)
16:     if len($L$) $= 0$ then
17:         return FAILURE
18:     else
19:         return CHOICE-NODE($L$)
```

However, deducing the domains of polymorphic functions in a symbolic setting is challenging. Consider deriving the child types for `vector-set!`, which has type `Vec`$(\alpha,\beta) \to \alpha \to \beta \to$ `Void`, with $\alpha \subseteq$ `Index`. Suppose that we have two vectors v_1 and v_2 that we could mutate, of types `Vec(Int, Int)` and `Vec(Int, Bool)` respectively. We would like to deduce that the third argument to `vector-set!` must be either an `Int` or a `Bool`, but since GET-CHILD-CONSTRAINTS must return constraints that are independent, we must deduce the type of the third argument from the type of `vector-set!` alone, without looking at v_1 and v_2, and so it seems as though we are forced to conclude that the third argument can be any arbitrary type β.

One solution is to rewrite Algorithm 1 and allow the constraint on the third argument to depend on the values of the first and second arguments. However, this results in a different problem – when we are inferring the type for the third argument, we will now infer that it is either `Int` or `Bool` depending on the value of the symbolic boolean formula that Rosette introduced to choose between v_1 and v_2. This means that when we make a recursive call to CREATE-SSG to generate the SSG for the third argument, we pass in a *symbolic* type, and so any further reasoning on the types will itself be symbolic, generating large formulas due to path explosion during symbolic execution. It is crucial that we avoid this, and make sure that all arguments to CREATE-SSG are concrete.

Our solution is deceptively simple. Since we have a relatively small number of terminals and components, and a small bound d on the sizes of programs, we can simply enumerate all possible monomorphic types that could be created by an ASG of size up to d, by instantiating every polymorphic type with the concrete monomorphic types that it could be applied to. (Note that this can still express a very large program space, since the program space grows exponentially in d.) In the example above, we would replace the `vector-set!` component with two copies of itself, one with type `Vec(Int, Int)` $\rightarrow$ `Int` $\rightarrow$ `Int` $\rightarrow$ `Void`, and the other with type `Vec(Int, Bool)` $\rightarrow$ `Int` $\rightarrow$ `Bool` $\rightarrow$ `Void`, and then run CREATE-SSG with these two new components.

4.3 Syntax Constraints

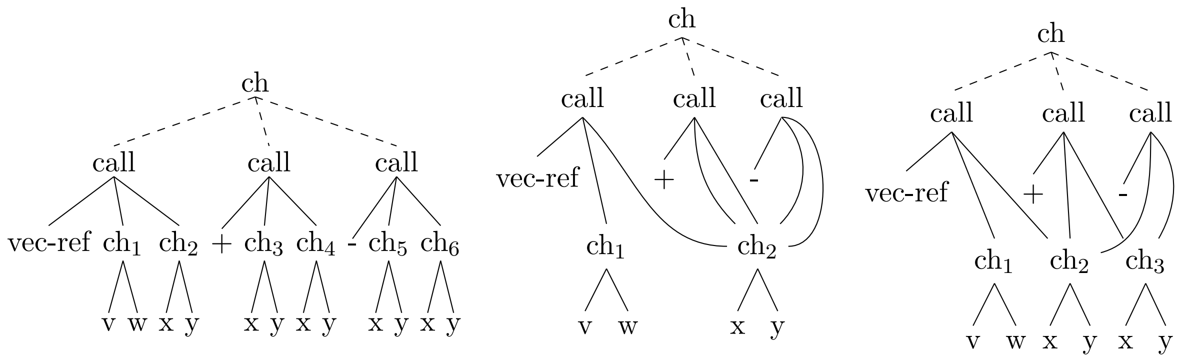
Syntax constraints are used by components that correspond to special forms in order to ensure that the child SSGs are syntactically valid. For our components, we only have one syntactic constraint – for `set!`, the first child must be a variable, not a nested expression.

4.4 Mutability Constraints

Often when working with imperative programs, we must write frame conditions that specify that certain variables remain unchanged after the program has executed. In the permutation example, we need to ensure that `permutation` is not changed, since we only want to modify `inverse`. If we don't specify this frame condition, the solve may come up with the program that simply undoes the transposition to `permutation`, since that satisfies our correctness condition (that `inverse` and `permutation` are inverses of each other).

However, frame conditions are evaluated after the SSG has been constructed, and so they do not prevent incorrect programs from being generated in the first place. As a result, the solver may have to consider these incorrect programs and reject them for violating the frame condition. We can instead use mutability constraints to avoid generating incorrect programs in the first place, leading to improved performance. In addition, the user can now *whitelist* variables that are allowed to be changed, rather than having to blacklist the (potentially many) variables that cannot be changed.

The mutability constraint on an SSG is a single bit, either enabled or disabled. When disabled, the mutability constraint is a no-op, and is always satisfied. When enabled, the SSG must evaluate to a value that is allowed to be mutated. The user must specify for each terminal whether it is allowed to be mutated. We must also add more information to our components. For the purposes of mutability constraints, a component can have two properties:



(a) No sharing of subgraphs.

(b) Memoize Create-SSG. This SSG cannot represent `(+ x y)`.

(c) Use the memoized SSG only when it is provably safe.

Figure 5: Three different approaches to sharing subgraphs. In all cases, we want to represent any program composed of a single call to `vec-ref`, `+` and `-`, with the vector variables `v` and `w`, and the integer variables `x` and `y`. Note that these SSGs are shown before merging for the sake of exposition.

1. Writer: These components mutate one or more of their arguments. Regardless of the mutability constraint on this component, the mutability constraint on the mutated argument must be enabled. For example, `vector-set!` is a writer component that always mutates its first argument.
2. Reader: These components return a value read from one of their arguments. In this case, if the mutability constraint is enabled for the reader component, then the mutability constraint must also be enabled for the argument from which the reader component reads. For example, if the mutability constraint is enabled and we are considering the `vector-ref` component, then the first argument must have the mutability constraint enabled, but every other argument will have it disabled.

It is possible for a component to be both a reader and a writer. If a component is not a reader, then it can never satisfy the enabled mutability constraint, since it cannot return a value allowed to be mutated.

4.5 Sharing Subgraphs

