Scalable Synthesis with Symbolic Syntax Graphs

Rohin Shah, **Sumith Kulal**, Ras Bodik
UC Berkeley, IIT Bombay and UW

18 July 2018, Oxford UK

[Solar-Lezama et al. ASPLOS06]
Combinatorial Sketching for Finite Programs

“

Sketching also promises to enable complex implementations that may be too tedious to develop and maintain without automatic synthesis of low-level detail.

… but what if the synthesis is too tedious itself?

1

Let’s dive right in

Given a new domain, how does one synthesize programs in it?

IN ORDER OF DECREASING EFFORT

- Build a new algorithm from scratch
- Specialize a generic meta-algorithm with domain specific operators
- Leverage an existing framework that provides a general-purpose synthesis algorithm

Using an existing framework means carefully constructing your grammar

IN THIS PAPER, WE PRESENT

A novel algorithm

that automatically constructs symbolic syntax graphs from specified components enforcing constraints like type safety, etc.

A case study

where we implement a system for synthesizing incremental operations on data-structures

2

Let’s look at an example

Consider the problem of incrementally maintaining inverse of a permutation

HOW DO WE DEFINE THE INVERSE GIVEN A PERMUTATION?

permutation	3	2	5	7	0	1	4	6
indices	0	1	2	3	4	5	6	7

inverse	4	5	1	0	6	2	7	3
indices	0	1	2	3	4	5	6	7

WE NEED TO SYNTHESIZE A FIX FOR INVERSE GIVEN TRANSPOSE

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

How do we synthesize `(fix-inverse! i j)`?

3

Let's incrementally construct grammars

We are trying to synthesize `(fix-inverse! i j)` function

EASIEST GRAMMAR TO START OFF WITH

E -> bin-proc E E | tern-proc E E E | atom

bin-proc -> begin | vec-ref | + | -

tern-proc -> vec-set!

atom -> variable | constant

problem? Yes, please!

Leads to bad programs like `(vec-ref 3 3)`

LET'S FIX THE TYPES IN THIS GRAMMAR

E -> int-gen | vec-gen | void-stmt

int-gen -> vec-ref vec-gen int-gen | +/- int-gen int-gen | int-atom

vec-gen -> vec-atom

void-stmt -> vec-set! vec-gen int-gen int-gen

int-atom -> int-var | int-const

vec-atom -> vec-var | vec-const

problem? **Yes, please!**

Can undo your initial operation (next slide)

THE GRAMMAR CAN SYNTHESIZE THE FOLLOWING

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

Need to encode which variables are mutable => more production rules☹☹☹

SEVERAL OPTIMIZATION ON TOP CONSTRUCTING THE RIGHT GRAMMAR

Sharing subgraphs

It may be possible to encode the symbolic syntax graph in fewer boolean variables make it easier for the SMT solver.

Straight-Line Grammar

Encodes programs in a straight-line format instead of the typical bushy-tree format, which gives an inductive bias towards realistic programs.

Extensibility

Makes it easy to add new operators. Just describe the type signature of the component.

CHECKPOINT

Manual constructing high-performant grammars is tedious error-prone task, we automate it!

OUR SSG CONSTRUCTION ALGORITHM: CREATE-SSG(F, V, C, d)

F -> is a list of components

V -> is a list of terminals

C -> is a list of constraints (type, mutability, etc.)

d -> is a maximum depth of the SSG

Intuition: Choose from components and terminals according to the constraints, recursively construct SSG for children till depth by passing appropriate constraints down!

