SPECTACULAR: Finding Laws from 25 Trillion Terms

Matthías Páll Gissurarson
Chalmers Institute of Technology
Gothenburg, Sweden
pallm@chalmers.se

Diego Roque
Dark Forest Technologies
New York, USA
diego9627@gmail.com

James Koppel
MIT
Cambridge, USA
jkoppel@mit.edu

CHALMERS

WASP | WALLENBERG AUTONOMOUS SYSTEMS AND SOFTWARE PROGRAM

ICST, *Dublin*,
April 18, 2023

UNIVERSITY OF GOTHENBURG

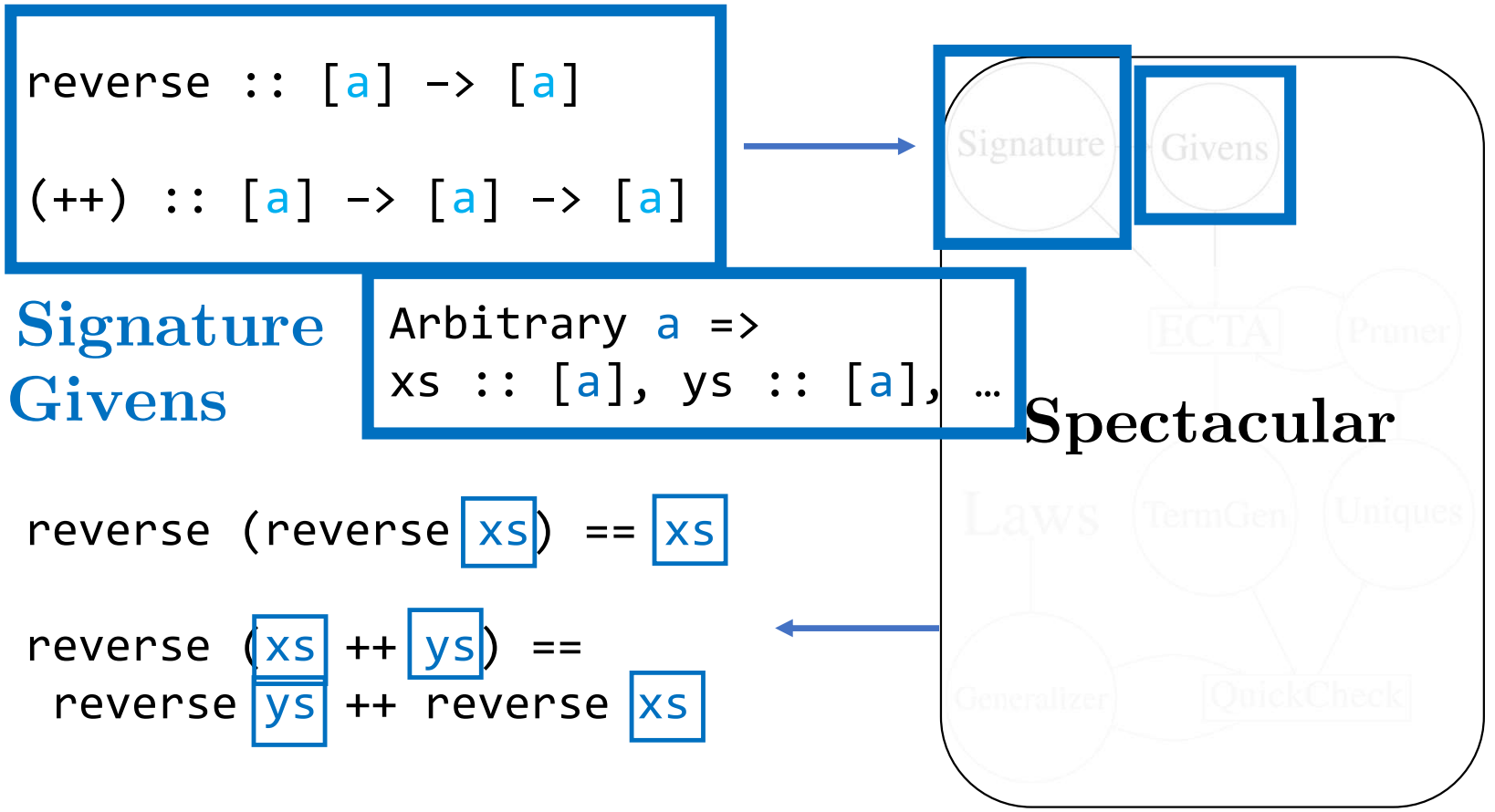
Spectacular: Background

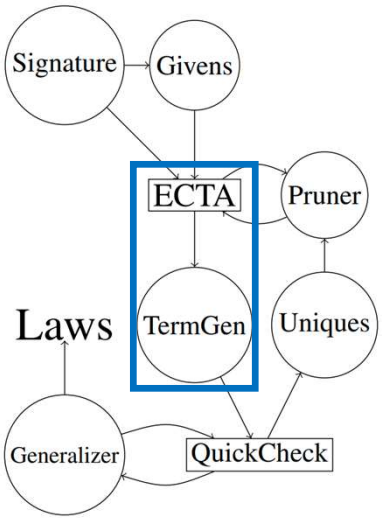
- ***Functional Programming (Haskell)***
  - Pure by default, state is explicit.
  - Great for equational reasoning!
- **Metamorphic Testing**
  - Declare *what* you want to test, not how.
  - Targets the *oracle problem*.
- **QuickCheck**
  - Uses *generators* to test *properties (test oracles!)*
  - *E.g.*, `reverse (reverse xs) == xs`
- **QuickSpec**
  - Generates QuickCheck properties!

Spectacular: Motivation

- Test-suites are often **lacking**
