PETSY: Polymorphic Enumerative Type-Guided Synthesis

Darya Verzhbinsky, Daniel Wang

Jan 19, 2021

Consider the following function description…

`concatNTimes`: concatenates a list `xs` to itself `n` times

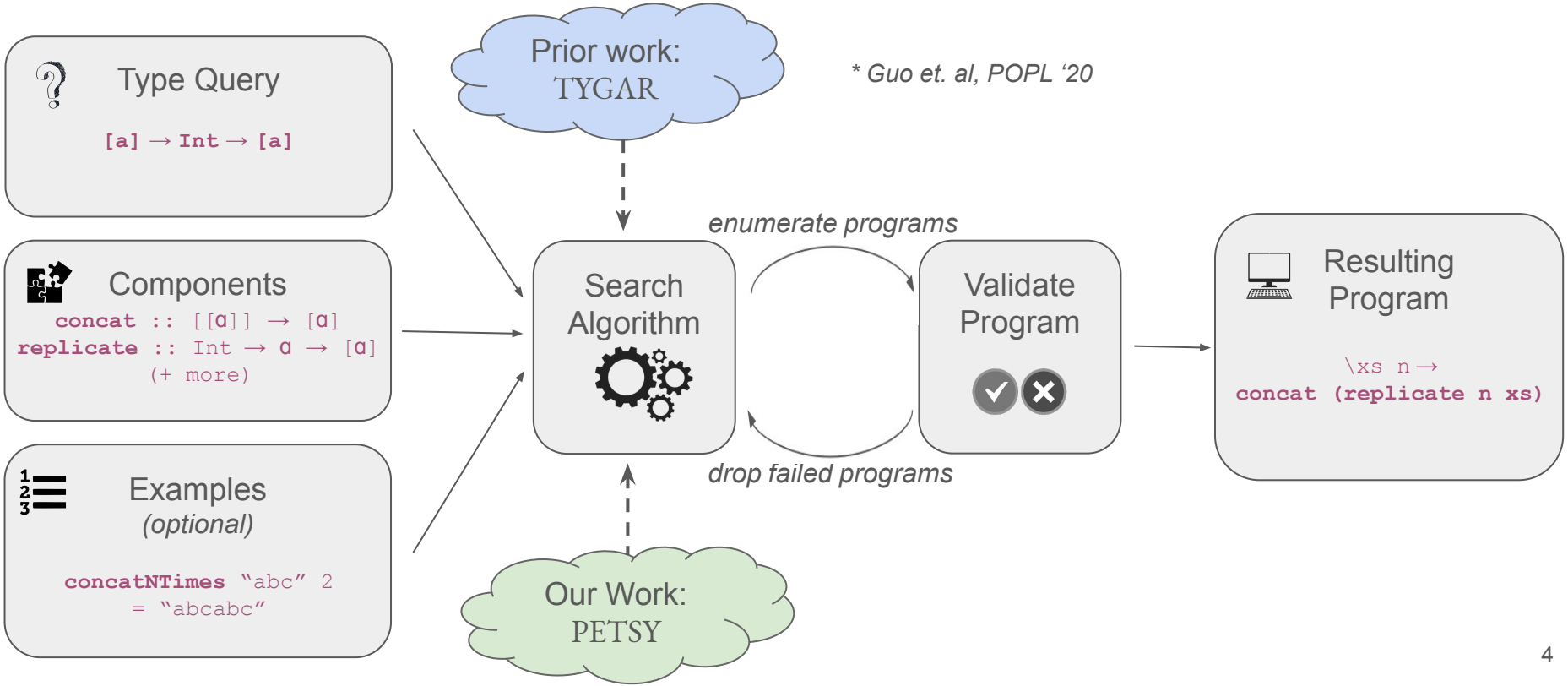
ex. `concatNTimes "abc" 2 = "abcabc"`

- In Haskell, this function can be implemented **concisely and idiomatically without explicit recursion**
  - Already existing recursive function synthesis tools for Haskell won't do
- Would be great it we a tool could "synthesize" this for us!

Challenges with synthesizing Haskell

- Many Haskell library functions make heavy use of:
  - **polymorphism**
  - **higher-order arguments**
  - **typeclasses**
- This makes search space quite complex

Synthesis problem overview



TYGAR vs. PETSY

Prior work:
TYGAR

- Synthesis via Petri net reachability
- Cannot synthesize programs with inner lambdas
  - e.g. `\xs -> map (\p → fst p + snd p) xs`
- Efficient but complex algorithm

Our Work:
PETSY

- Top-down enumerative search
- Can synthesize programs with inner lambdas
- Simpler algorithm: can it compete?

Enumeration

- **Challenge: polymorphism**

Memoization

Evaluation

Enumeration

Components

```
concat :: [[a]] → [a]
replicate :: Int → a → [a]
```

Examples

```
concatNTimes "abc" 2
= "abcabc"
```

```
?? :: [a] → Int → [a]
```

Steps:

1. Component with matching type?
2. Is it a function type? Yes ⇒ create lambda

Enumeration

Components

```
concat :: [[a]] → [a]
replicate :: Int → a → [a]
xs :: [a]
```

Examples

```
concatNTimes "abc" 2
= "abcabc"
```

```
\xs →    ?? :: Int → [a]
```

Steps:

1. Component with matching type?
2. Is it a function type? Yes ⇒ create lambda

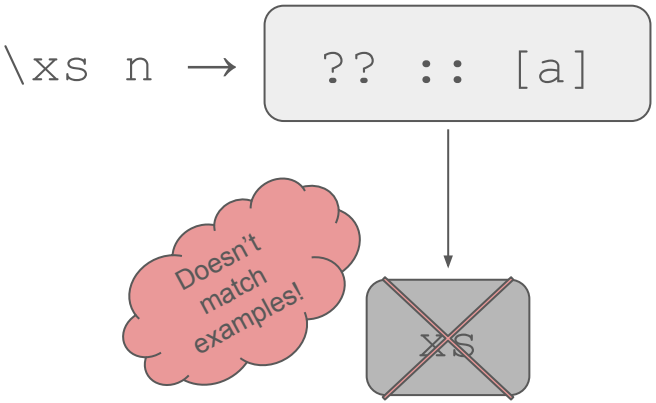
Enumeration

Components

```
concat :: [[a]] → [a]
replicate :: Int → a → [a]
xs :: [a]
n :: Int
```

Examples

```
concatNTimes "abc" 2
= "abcabc"
```



Steps:

1. Component with matching type?
2. Is it a function type? Yes ⇒ create lambda

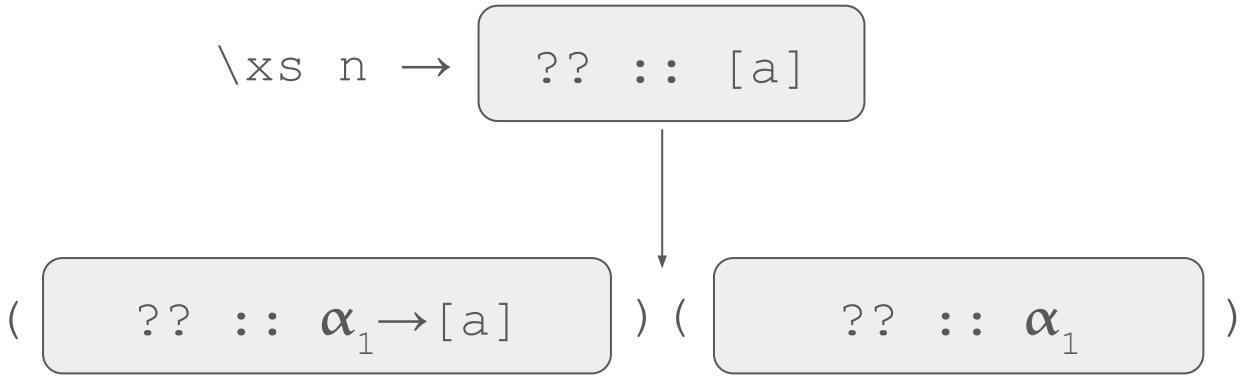
Enumeration

Components

```
concat :: [[a]] → [a]
replicate :: Int → a → [a]
xs :: [a]
n :: Int
```