As presented so far, the generated SSGs can have many identical subgraphs. This is illustrated in Figure 5a, where we are choosing between `vector-ref`, `+`, and `-`, and there are five different subgraphs that all encode the SSG for programs of a certain depth that produce integers. Since these subgraphs are all encoding the same program space, we might expect that we can do better by sharing subgraphs. If we replace two separate subgraphs t_1 and t_2 by two copies of t_1, the two subgraphs will now share the boolean variables that control the choice nodes, reducing the number of boolean variables in the SMT formula generated by Rosette and allowing conflict clauses learned by the solver to apply to both subgraphs. In particular, with separate subgraphs, if we have choice nodes $c_1 \in t_1$ and $c_2 \in t_2$, the solver can choose the values for c_1

and c_2 independently. If we instead have two copies of c_1, the solver can only choose a value of c_1, and the subgraphs must necessarily make the same choice.

A simple approach for sharing structure would be to simply memoize CREATE-SSG, so that whenever it is called with the same arguments, it returns the previously generated SSG, as illustrated in Figure 5b. However, this optimization is too strong, and eliminates programs that we care about. Since all of the subgraphs are now identical, they must make the same choice at each node, and so the SSG can no longer represent the program `(+ x y)` – it is forced to instead choose `(+ x x)` or `(+ y y)`. We can still take advantage of some memoization, as illustrated in Figure 5c. If the SSG is generated in a bottom up fashion, then equivalent grandchildren of a choice node (that don't share a parent) can be shared without losing expressiveness. Intuitively, for any two nodes t_1 and t_2 that have the same grandparent but different parents p_1 and p_2, any valuation of the SSG must choose at most one out of p_1 and p_2 to include in the AST, and so at most one out of t_1 and t_2 can be included in the AST, and so we don't lose expressiveness by requiring t_1 and t_2 to be the same.

4.6 Straight-Line Grammar

There are still a few issues with the construction of SSGs:

- There is no easy way to reuse common subexpressions. If the same expression is used in two different parts of the same AST, the synthesizer must discover it twice.
- The majority of the programs encoded in the SSG are atypical. Most ASTs that the SSG encodes are very bushy, that is, most of the leaves are at or near the maximum depth. However, in realistic programs, there are typically only a few leaves near the maximum depth, while most leaves are quite close to the root.

We can solve these problems by introducing temporary variables, which can store the results of small subcomputations. The SSG then consists of a series of temporary variable definitions whose values are given by small (depth 1) sub-SSGs, followed by a single small expression that defines the return value. Now, in order to increase the size of the search space, we add in extra temporary variable definitions, instead of increasing the maximum depth of the SSG.

We have to figure out what the types of the temporary variables should be, so that the temporary variables to be used in subsequent parts of the SSG, and CREATE-SST can be called with the right type as its argument. However, we do not know a priori what type of temporary variables we need. We solve this in a similar way as with polymorphism – since we have a small number of types, we simply create a temporary variable for each type. This results in rather long and cumbersome programs, but most of the generated code is dead code and is removed in a postprocessing step.

We hypothesize that this is more representative of realistic programs than bushy trees, and so will provide a useful inductive bias so that the synthesizer has to consider a smaller set of programs. Any program can be converted to the form of a sequence of temporary variable definitions. However, if we consider the programs represented by bushy SSGs, the common bushy ASGs will be converted to very long sequences of temporary variables (since they have many intermediate nodes), whereas the more realistic programs with only a few leaves at high depth will be converted to relatively short sequences. So, we should expect that putting a bound on the temporary variables instead of the maximum depth would constrain us more closely to realistic programs. We also test this hypothesis experimentally and find some evidence that it is true in Section 6.2.1.

4.7 Extensibility

A key feature of Algorithm 1 is that it makes minimal demands on the components, which makes it easy to add new components. If the new component is a procedure, we only need to specify its type and the arguments it reads or writes. An example is given in Figure 10 in the appendix.

If the component is not a procedure, then it depends on the component in question. Some special forms are just as easy to specify. For example, `if` can be specified in the same way as a procedure. In general, the component must implement GET-CHILD-CONSTRAINTS for type constraints, mutability constraints and any syntax constraints it may have.

5 Case Study: Syncro

To demonstrate the utility of our algorithm, we implement SYNCRO, a tool for synthesis of incremental operations. Incremental computation allows the value of a program to be updated efficiently upon small changes to its inputs, leading to asymptotic speedups which are crucial for reasonable performance in many domains. However, an incremental program is often more complicated than its non-incremental counterpart, so we would like to generate an incremental program from a non-incremental one.

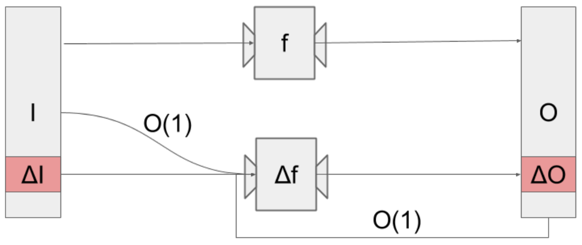


Figure 6: The problem setup for incrementality. We are given a function f that computes output O from input I. When we make a change ΔI to I, we want an incremental update function Δf that produces ΔO, the change to O.

Consider again the permutation example from Section 2. This is an example of an incremental problem – we have a program that computes the inverse of a permutation, and we make a small change to the input by transposing two elements in the permutation, and we now want to update the inverse efficiently. The full specification of this problem in SYNCRO is given in Figure 7. Note that the specification makes no reference to grammars, SSGs, or correctness conditions.