- **Property-based** tests (i.e. *metamorphic relations*) cover more, but **hard to write and identify**
- **Synthesizing properties** helps!
- Current approaches (like QuickSpec) **don't scale**
- *But **Spectacular does** (better)**!***

Spectacular: Outline





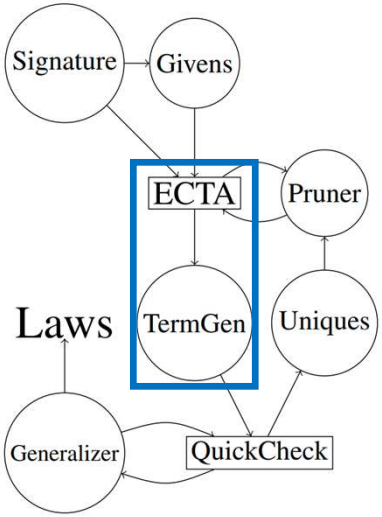
Spectacular: Generating Terms

```
reverse :: [a] -> [a]
(++) :: [a] -> [a] -> [a]
xs :: [a], ys :: [a]
```

Type-checker says...

xs ✓
ys ✓
reverse xs ✓
reverse ys ✓
reverse (++)? ✗
reverse ++ reverse? ✗

For these 2 functions with 2 added givens:
5460 possible programs of size <= 6
Only **128** are well-typed
and **84** are base values!
For 32 functions and 60 added givens:
25 trillion possible programs!



Spectacular: Generating Terms

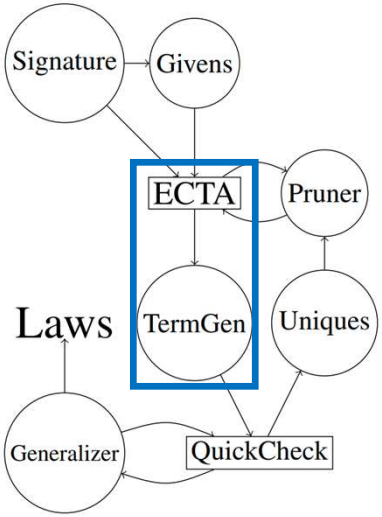
```
reverse :: [a] -> [a]
(++) :: [a] -> [a] -> [a]
xs :: [a], ys :: [a]
```

Equality-
Constrained
Tree
Automata

To the rescue!