Examples

```
concatNTimes "abc" 2
= "abcabc"
```



Steps:

1. Component with matching type?
2. Is it a function type? Yes ⇒ create lambda
3. Not a function type ⇒ function application, recurse on 2 subgoals

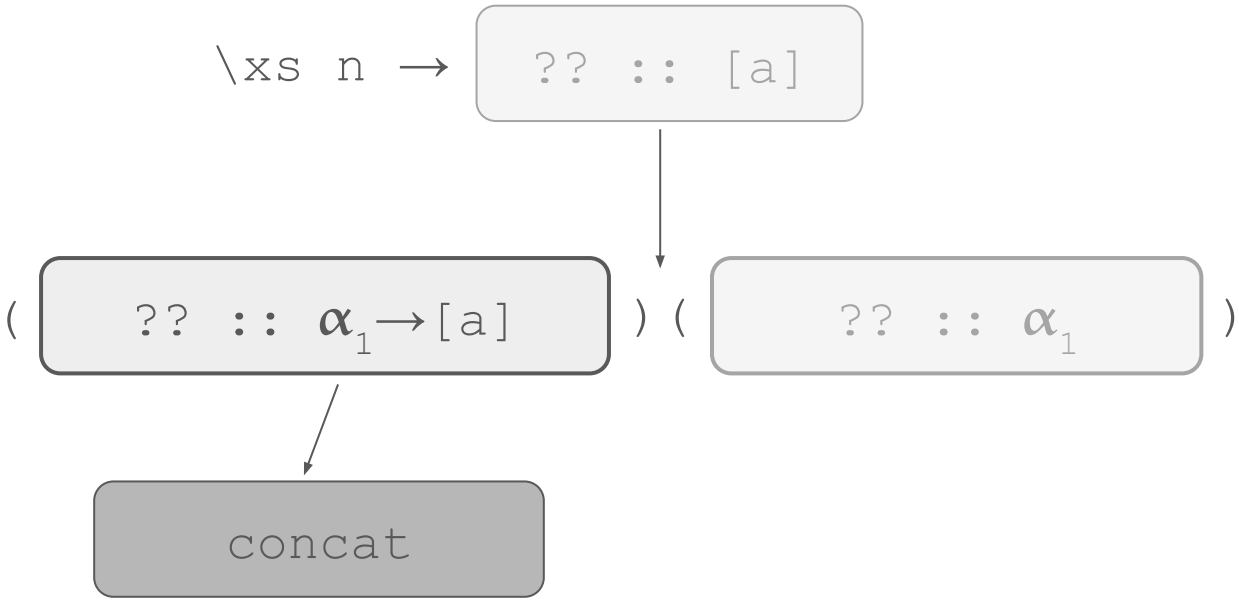
Enumeration

Components

```
concat :: [[a]] → [a]
replicate :: Int → a → [a]
xs :: [a]
n :: Int
```

Examples

```
concatNTimes "abc" 2
= "abcabc"
```



Steps:

1. Component with matching type?
2. Is it a function type? Yes ⇒ create lambda
3. Not a function type ⇒ function application, recurse on 2 subgoals

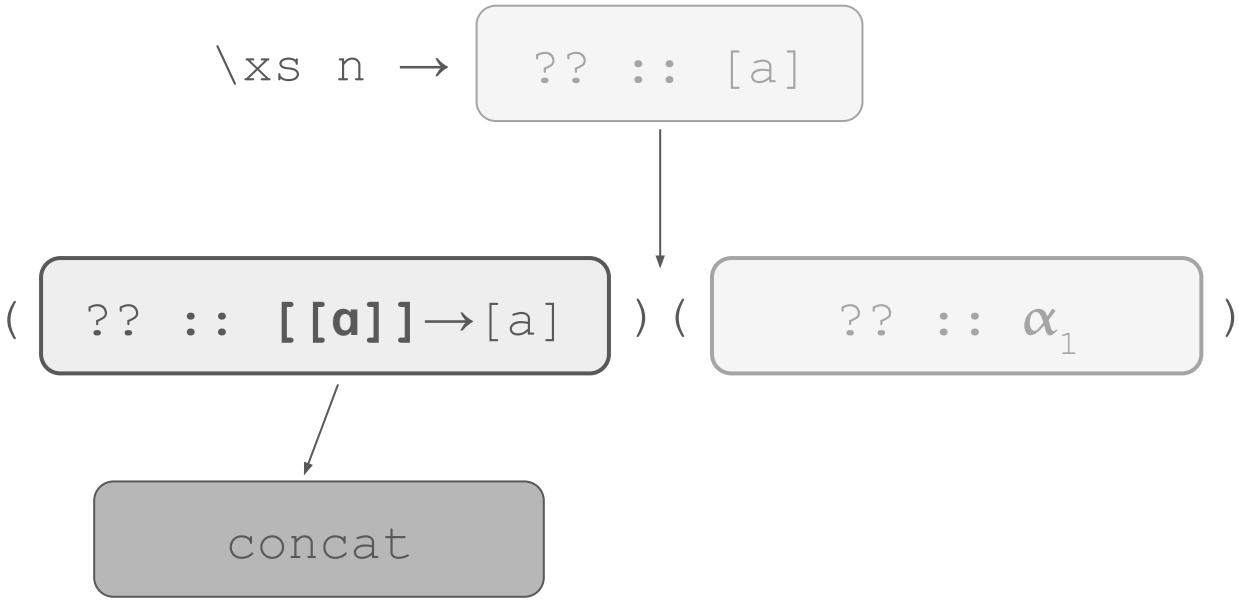
Enumeration

Components

```
concat :: [[a]] → [a]
replicate :: Int → a → [a]
xs :: [a]
n :: Int
```

Examples

```
concatNTimes "abc" 2
= "abcabc"
```



Steps:

1. Component with matching type?
2. Is it a function type? Yes ⇒ create lambda
3. Not a function type ⇒ function application, recurse on 2 subgoals

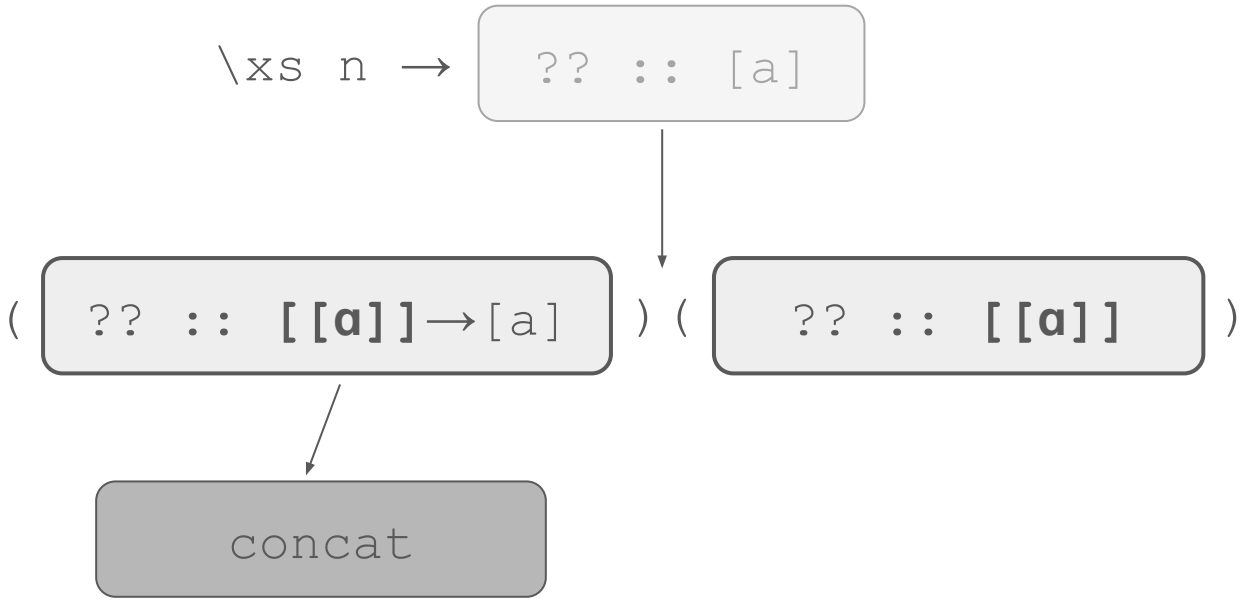
Enumeration

Components

```
concat :: [[a]] → [a]
replicate :: Int → a → [a]
xs :: [a]
n :: Int
```