```
(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))
```

Figure 7: SYNCRO specification for the permutation problem.

Given this specification, we want to find a fast *in-place* 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:

$$\exists \Delta f\ \forall I, \Delta I:\ \ O + \Delta f(I, \Delta I, O) = f(I + \Delta I) \tag{1}$$

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!`), Δf is the desired update function, and $+$ denotes the "apply change" operation. This is illustrated in Figure 6.

SYNCRO generates a program from the specification that performs synthesis, shown at a high level in Figure 8. The correctness condition is defined by Equation 1. In order to encode the $\forall$ quantifiers, we need to generate symbolic versions of I and ΔI that Rosette can then quantify over. This is done by implementing a function `sym` that given a type can produce a symbolic value encoding all concrete values of that type up to a certain size.

At the top level call to CREATE-SSG, the list of terminals includes all of the variables defined in the specification (for which we have type information), all of which are marked not mutable, except for the one variable we do want to mutate (`inverse` in this case). We use a predefined set of default components that includes `vector-ref` and `vector-set!`, which are enough to solve the permutation problem, though the user can add more components if she wishes. The generated SSG should return type `void`, since we care about statements with side effects. The maximum depth of the generated SSG can be controlled through a command line flag.

```
(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 8: Pseudocode for the Rosette program generated from the specification in Figure 7. 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 up to a certain size.

6 Evaluation

We performed an experimental evaluation of our algorithm to test its expressiveness and scalability compared to existing alternatives, and to evaluate each of our proposed optimizations. We sought to answer the following questions:

1. Is it scalable? Can we solve new challenging benchmarks that prior work struggles with?

2. Do each of the optimizations result in performance improvements?

3. Does a straight-line grammar provide a better inductive bias, populating the search space with more realistic programs?

4. Does our algorithm work well in a domain-specific tool, where the parameters may not be set perfectly?

Our experiments provide affirmative answers to all of the questions above. All experiments were carried out on a 2 core, 2.00 GHz, Intel Core i7 processor, 8GB RAM, running the Ubuntu operating system, with a timeout of 1 hour.