LET'S EXPLORE THE CORRECT BRANCH OF PERMUTATION

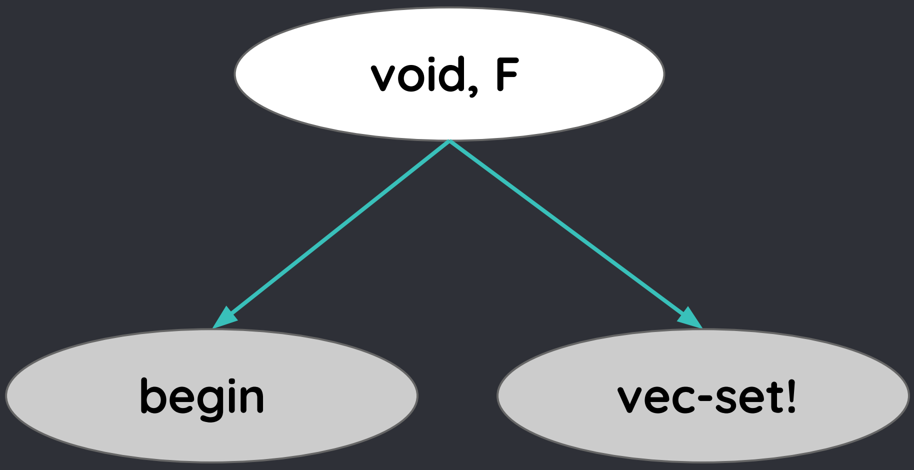
CREATE-SSG(

F -> begin, vec-ref, vec-set!, +, -,

V -> inverse (mutable), perm, i, j (immutable),

C -> type-void, F,

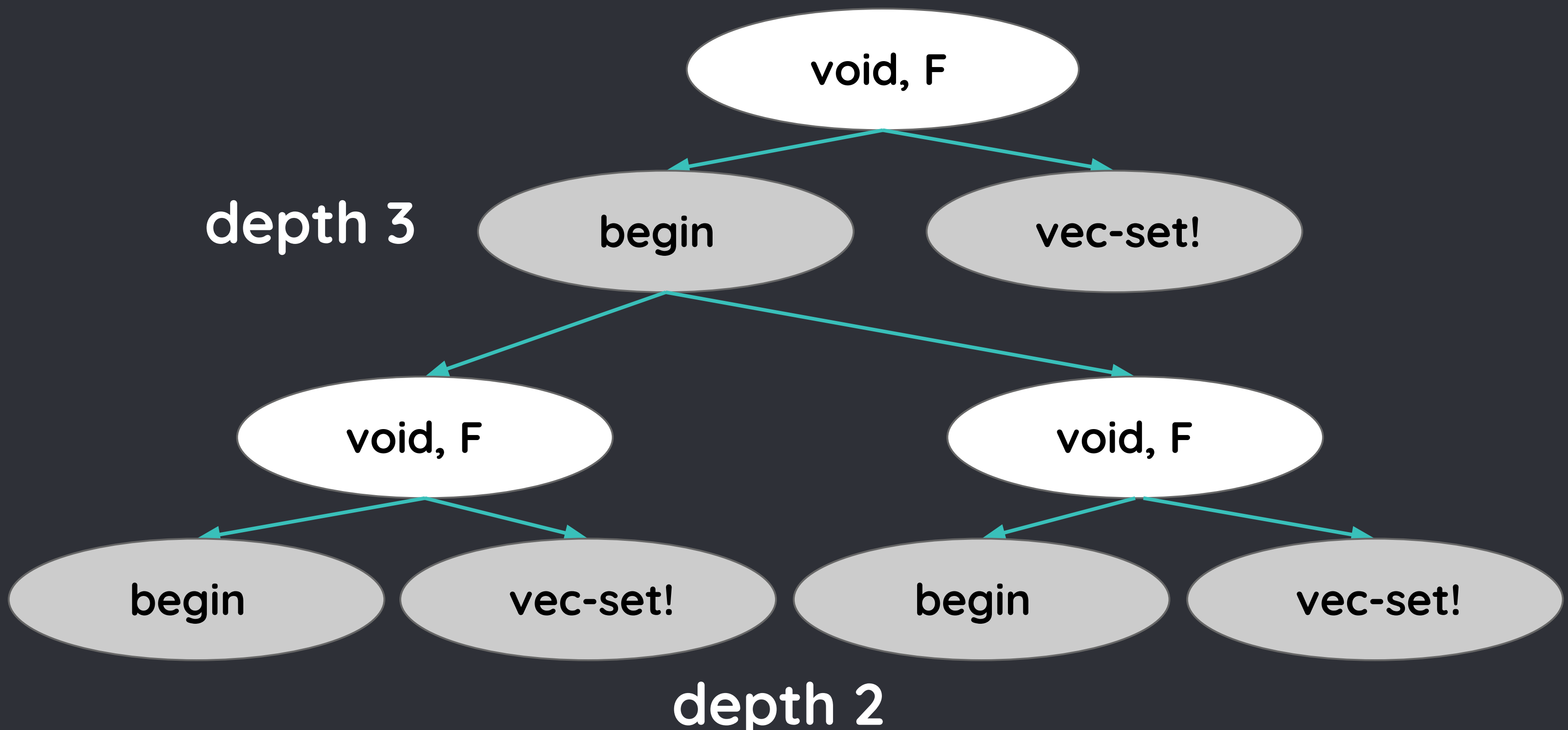
d -> 3)