Examples

```
concatNTimes "abc" 2
= "abcabc"
```



Steps:

1. Component with matching type?
2. Is it a function type? Yes ⇒ create lambda
3. Not a function type ⇒ function application, recurse on 2 subgoals

Enumeration

Components

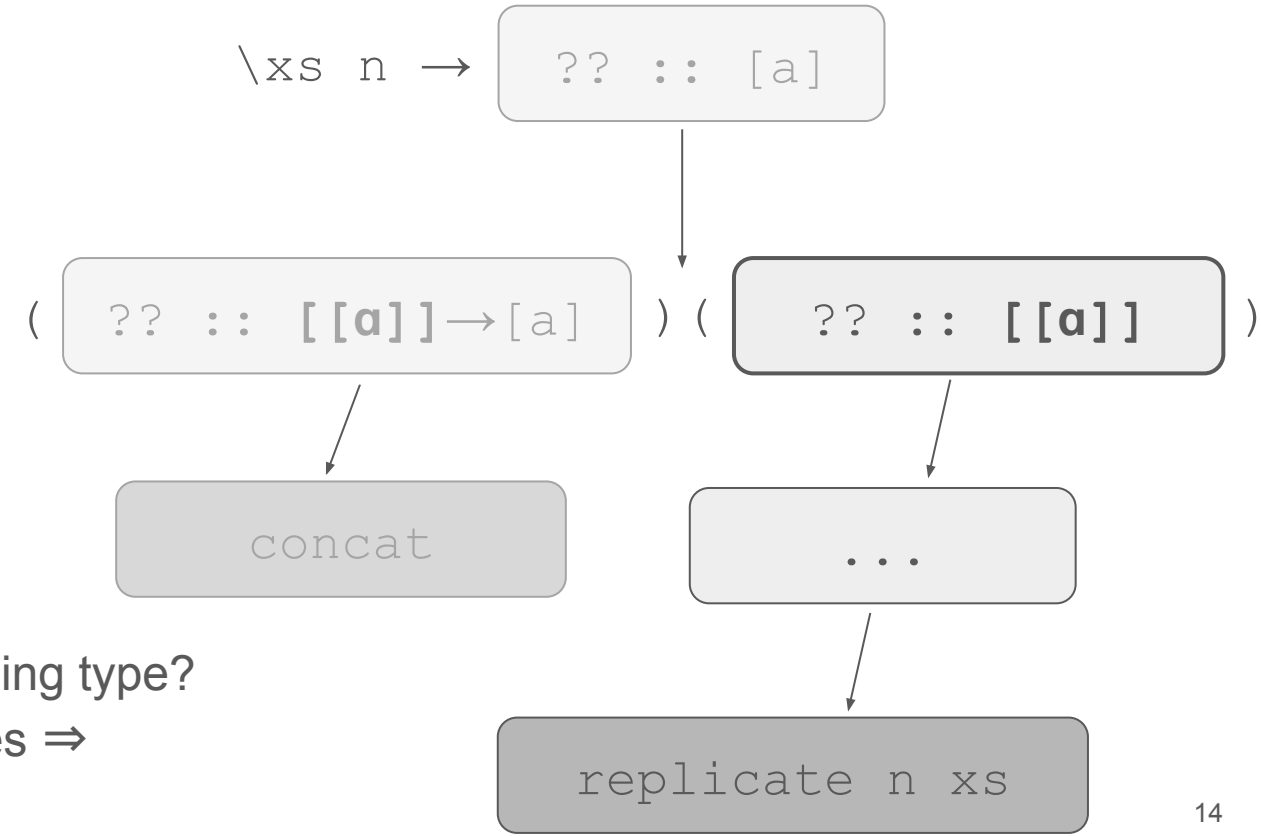
```
concat :: [[a]] → [a]
replicate :: Int → a → [a]
xs :: [a]
n :: Int
```

Examples

```
concatNTimes "abc" 2
= "abcabc"
```

Steps:

1. Component with matching type?
2. Is it a function type? Yes ⇒ create lambda



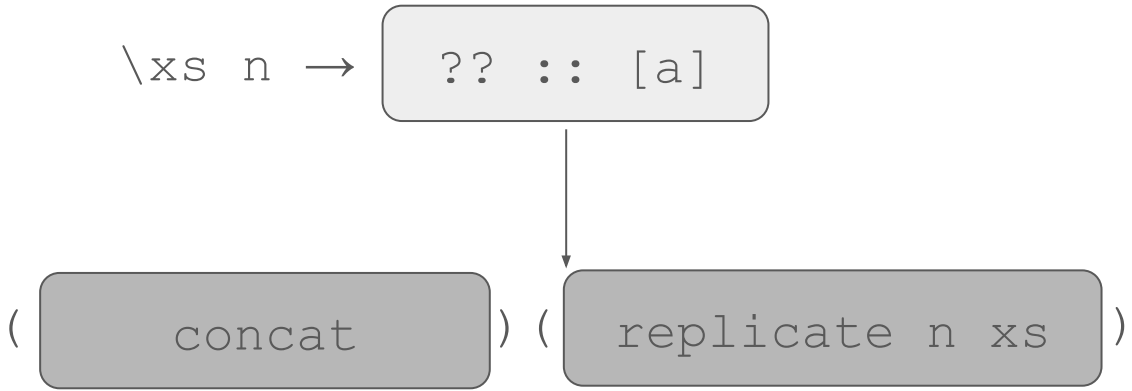
Enumeration

Components

```
concat :: [[a]] → [a]
replicate :: Int → a → [a]
xs :: [a]
n :: Int
```

Examples

```
concatNTimes "abc" 2
= "abcabc"
```



Steps:

1. Component with matching type?
2. Is it a function type? Yes ⇒ create lambda
3. Not a function type ⇒ function application, recurse on 2 subgoals

Enumeration

Components

```
concat :: [[a]] → [a]
replicate :: Int → a → [a]
xs :: [a]
n :: Int
```

Examples

```
concatNTimes "abc" 2
= "abcabc"
```

```
\xs n → ?? :: [a]
```

↓

```
concat (replicate n xs)
```

Steps:

1. Component with matching type?
2. Is it a function type? Yes ⇒ create lambda
3. Not a function type ⇒ function application, recurse on 2 subgoals
4. Return :)

Enumeration

Components

```
concat :: [[a]] → [a]
replicate :: Int → a → [a]
xs :: [a]
n :: Int
```