Benchmark	***Description***	$\|Sol\|$	$\|LSpace\|$	$\|TSpace\|$	***Time(s)***
	Incremental updates for:				
Skosette	Manipulation of an array of booleans.	10	2ˆ18	2ˆ13	0.2
Permutation	Permutation inverse (see Figure 1)	16	2ˆ39	2ˆ28	0.3
Exists	Data structure that checks $\exists x : \phi(x)$ given $\|\{x : \phi(x)\}\|$	50	2ˆ72	2ˆ80	0.6
	Dynamic programming recurrences:				
Count change	Count the number of ways to make change.	26	2ˆ75	2ˆ395	2685.1
Edit distance	Compute edit distance between two strings.	51	2ˆ525	2ˆ7022	TO

Table 1: The suite of expression synthesis benchmarks.

6.1 Comparison to Prior Work

We consider Sketch [19] and Bonsai [7]. Sketch is a system that is primarily aimed at performing arbitrary program synthesis, and so is a natural point to compare against. Bonsai is built on top of Rosette and is aimed at detecting bugs in type systems. It proposes a new way of encoding spaces of programs called Bonsai trees, which could be used instead of our symbolic syntax graphs. Note that Bonsai trees were designed to make symbolic evaluation and merging of program spaces efficient, whereas in program synthesis the bottleneck is typically the solver time. We include Bonsai as a point to compare against anyway because it is the only other work we know of that builds a new encoding for program spaces in Rosette.

We use the set of benchmarks in Table 1. We selected three benchmarks from our suite of incremental benchmarks with a range of difficulties and synthesized update rules by providing input-output examples. We also use two dynamic programming problems, in which we synthesize the required recurrence relation. For all benchmarks, we developed a minimal solution by hand, and report the number of nodes as the solution size. We report the handwritten solution instead of the synthesized one because the solver may not produce minimal solutions. We also report program space sizes (LSpace and TSpace), discussed further in Section 6.2.1.

For each benchmark, we manually developed a grammar for Bonsai and Sketch, using the same components as in our algorithm. These grammars ensured type safety, and in the case of Sketch, we wrote the grammars so that they would share subgraphs where it was natural to do so, though our algorithm shares more subgraphs because it can optimize across all components at the same depth, even ones that produce different types. Unfortunately, Bonsai does not give us enough control over the grammar to share subtrees. The depth of each grammar was set to be the minimum number for which we actually could find a solution.

The results are given in Figure 9a, and show that our algorithm outperforms Sketch and Bonsai on all benchmarks except one, where all three algorithms time out. We solve one new benchmark, count-change, that neither Sketch nor Bonsai are able to solve.

6.2 Effect of each Optimization

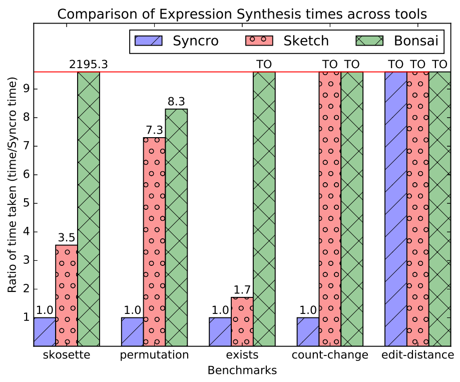
In order to quantify the benefits of optimizations, we modified our algorithm to selectively disable type analysis, mutability analysis, subtree sharing, and straight line grammar. We evaluated the expression synthesis benchmarks from Table 1 once with each optimization disabled.

The results are shown in Figure 9b. We can see that unsurprisingly the most important optimization is the type analysis, which eliminates a huge set of ill-typed programs. Focusing primarily on the larger benchmarks, we can see that as the benchmarks get larger, the straight-line grammar becomes more useful; we discuss this phenomenon below. The sharing of subgraphs does not benefit the smaller benchmarks but it was necessary to solve count-change, and so is likely important for larger benchmarks.

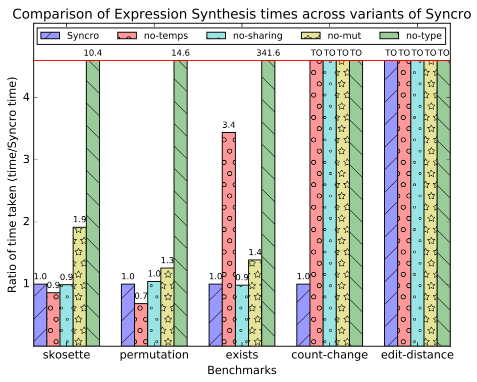
6.2.1 Inductive Bias of the Straight Line Grammar

In Section 4.6, we hypothesized that the straight line grammar represents more realistic programs, compared to Create-sst which consists mainly of bushy ASTs. To test this, in Table 1 we measure program space sizes *modulo integer constants*; that is, we consider the programs `(+ x 1)` and `(+ x 2)` to be the same. We do this because otherwise the program space size would be dominated by the number of possible choices of values for integer constants, which is not useful for comparing the two grammars. LSpace is the program space generated by the straight-line grammar while TSpace is the program space generated when using Create-sst. In both cases, we choose the smallest instantiation of the grammar in which synthesis still succeeds. In cases where synthesis never succeeded due to timeout, we report the size of the smallest instantiation of the grammar that we know contains a correct solution.