depth 3

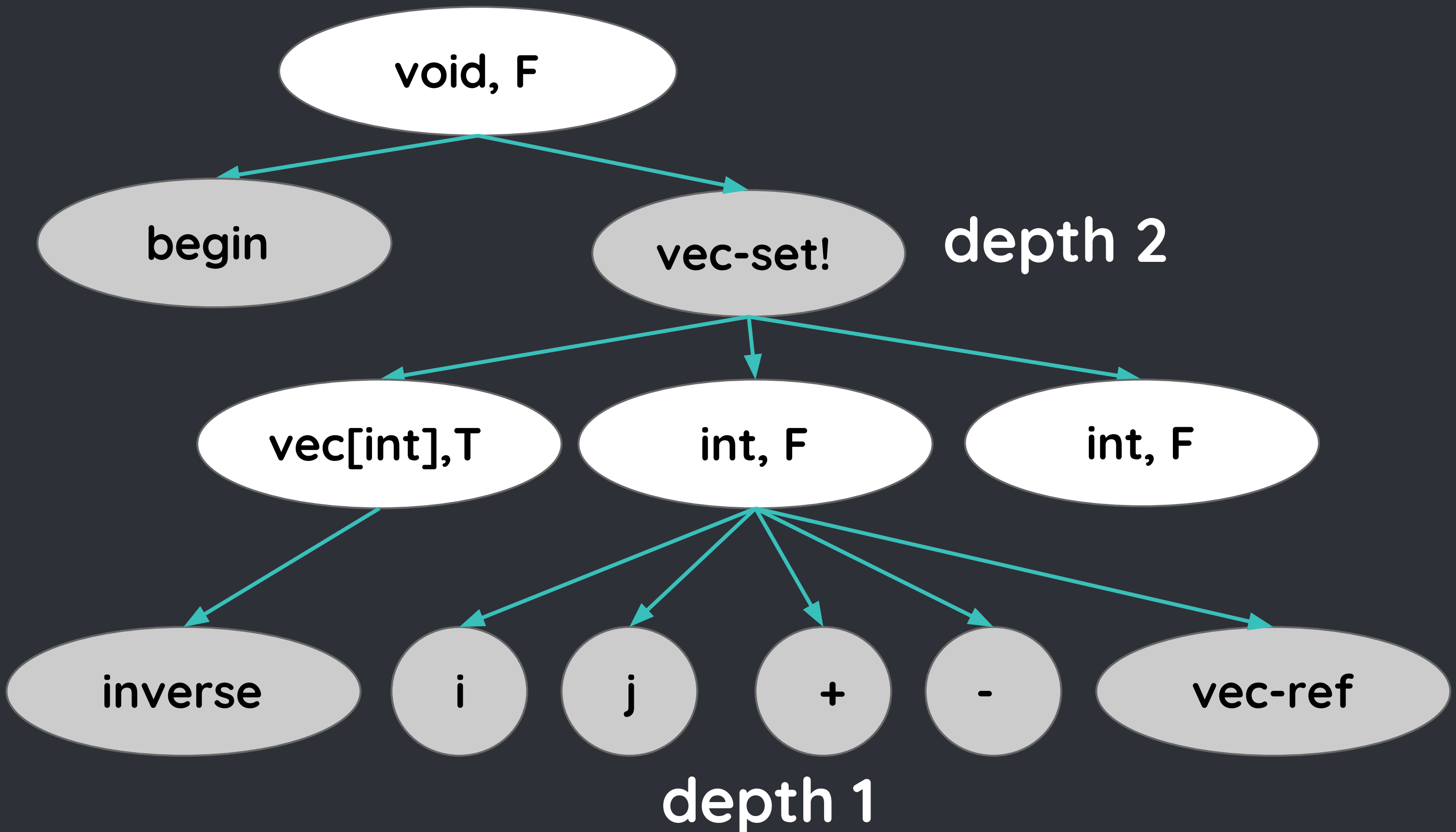
LET'S EXPLORE BEGIN BRANCH OF THE SSG

begin: void, F -> void, F -> void, F



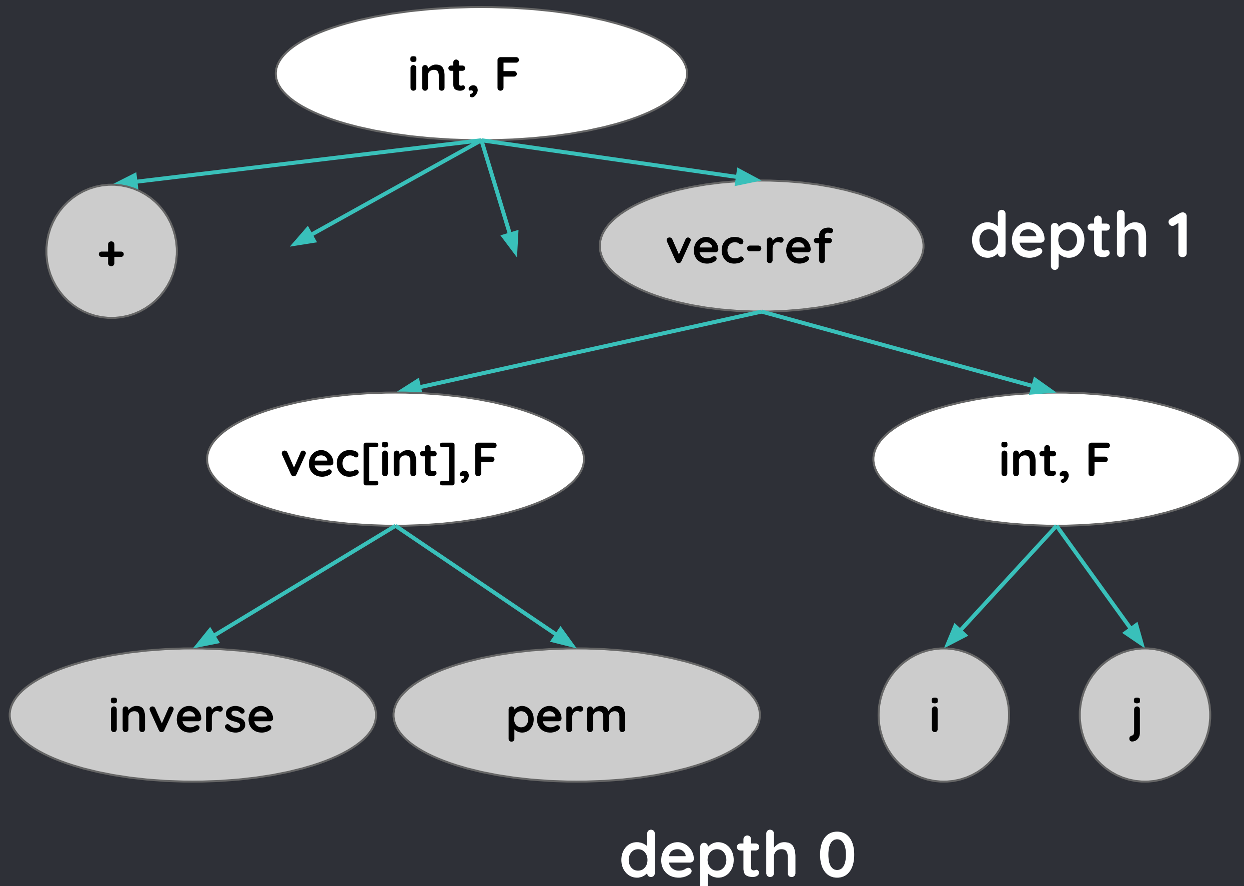
LET'S EXPLORE VEC-SET! BRANCH OF THE SSG