Examples

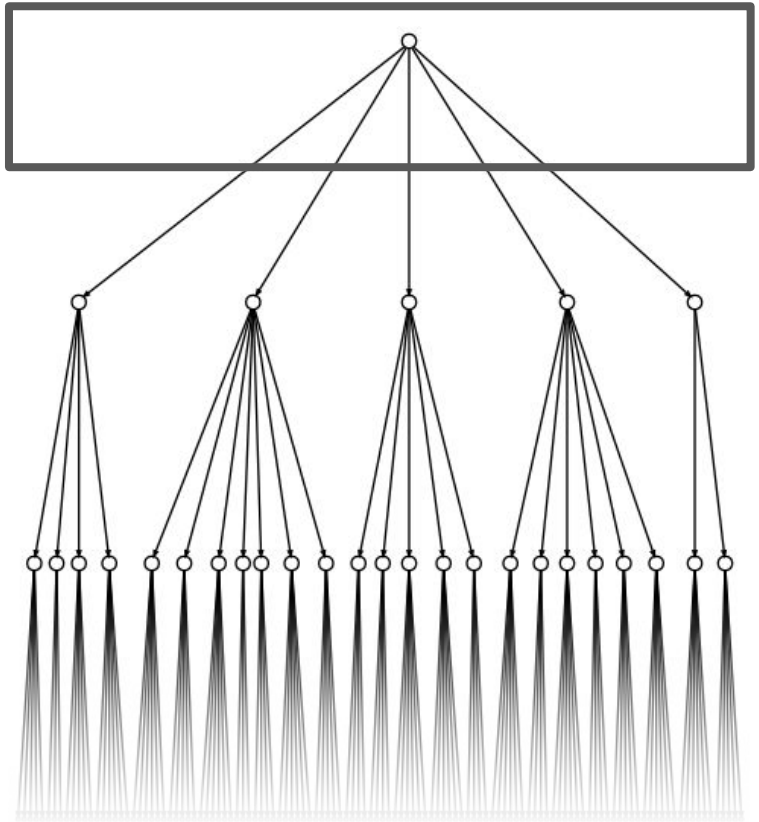
```
concatNTimes "abc" 2
= "abcabc"
```

```
\xs n → concat (replicate n xs)
```

Steps:

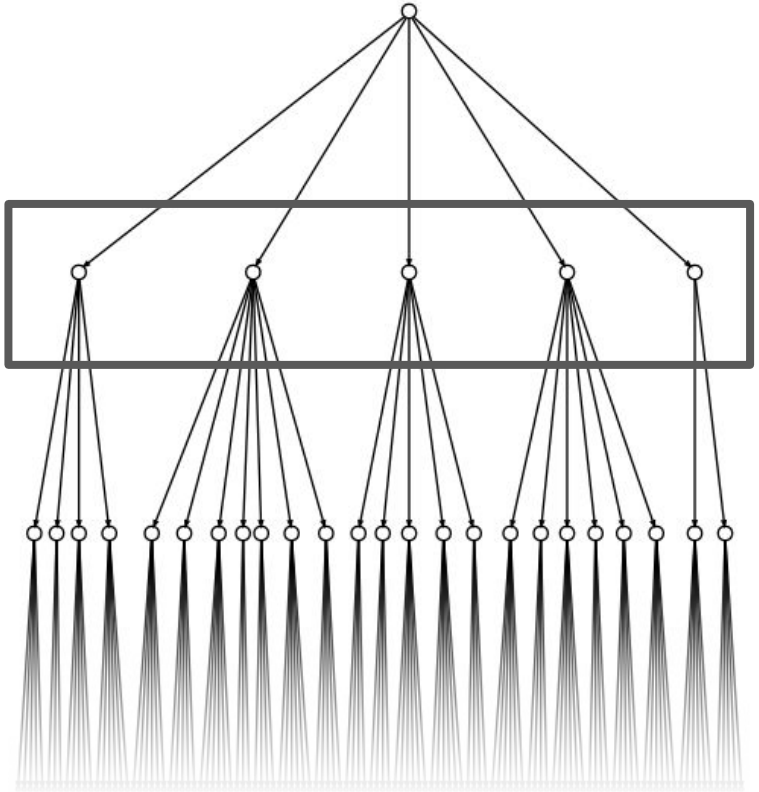
1. Component with matching type?
2. Is it a function type? Yes ⇒ create lambda
3. Not a function type ⇒ function application, recurse on 2 subgoals
4. Return :)

Making search feasible



~140 programs at size 1

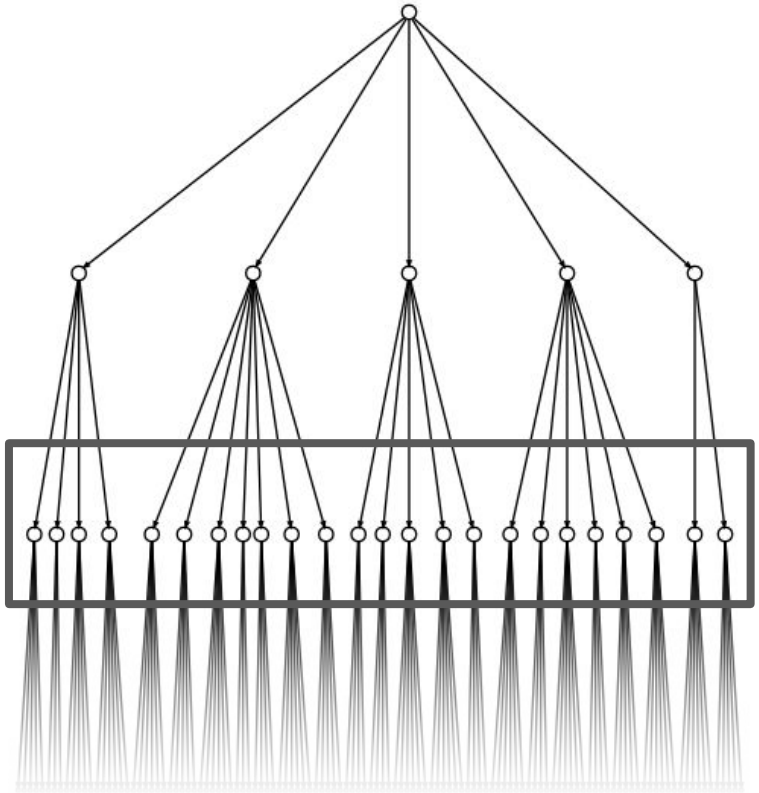
Making search feasible



~140 programs at size 1

~19,600 programs at size 2

Making search feasible

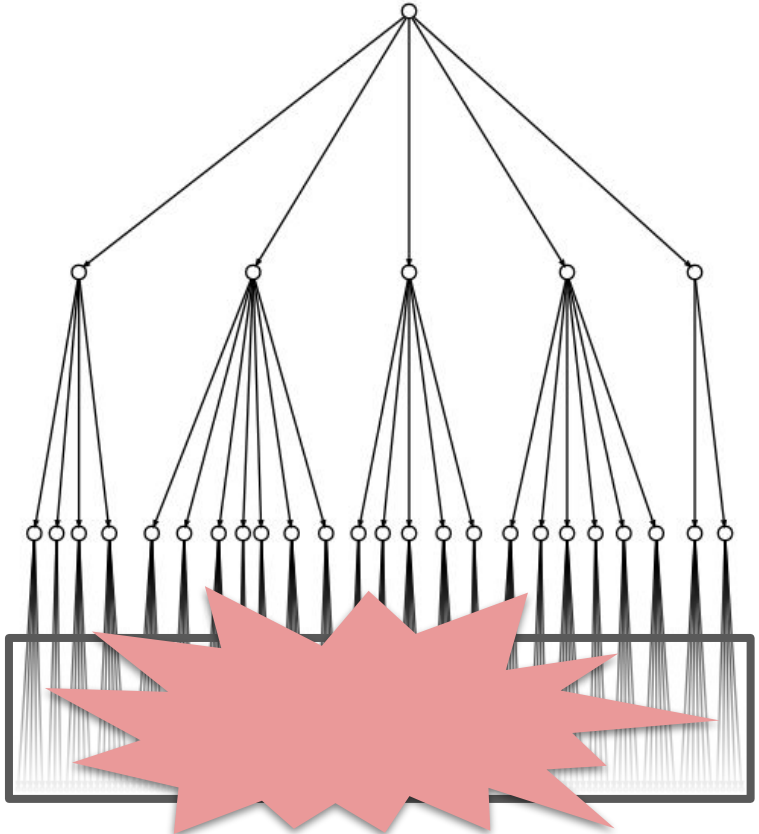


~140 programs at size 1

~19,600 programs at size 2

~2,744,000 programs at size 3

Making search feasible



~140 programs at size 1

~19,600 programs at size 2

~2,744,000 programs at size 3

EXPLOSION !