For small benchmarks (depth of around 2), the two approaches are roughly comparable, because the program spaces are small enough that any variations are minor and unimportant. However, for larger benchmarks, the straight line grammar creates smaller program spaces and so is much more performant. This is because the bushiness of Create-sst causes problems. For example, the optimal solution for count change has only one node at depth 4 with children, which forces us to increase the depth from 4 (program space size 2^{150}) to 5 (program space size 2^{395}). The vast majority of these 2^{395} programs have nearly all of their leaves at depth 5, unlike the optimal solution. In contrast, the straight-line grammar produces a space of size 2^{75}. Similarly for edit distance we are forced to go from a space of size 2^{1071} to one of size 2^{7022}, even though most of the leaf nodes in the solution have depth less than 5. Overall, this suggests that the straight line grammar does provide a useful inductive bias.



(a) Times taken to solve the expression synthesis benchmarks, relative to our algorithm's time.



(b) Times taken by variants of our algorithm, relative to our algorithm with all optimizations enabled.

Figure 9: Evaluation graphs.

6.3 Performance of Syncro

For evaluation of Syncro, we used the set of benchmarks described in Table 2. Unlike with expression synthesis, we set all parameters to be the same across all benchmarks, since in a

Benchmark	***Description***	***Size***	***Time (s)***
Permutation	Permutation inverse (see Figure 1)	2^221	153.2
Counting	Count the number of objects assigned to a category, when category assignments change.	2^324	9.2
Exists	Checks $\exists x : \phi(x)$ given counts $\lvert\{x : \phi(x)\}\rvert$	2^713	19.0
Swift	Compute Markov Blankets in a Contingent Bayes Net when adding edges (see [31]).	2^283	56.8
Grades	Compute grades from exam scores.	2^194	93.0
Record	Like Grades, but using structs to represent students.	2^226	237.2
Set-union	Compute the union of two sets.	2^144	1.5

Table 2: Selected benchmarks used for incremental update synthesis, their description, size of search space and time taken by SYNCRO. Complete table in Table 3 in the appendix.

realistic setting the user would write an incremental program and is not likely to tune the underlying parameters away from the defaults. We use the straight-line grammar with all optimizations enabled. The *Size* column lists the size of the constructed program space. The *Time* column lists the total time taken by SYNCRO to solve the benchmark.

The search spaces for these benchmarks are much larger, even when solving the same problem (Skosette, permutation, and exists). This is partly because the SSG is no longer the minimal SSG required to solve the benchmark, but primarily it is because we use the full set of default components, and there are many extra variables that are defined in the specification that are used in the SSG but are not actually needed in the synthesized program. The solver must spend additional time determining that these programs are not useful.

Despite the much larger search spaces, our algorithm is still able to find solutions in a reasonable amount of time, solving every benchmark in under 5 minutes. Contrast this with count-change from Table 1, which has a much smaller search space of size 2^{75} but takes over half an hour to solve, because there are no irrelevant variables or components. This illustrates a major advantage of solving the entire problem using an SMT solver – it can very quickly determine which portions of the formula are irrelevant and ignore those. If we built a domain-specific synthesis algorithm, it seems distinctly more difficult to ensure that this property holds.

7 Related Work

Expression synthesis is an active area of research, with many algorithms for synthesis of functional programs [26, 11, 17]. Sketch [19] and Rosette [29] can be used for expression synthesis of both functional and imperative programs. Our work makes it easier and faster to synthesize imperative programs in Rosette, and similar ideas could be used for Sketch as well. In particular, the straight line grammar should provide similarly good speedups for Sketch.

Bonsai [7] also builds on top of Rosette to introduce a new encoding for the space of programs called Bonsai trees, that allow for efficient merging, leading to efficient symbolic evaluation, which can then be used to check the soundness of type systems. However, Bonsai trees do not work well for expression synthesis, primarily because the bottleneck is now solver time and not symbolic execution. Metasketches [6] allow users to solve the optimal synthesis problem, where we want to find a correct program that minimizes a cost. They also allow us to parallelize the solver search by cutting up the program space into many parts, and searching over each part in parallel. Each individual part of a metasketch is a symbolic syntax graph, and so in future work we would like to combine the two approaches in order to get the best of both worlds.