vec-set!: vec[int], T -> int, F -> int, F -> void, F



LET'S EXPLORE VEC-REF! BRANCH OF THE SSG

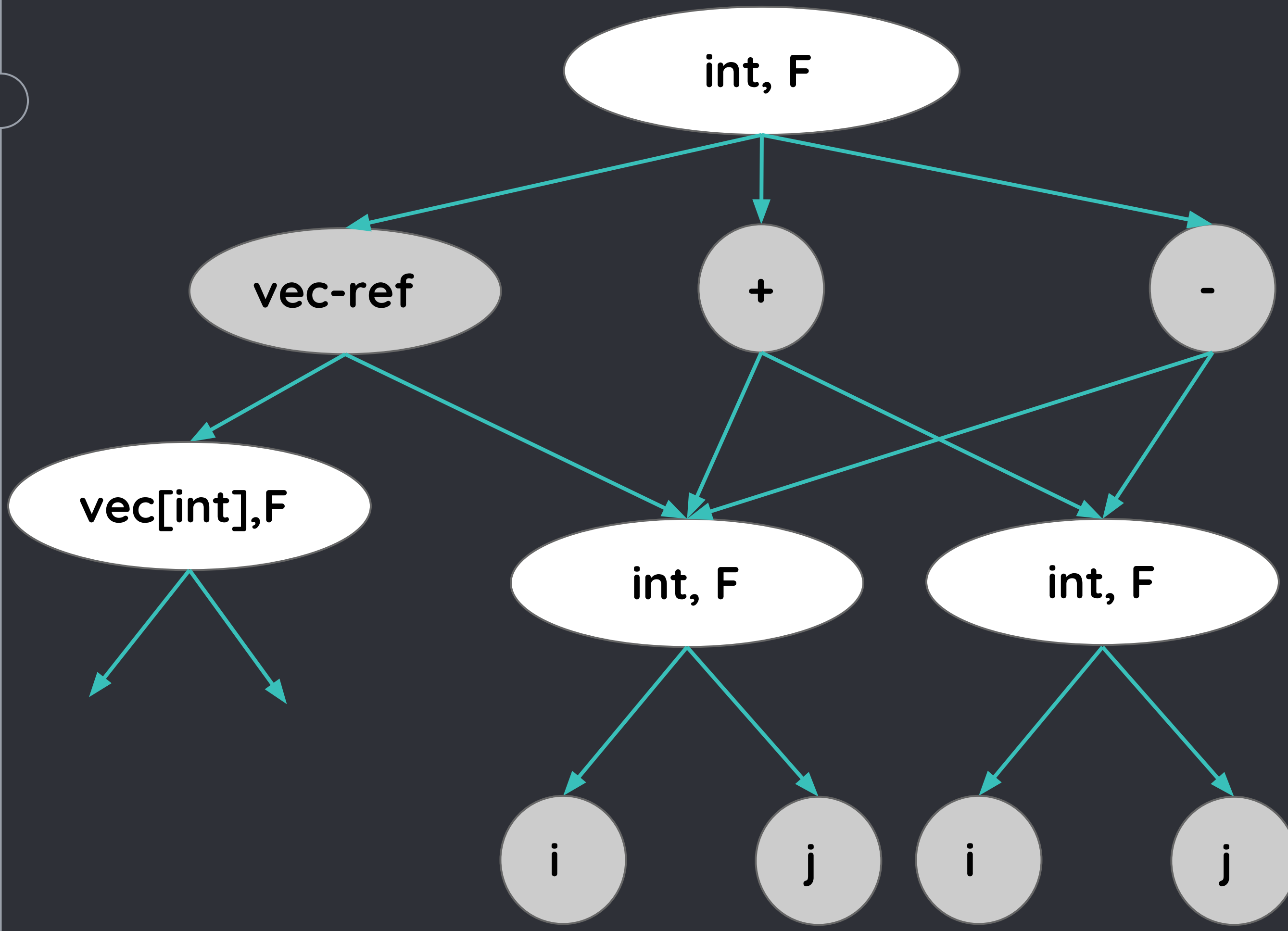
vec-ref: vec[int], F -> int, F -> int, F