Enumeration

Memoization

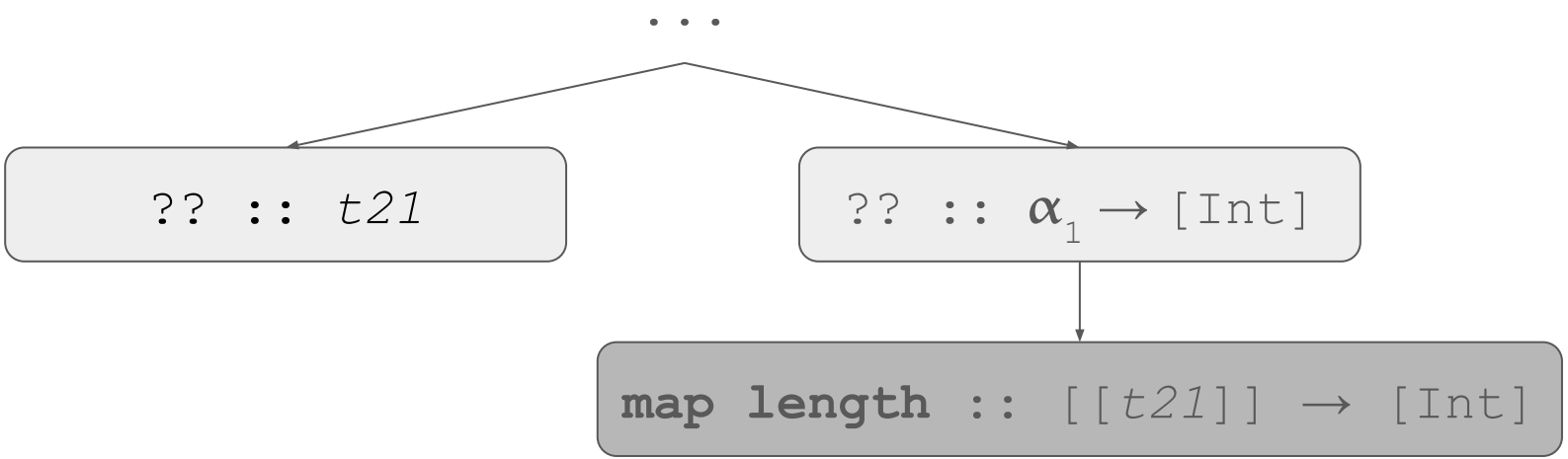
- **Making enumeration scale**

Evaluation

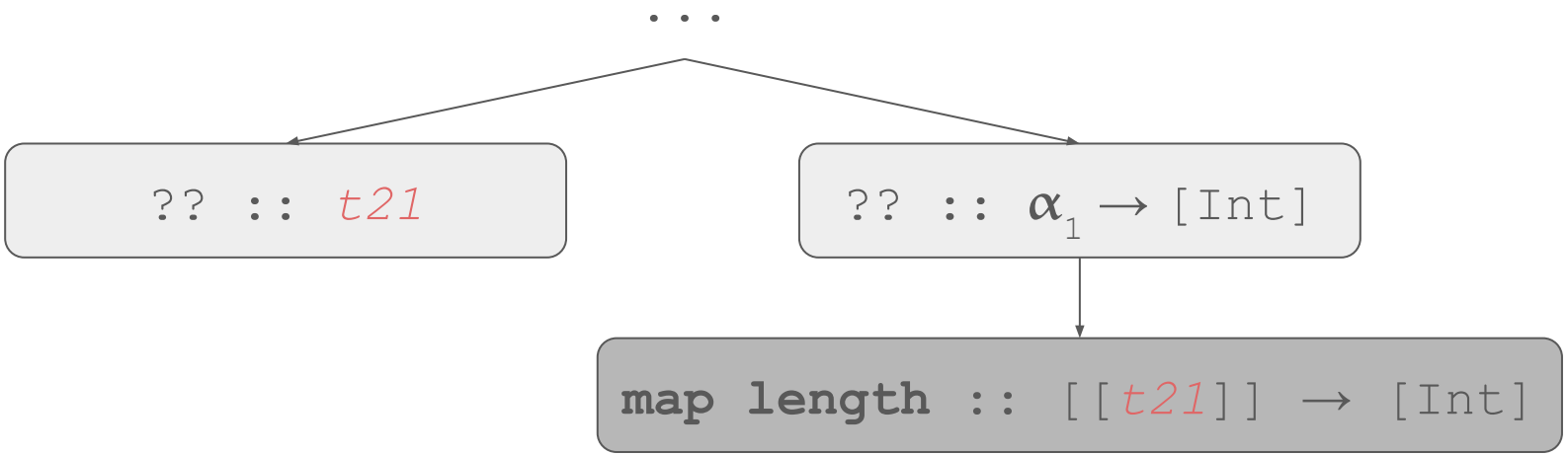
Memoization - Prior Work

- Myth *(Osera et. al, PLDI '15)* showed memoization is **crucial** for fast enumerative synthesis
- However, their memoization technique only worked in monomorphic situations