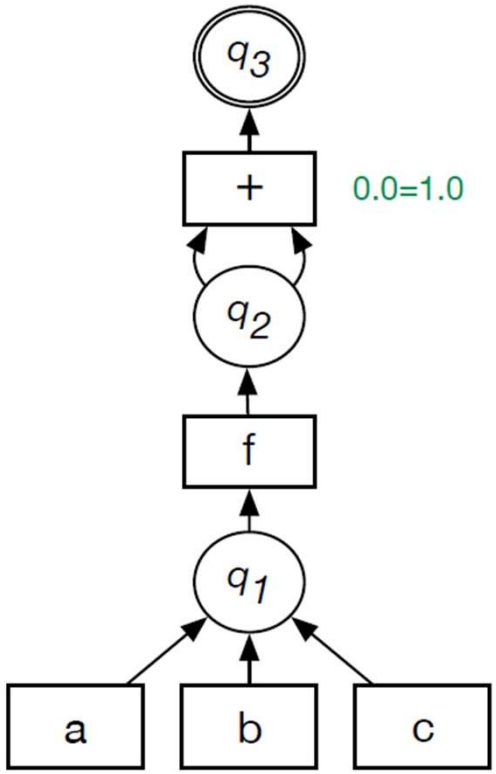
xs ✓
ys ✓
reverse xs ✓
reverse ys ✓
reverse (++)? ✗
reverse ++ reverse? ✗

By combining **compact representation** and **constraint solving** we can **efficiently enumerate** well-typed programs!



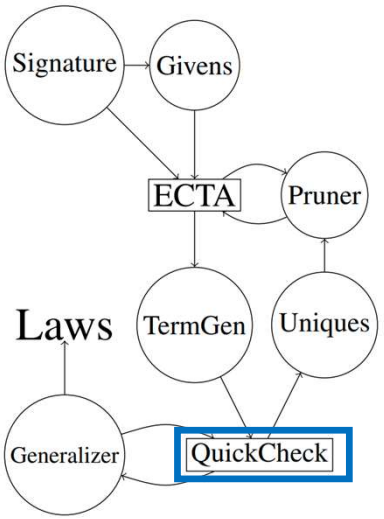
Spectacular: ECTAs (Koppel et al., ICFP 2022)

```
reverse :: [a] -> [a]
(++) :: [a] -> [a] -> [a]
xs :: [a], ys :: [a]
```



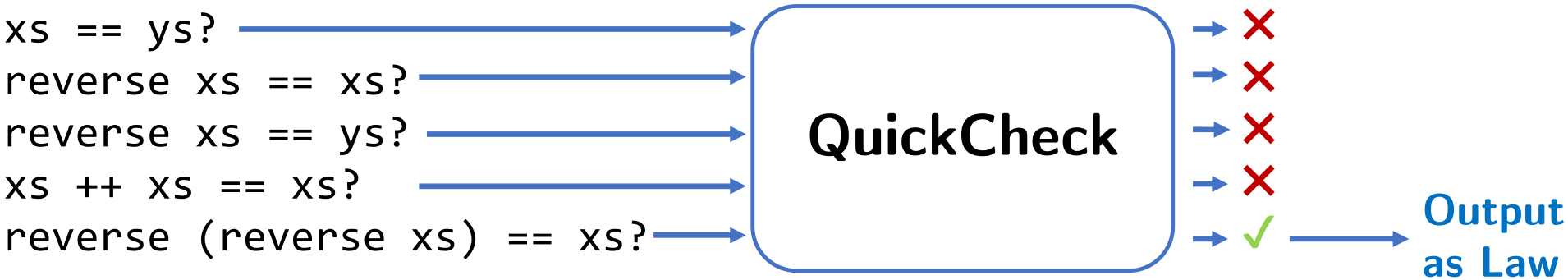
- Uses **constraint solving** and **merging** to guide choices to avoid backtracking
- Allows enumeration of *massive* sets!