TAKEAWAYS

- **Assume** components are annotated with type signatures and mutability requirements
- Generate a symbolic syntax graph top-down.
- Use component information to recursively pass down type and mutability constraints
- Share subtrees where possible
- Generate straight-line programs instead of bushy ones

EXAMPLE OF SHARING SUBGRAPHS WHENEVER POSSIBLE



CHECKPOINT

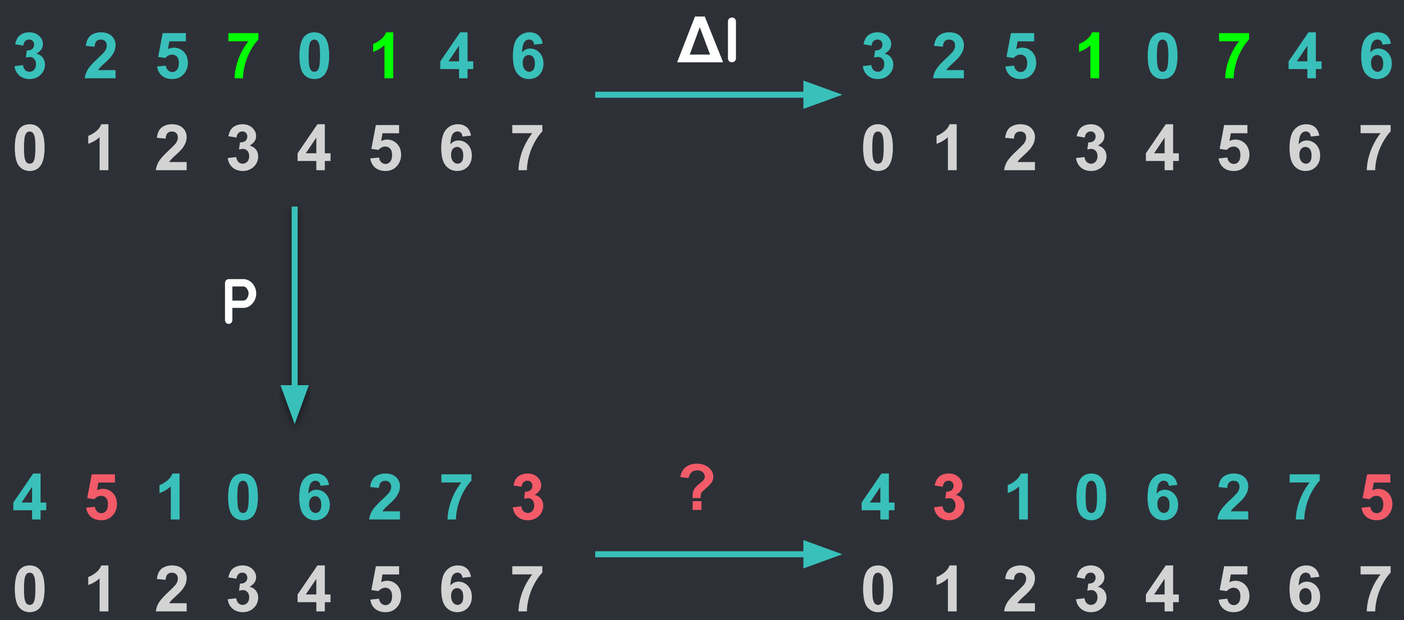
We have presented a novel algorithm to construct symbolic syntax trees. Now we present our evaluation.

1

We present a case study: SyncrO

SyncrO is a synthesizer for automatic incrementalization of programs built with the help of the above algorithm.