Memoization - complications with Polymorphism



Memoization - complications with Polymorphism



Working solution: we ignore the stored type and infer the type at retrieval

Enumeration

Memoization

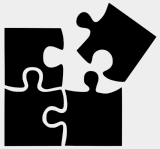
Evaluation

Benchmarking PETSY

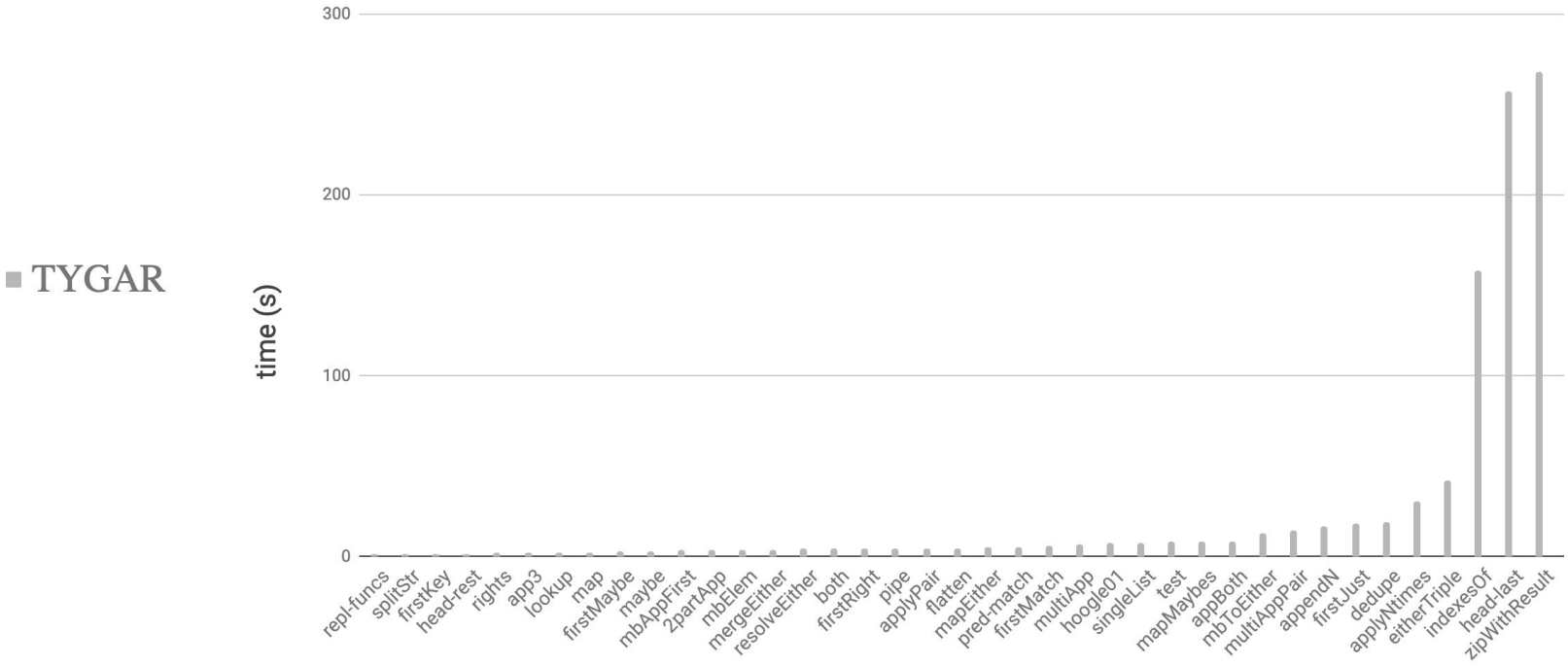