Component-based synthesis was introduced in [16] and has been used recently for synthesis of API calls [10] and table transformations [9]. Besides component-based synthesis, other frameworks have also been developed to generalize domain-specific synthesis algorithms, by separating the generic synthesis algorithm from the domain-specific details [27, 5]. Our work can be seen as a way to provide general-purpose component-based synthesis, with very minimal requirements on the information needed for each component (usually just a type).

Synthesis of incremental update rules has been briefly studied in [15], where it is used as an example application for an inductive synthesis algorithm. However, the algorithm works on statements in logic which are then translated into imperative code, so it would not work for general-purpose imperative expression synthesis without axiomatizing an imperative programming language into first-order logic.

General dynamic incremental computation for functional programs was first introduced with self-adjusting computation [1]. Later work generated it automatically with type annotations [8] and applied it to imperative programs [2]. It continues to be improved today in Adapton [13, 12]. It works by propagating changes to the inputs through a dependency graph, though differential dataflow [24] works with deltas instead. These systems get big asymptotic speedups on large, complex programs that SYNCRO cannot scale to. However, maintaining internal data structures leads to large runtime overhead, and they usually do not find the optimal solution. SYNCRO can work with both changes in inputs as well as new values, and has more context and knowledge since the program is represented as an SMT formula. For example, the permutation solution in Figure 1 is only correct because of the invariant that the input array must be a permutation, which other tools would not be able to use or understand.

There are also several techniques for static incremental computation, that rely on static analysis of programs [25, 20, 22]. In the databases literature, researchers focus on solving the incremental view maintenance problem, which aims to keep the view of an underlying table up to date as the data is changed [3, 18]. However, these systems cannot take into account context from the program or hints from the user, and they too would not be able to solve permutation (Figure 7) since they cannot use the invariant that the input array must be a permutation.

8 Conclusion and Future Work

We presented an algorithm for automatically constructing symbolic syntax graphs (SSGs) from components with type information, leading to efficient expression synthesis. We used this algorithm to implement SYNCRO, which can synthesize incremental update rules for interesting programs, including one new benchmark that prior work times out on.

An immediate direction for future work is to integrate our approach with metasketches [6] to combine the advantages of both. Another direction is to explore the straight-line encoding in more detail – while it does provide a useful inductive bias, decreasing the size of the search space, the solver finds it harder to reason about this encoding of the program. How can we get the benefit of better inductive bias without losing performance to suboptimal encodings?

References

[1] Umut A. Acar (2005): *Self-adjusting Computation*. Ph.D. thesis, Pittsburgh, PA, USA. AAI3166271.

[2] Umut A. Acar, Amal Ahmed & Matthias Blume (2008): *Imperative Self-adjusting Computation*. In: *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, ACM, New York, NY, USA, pp. 309–322, doi:10.1145/1328438.1328476. Available at `http://doi.acm.org/10.1145/1328438.1328476`.

[3] Yanif Ahmad, Oliver Kennedy, Christoph Koch & Milos Nikolic (2012): *DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views*. *Proc. VLDB Endow.* 5(10), pp. 968–979, doi:10.14778/2336664.2336670. Available at `http://dx.doi.org/10.14778/2336664.2336670`.

[4] David M. Blei, Andrew Y. Ng & Michael I. Jordan (2003): *Latent Dirichlet Allocation*. *J. Mach. Learn. Res.* 3, pp. 993–1022. Available at `http://dl.acm.org/citation.cfm?id=944919.944937`.

[5] Rastislav Bodík, Kartik Chandra, Phitchaya Mangpo Phothilimthana & Nathaniel Yazdani (2017): *Domain-Specific Symbolic Compilation*. In Benjamin S. Lerner, Rastislav Bodík & Shriram Krishnamurthi, editors: *2nd Summit on Advances in Programming Languages (SNAPL 2017)*, *Leibniz International Proceedings in Informatics (LIPIcs)* 71, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, pp. 2:1–2:17, doi:10.4230/LIPIcs.SNAPL.2017.2. Available at `http://drops.dagstuhl.de/opus/volltexte/2017/7133`.

[6] James Bornholt, Emina Torlak, Dan Grossman & Luis Ceze (2016): *Optimizing Synthesis with Metasketches*. In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, ACM, New York, NY, USA, pp. 775–788, doi:10.1145/2837614.2837666. Available at `http://doi.acm.org/10.1145/2837614.2837666`.

[7] Kartik Chandra & Rastislav Bodík (2017): *Bonsai: Synthesis-Based Reasoning for Type Systems*. *CoRR* abs/1708.00551. Available at `http://arxiv.org/abs/1708.00551`.