INCREMENTALLY MAINTAINING INVERSE GIVEN PERMUTATION CHANGES



$$\exists \Delta f \; \forall I, \Delta I \; : \; O + \Delta f(I, \Delta I, O) = f(I + \Delta I)$$

2

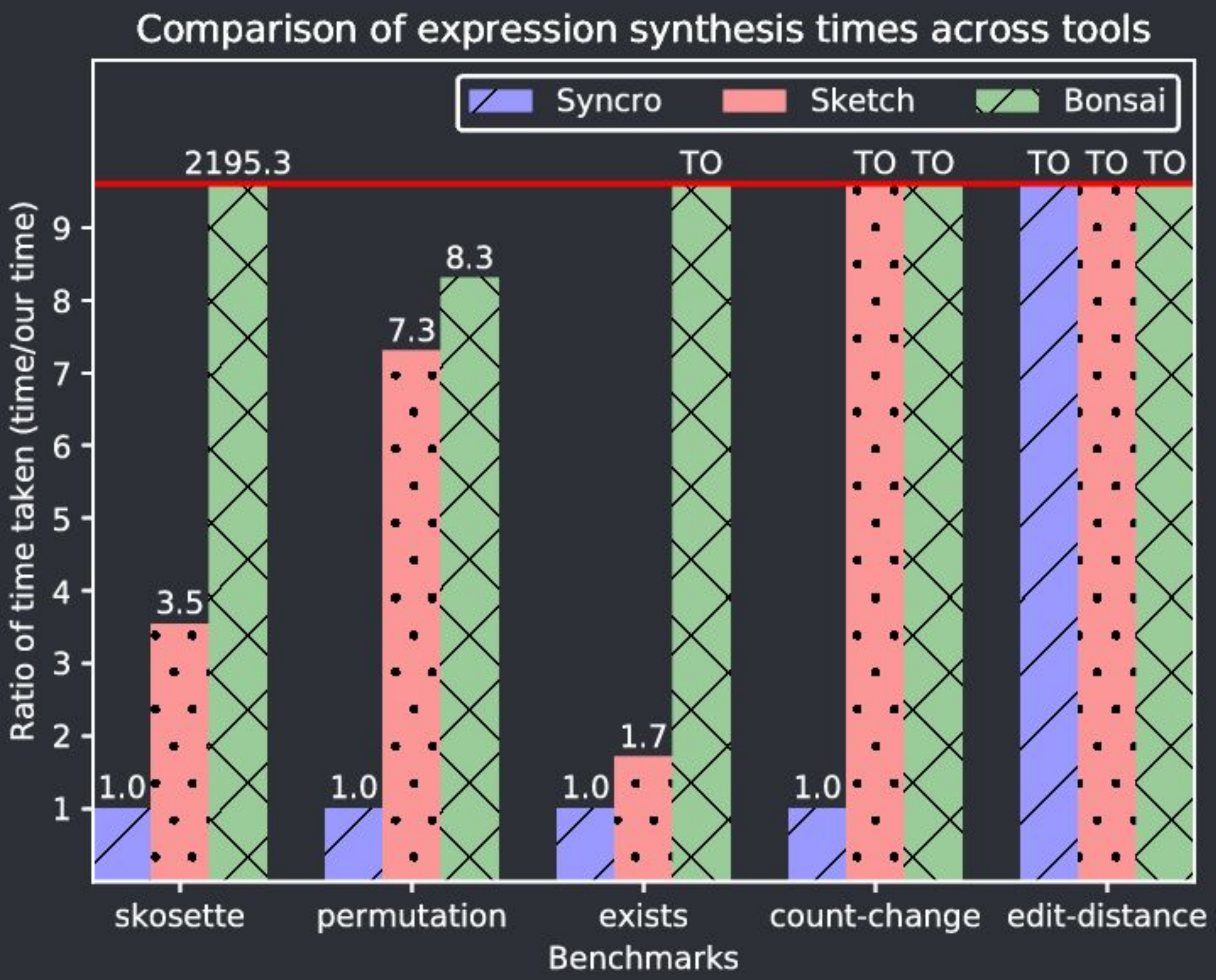
Numbers

Let’s now have a look at the numbers we obtained

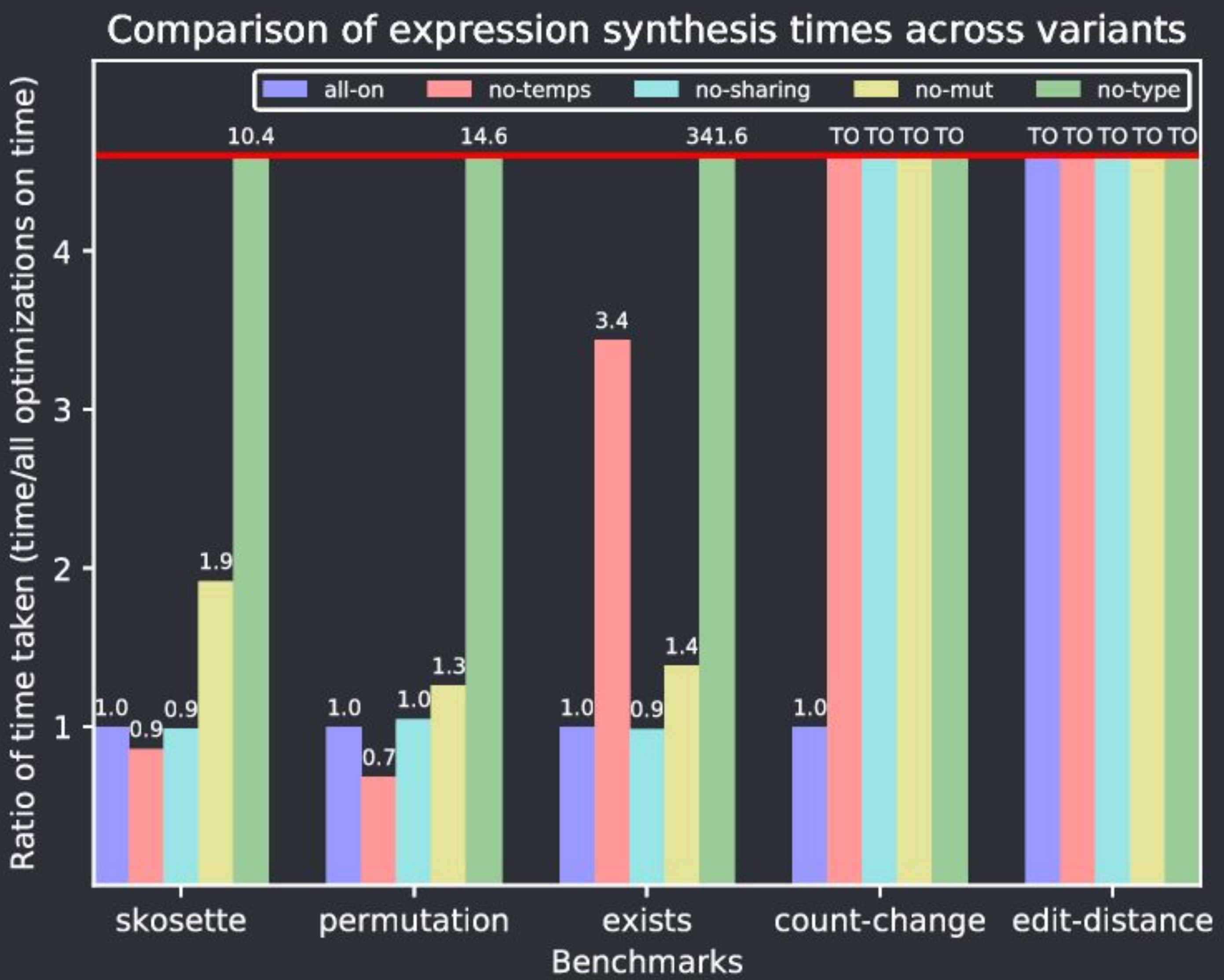
THE SUITE OF EXPRESSION SYNTHESIS BENCHMARKS

Benchmark	|Sol|	|LSpace|	|TSpace|	Time(s)
Skosette	**10**	**2^18**	**2^13**	**0.2**
Permutation	**16**	**2^39**	**2^28**	**0.3**
Exists	**50**	**2^72**	**2^80**	**0.6**
Count change	**26**	**2^75**	**2^395**	**2685.1**
Edit distance	**51**	**2^525**	**2^7022**	**TO**

Times taken (relative) to solve the expression synthesis benchmarks



- Times taken (relative) by variants of our algorithm



Thanks!

ANY QUESTIONS?

You can find me at

@sumith1896

sumith@stanford.edu

CREDITS

Special thanks to all the people who made and released these awesome resources for free:

- Presentation template by SlidesCarnival
- Photographs by Unsplash