Tested against TYGAR

39 Benchmarks (from TYGAR[1])

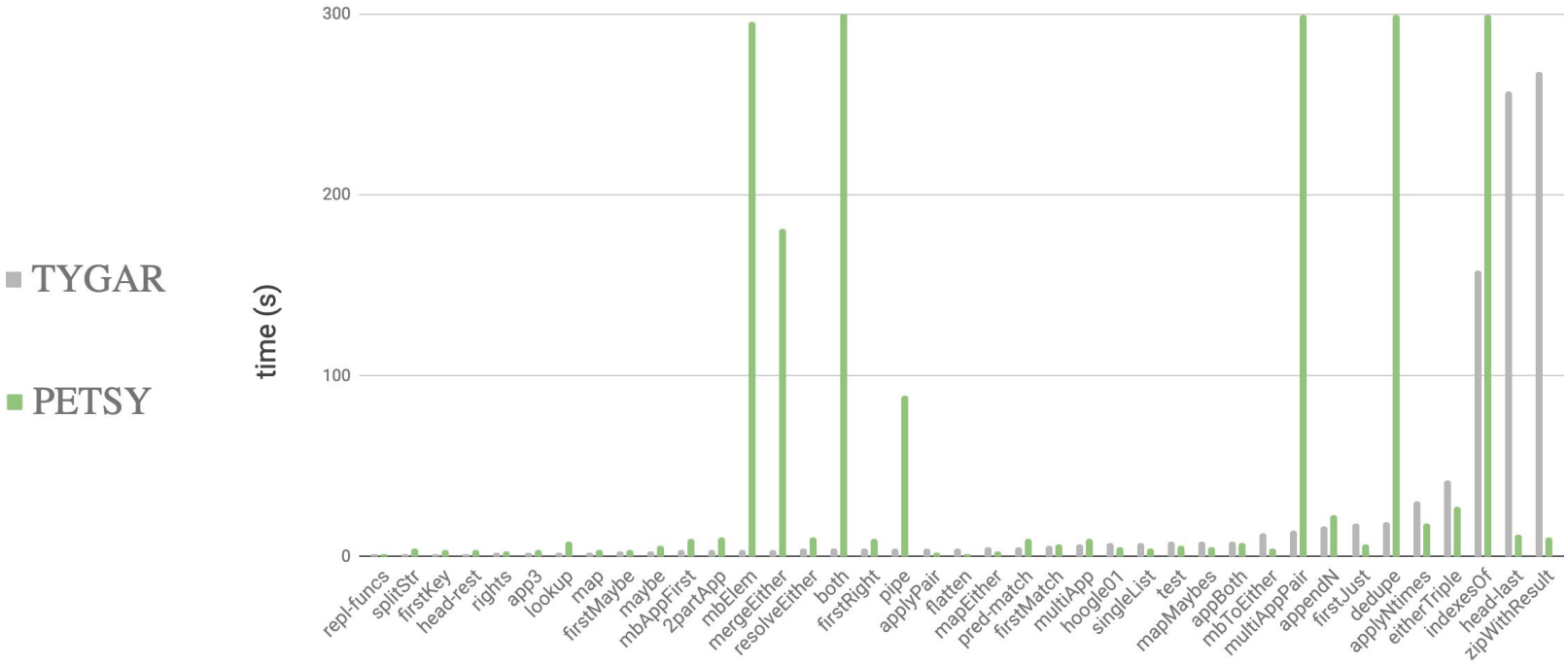
140 Components



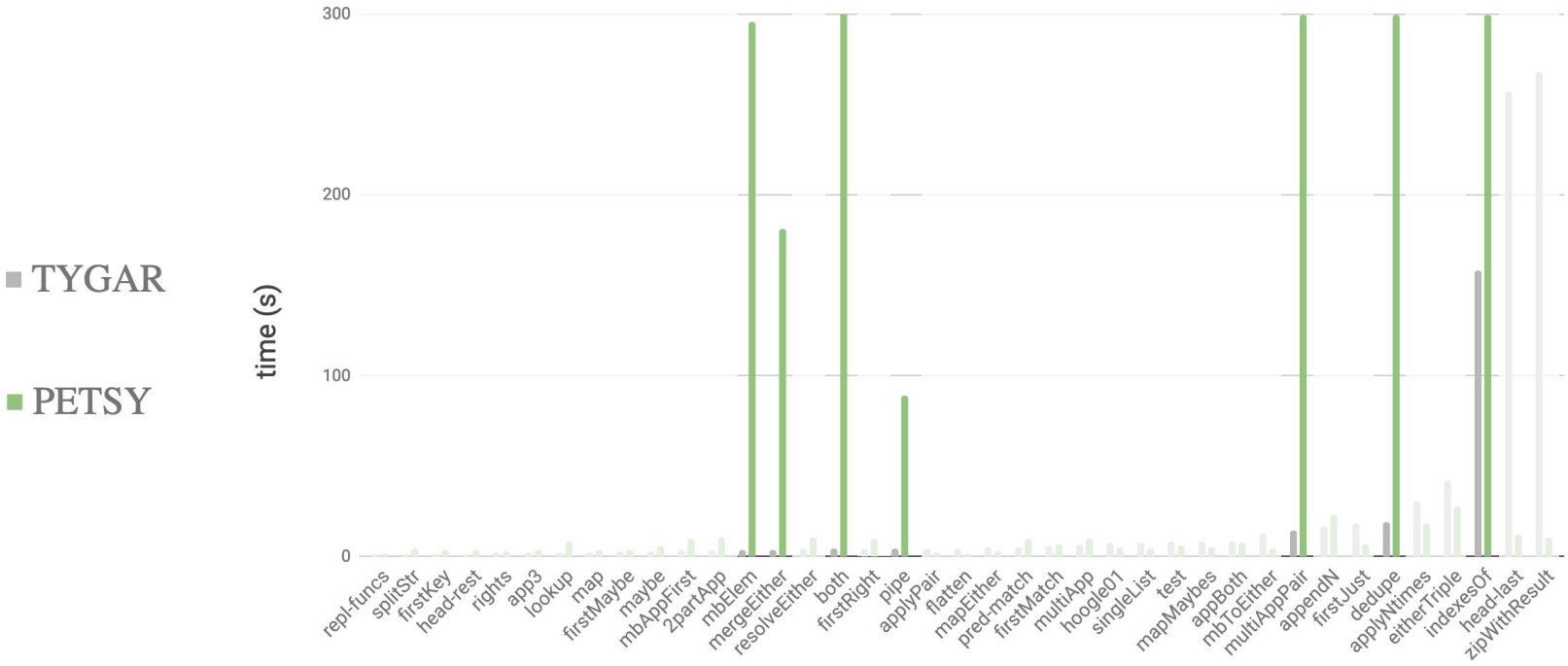
Evaluation - TYGAR vs PETSY



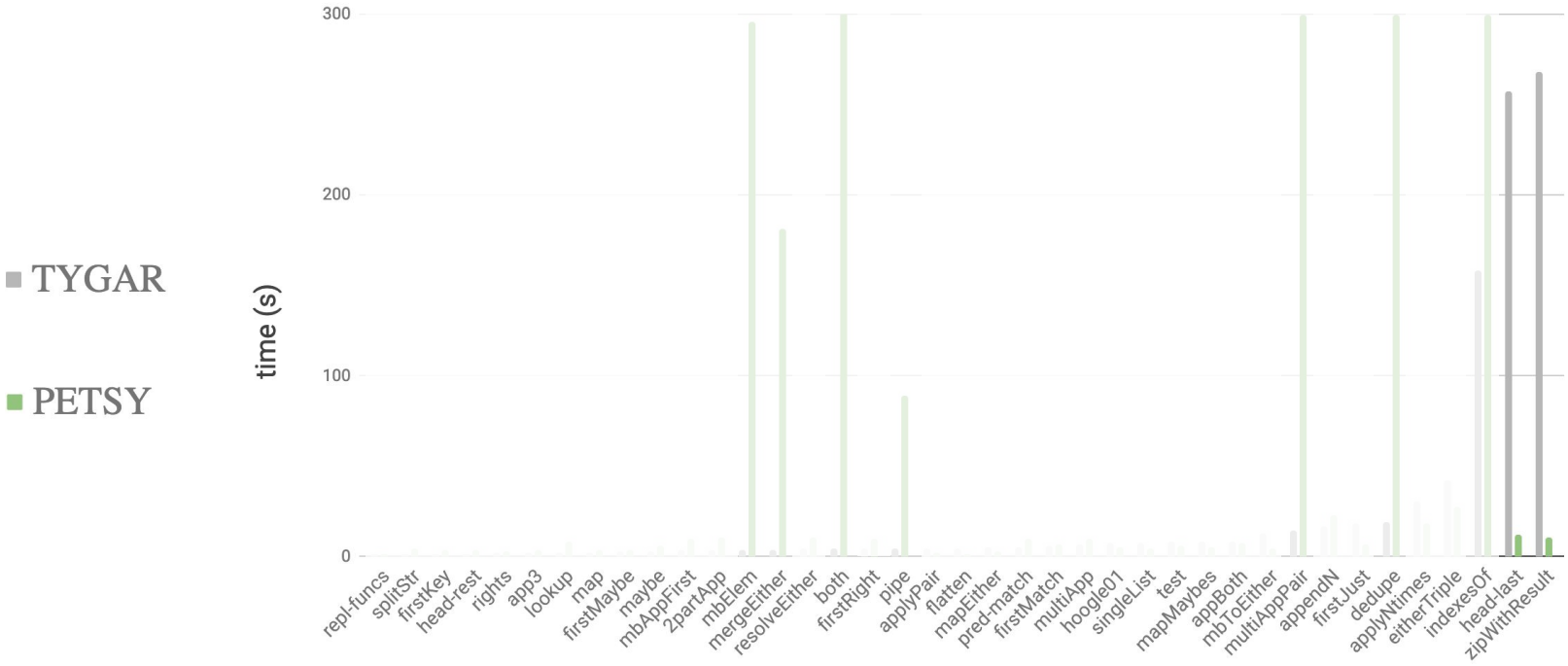
Evaluation - TYGAR vs PETSY



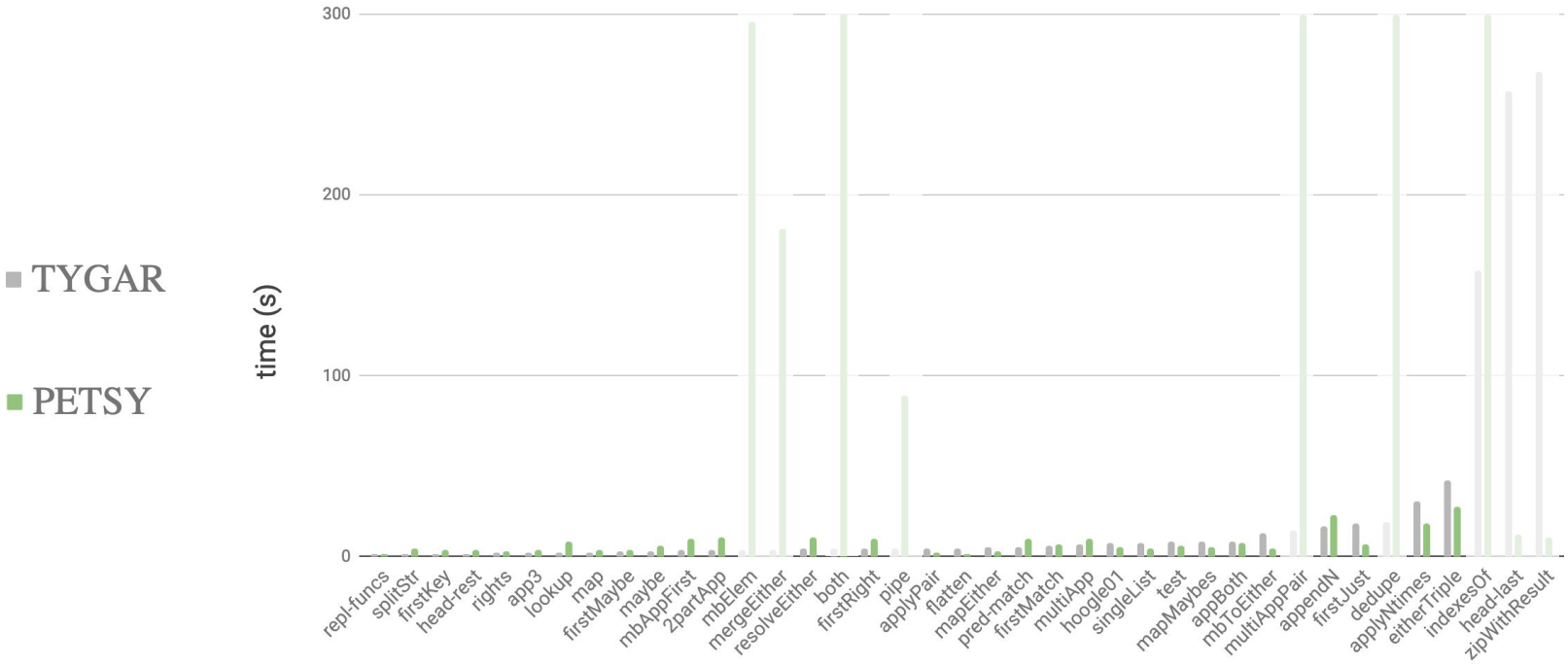