[8] Yan Chen, Joshua Dunfield & Umut A. Acar (2012): *Type-directed Automatic Incrementalization*. In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, ACM, New York, NY, USA, pp. 299–310, doi:10.1145/2254064.2254100. Available at `http://doi.acm.org/10.1145/2254064.2254100`.

[9] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig & Swarat Chaudhuri (2017): *Component-based Synthesis of Table Consolidation and Transformation Tasks from Examples*. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, ACM, New York, NY, USA, pp. 422–436, doi:10.1145/3062341.3062351. Available at `http://doi.acm.org/10.1145/3062341.3062351`.

[10] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig & Thomas W. Reps (2017): *Component-based Synthesis for Complex APIs*. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, ACM, New York, NY, USA, pp. 599–612, doi:10.1145/3009837.3009851. Available at `http://doi.acm.org/10.1145/3009837.3009851`.

[11] Jonathan Frankle, Peter-Michael Osera, David Walker & Steve Zdancewic (2016): *Example-directed Synthesis: A Type-theoretic Interpretation*. In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, ACM, New York, NY, USA, pp. 802–815, doi:10.1145/2837614.2837629. Available at `http://doi.acm.org/10.1145/2837614.2837629`.

[12] Matthew A. Hammer, Joshua Dunfield, Kyle Headley, Nicholas Labich, Jeffrey S. Foster, Michael Hicks & David Van Horn (2015): *Incremental Computation with Names*. In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, ACM, New York, NY, USA, pp. 748–766, doi:10.1145/2814270.2814305. Available at `http://doi.acm.org/10.1145/2814270.2814305`.

[13] Matthew A. Hammer, Khoo Yit Phang, Michael Hicks & Jeffrey S. Foster (2014): *Adapton: Composable, Demand-driven Incremental Computation*. In: *Proceedings of the 35th ACM SIGPLAN*

Conference on Programming Language Design and Implementation, PLDI '14, ACM, New York, NY, USA, pp. 156–166, doi:10.1145/2594291.2594324. Available at `http://doi.acm.org/10.1145/2594291.2594324`.

[14] Roger Hoover (1992): *Alphonse: Incremental Computation As a Programming Abstraction.* In: *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, ACM, New York, NY, USA, pp. 261–272, doi:10.1145/143095.143139. Available at `http://doi.acm.org/10.1145/143095.143139`.

[15] Shachar Itzhaky, Sumit Gulwani, Neil Immerman & Mooly Sagiv (2010): *A Simple Inductive Synthesis Methodology and Its Applications. SIGPLAN Not.* 45(10), pp. 36–46, doi:10.1145/1932682.1869463. Available at `http://doi.acm.org/10.1145/1932682.1869463`.

[16] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia & Ashish Tiwari (2010): *Oracle-guided Component-based Program Synthesis.* In: *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, ACM, New York, NY, USA, pp. 215–224, doi:10.1145/1806799.1806833. Available at `http://doi.acm.org/10.1145/1806799.1806833`.

[17] Etienne Kneuss, Ivan Kuraj, Viktor Kuncak & Philippe Suter (2013): *Synthesis Modulo Recursive Functions.* In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications*, OOPSLA '13, ACM, New York, NY, USA, pp. 407–426, doi:10.1145/2509136.2509555. Available at `http://doi.acm.org/10.1145/2509136.2509555`.

[18] Christoph Koch, Daniel Lupei & Val Tannen (2016): *Incremental View Maintenance For Collection Programming.* In: *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '16, ACM, New York, NY, USA, pp. 75–90, doi:10.1145/2902251.2902286. Available at `http://doi.acm.org/10.1145/2902251.2902286`.

[19] Armando Solar Lezama, Leo Harrington & Rastislav Bodik Chair (2008): *Program Synthesis By Sketching.* Technical Report.

[20] Yanhong A. Liu (2000): *Efficiency by Incrementalization: An Introduction. Higher Order Symbol. Comput.* 13(4), pp. 289–313, doi:10.1023/A:1026547031739. Available at `http://dx.doi.org/10.1023/A:1026547031739`.

[21] Yanhong A. Liu & Scott D. Stoller (2009): *From Datalog Rules to Efficient Programs with Time and Space Guarantees. ACM Transactions on Programming Languages and Systems* 31(6), pp. 1–38.

[22] Yanhong A. Liu, Scott D. Stoller, Michael Gorbovitski, Tom Rothamel & Yanni Ellen Liu (2005): *Incrementalization Across Object Abstraction.* In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, ACM, New York, NY, USA, pp. 473–486, doi:10.1145/1094811.1094848. Available at `http://doi.acm.org/10.1145/1094811.1094848`.

[23] John McCarthy & Patrick J Hayes (1981): *Some philosophical problems from the standpoint of artificial intelligence.* In: *Readings in artificial intelligence*, Elsevier, pp. 431–450.

[24] Frank D McSherry, Rebecca Isaacs, Michael A Isard & Derek G Murray (2013): *Differential dataflow.* CIDR '13.

[25] R. Paige (1981): *Formal Differentiation: A Program Synthesis Technique.* Computer Science Series, UMI Research Press. Available at `https://books.google.com/books?id=xNImAAAAMAAJ`.

[26] Nadia Polikarpova, Ivan Kuraj & Armando Solar-Lezama (2016): *Program Synthesis from Polymorphic Refinement Types.* In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, ACM, New York, NY, USA, pp. 522–538, doi:10.1145/2908080.2908093. Available at `http://doi.acm.org/10.1145/2908080.2908093`.

[27] Oleksandr Polozov & Sumit Gulwani (2015): *FlashMeta: A Framework for Inductive Program Synthesis.* In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, ACM, New York, NY, USA,

pp. 107–126, doi:10.1145/2814270.2814310. Available at `http://doi.acm.org/10.1145/2814270.2814310`.

[28] Ian Porteous, David Newman, Alexander Ihler, Arthur Asuncion, Padhraic Smyth & Max Welling (2008): *Fast Collapsed Gibbs Sampling for Latent Dirichlet Allocation.* In: *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '08, ACM, New York, NY, USA, pp. 569–577, doi:10.1145/1401890.1401960. Available at `http://doi.acm.org/10.1145/1401890.1401960`.

[29] Emina Torlak & Rastislav Bodík (2014): *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, ACM, New York, NY, USA, pp. 530–541, doi:10.1145/2594291.2594340. Available at `http://doi.acm.org/10.1145/2594291.2594340`.

[30] Chenglong Wang, Alvin Cheung & Rastislav Bodik (2017): *Synthesizing Highly Expressive SQL Queries from Input-output Examples.* *SIGPLAN Not.* 52(6), pp. 452–466, doi:10.1145/3140587.3062365. Available at `http://doi.acm.org/10.1145/3140587.3062365`.

[31] Yi Wu, Lei Li, Stuart Russell & Rastislav Bodík (2016): *Swift: Compiled Inference for Probabilistic Programming Languages.* In: *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, pp. 3637–3645. Available at `http://www.ijcai.org/Abstract/16/512`.

[32] Limin Yao, David Mimno & Andrew McCallum (2009): *Efficient Methods for Topic Model Inference on Streaming Document Collections.* In: *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, ACM, New York, NY, USA, pp. 937–946, doi:10.1145/1557019.1557121. Available at `http://doi.acm.org/10.1145/1557019.1557121`.

9 Appendix

Figure 10 shows all of the code required to implement a new operator that increments an element of a vector.

```
;; Rosette implementation of the procedure
(define (vec-incr! v i)
  (vector-set! v i (+ 1 (vector-ref v i))))

;; Type of the operator
(define a (Type-var (Index-type)))
(define type
  (-> (list (Vector-type a (Integer-type)) a)
      (Void-type)
      #:write-index 0))

;; Create the operator
(define-lifted [vec-incr! vec-incr!^ type])
```

Figure 10: Implementation of the `vec-incr!` component.

Benchmark	***Description***	***Size***	***Time (s)***
Skosette	Manipulation of an array of booleans.	2^265	32.7
Permutation	Permutation inverse (see Figure 1)	2^221	153.2
Permutation 2	Permutation with a new implementation of transpose.	2^221	186.2
Counting 1	Count the number of objects assigned to a category, when category assignments change.	2^324	9.2
Counting 2	Like Counting 1, but with two kinds of categories.	2^428	19.9
Exists	Checks $\exists x : \phi(x)$ given counts $\|\{x : \phi(x)\}\|$	2^713	19.0
Swift	Compute Markov Blankets in a Contingent Bayes Net when adding edges (see [31]).	2^283	56.8
Grades	Compute grades from exam scores.	2^194	93.0
Record	Like Grades, but using structs to represent students.	2^226	237.2
Swap grades	Like Grades, but considering updates where two student's grades are swapped.	2^211	3.8
Set-size	Compute the size of a set.	2^209	42.9
Set-union	Compute the union of two sets.	2^144	1.5
Set-difference	Compute the difference between two sets.	2^144	1.5
Sum	Compute the sum of an array of numbers.	2^211	30.9

Table 3: Complete set of benchmarks used for incremental update synthesis, their description, size of search space and time taken by SYNCRO.