Evaluation - TYGAR vs PETSY



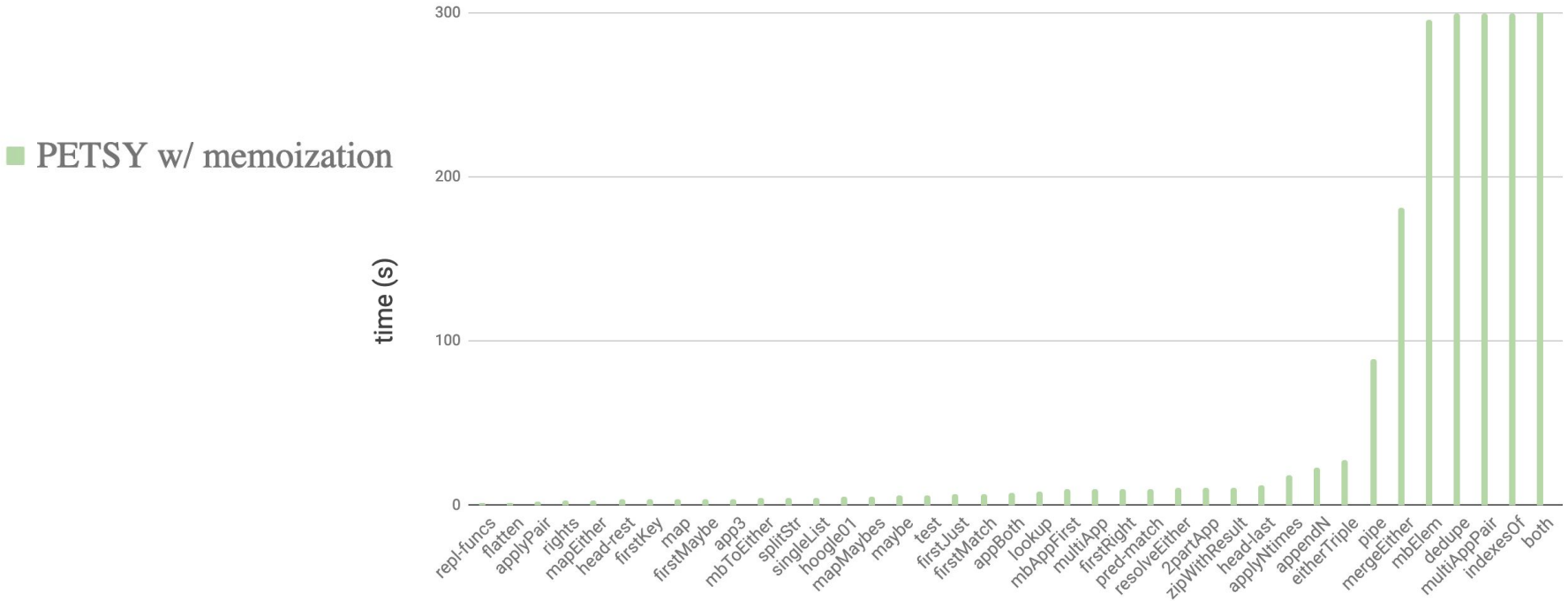
Evaluation - TYGAR vs PETSY



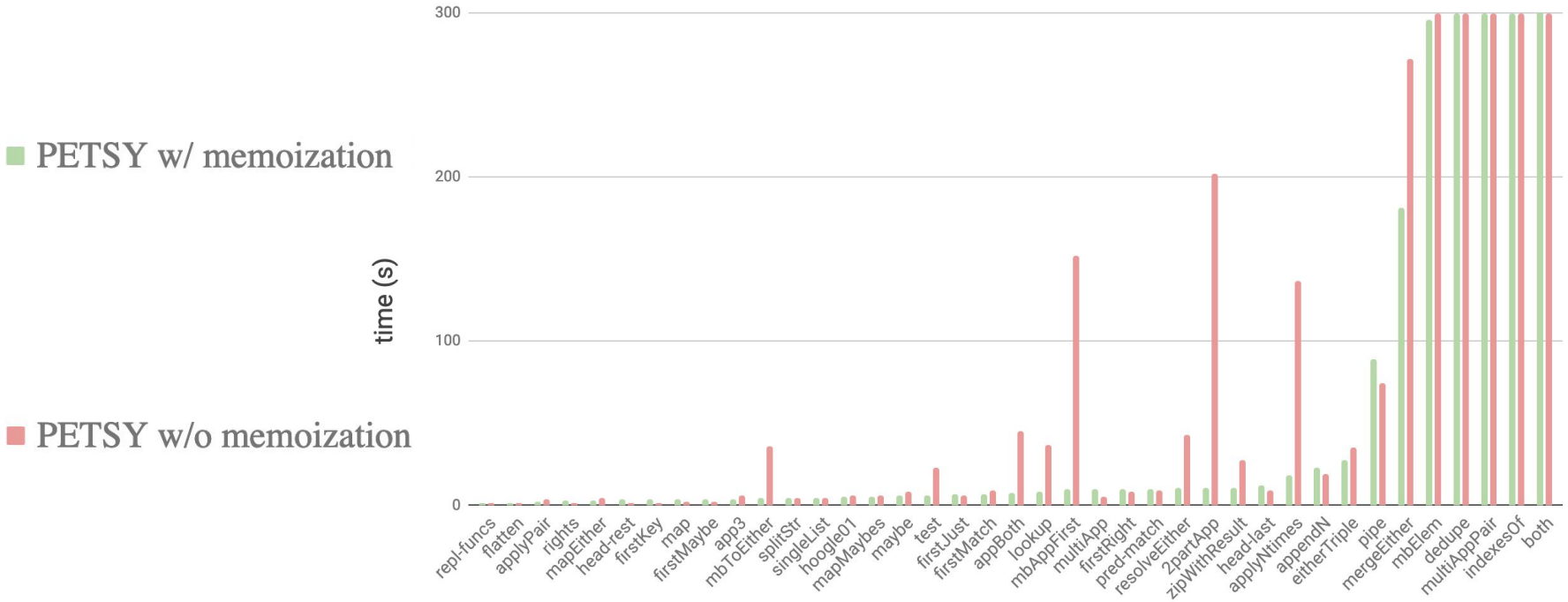
Evaluation - TYGAR vs PETSY



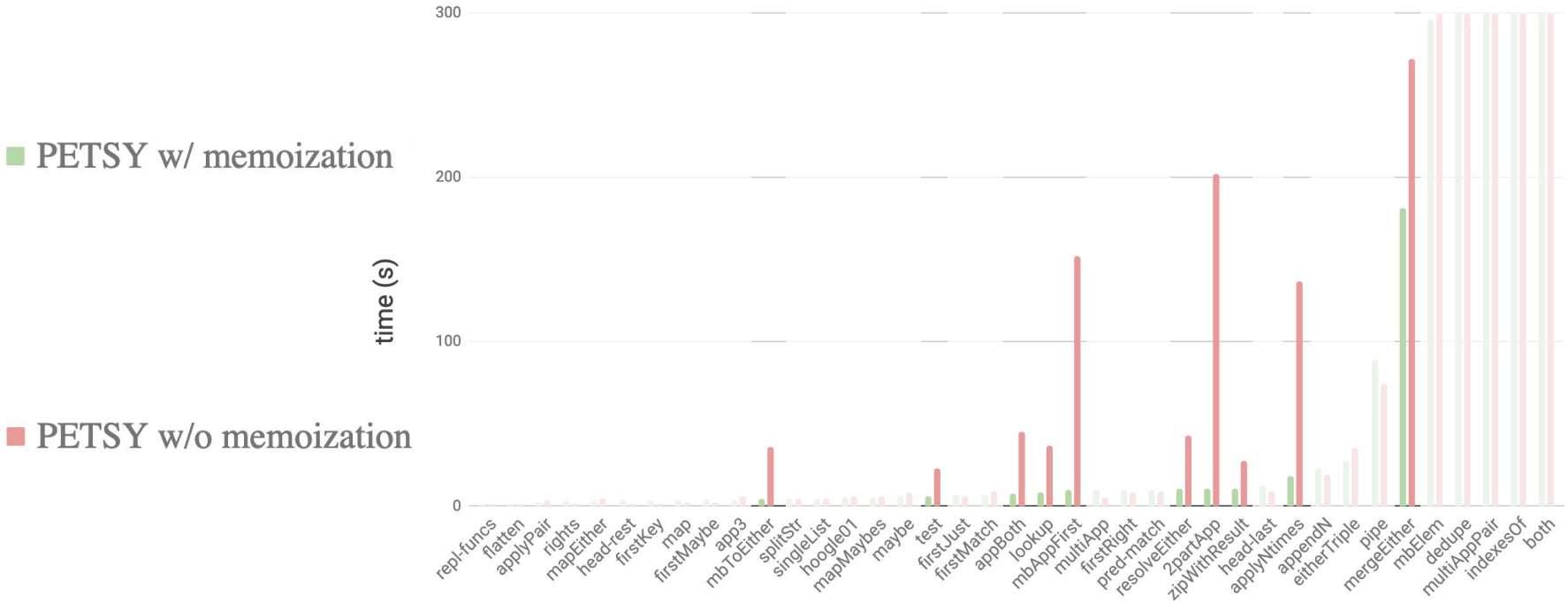
Evaluation - PETSY memoization



Evaluation - PETSY memoization



Evaluation - PETSY memoization



Thank you!