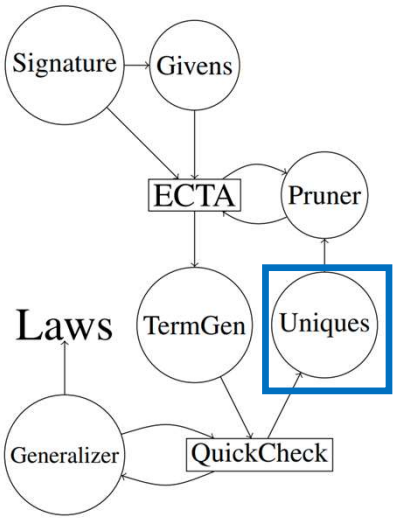
- The **key** difference from **QuickSpec**
- **Filtering** happens *before generation!*



Spectacular: Testing

```
reverse :: [a] -> [a]
(++) :: [a] -> [a] -> [a]
xs :: [a], ys :: [a]
```





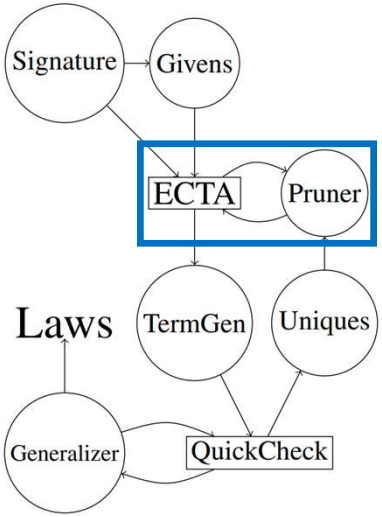
Spectacular: Uniques

```
reverse :: [a] -> [a]
(++) :: [a] -> [a] -> [a]
xs :: [a], ys :: [a]
```

```
xs == ys? ✗
reverse xs == xs? ✗
reverse xs == ys? ✗
xs ++ xs == xs? ✗
reverse (reverse xs) == xs? ✓
```

```
{xs, ys,
 reverse xs,
 xs ++ xs} :: Set [a]
```

No need to compare to reverse (reverse xs) again!



Spectacular: Pruning

```
reverse :: [a] -> [a]
(++) :: [a] -> [a] -> [a]
xs :: [a], ys :: [a]
```

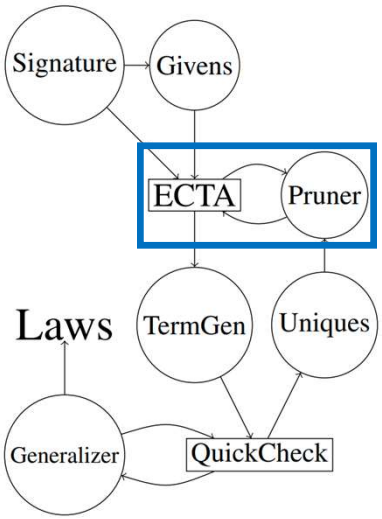
```
xs == ys? ✗
reverse xs == xs? ✗
reverse xs == ys? ✗
xs ++ xs == xs? ✗
reverse (reverse xs) == xs? ✓
reverse (reverse (reverse xs)) == xs?
```

```
{xs, ys,
 reverse xs,
 xs ++ xs} :: Set [a]
```

xs is any list!

We generate by size, so we'll have seen `_ … xs` earlier

We can prune every program containing `_ … (reverse (reverse _)` anywhere!



Spectacular: Generalization

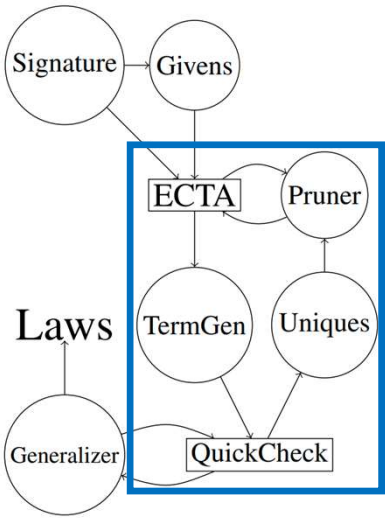
```
reverse :: [a] -> [a]
(++) :: [a] -> [a] -> [a]
xs :: [a], ys :: [a]
```

```
reverse (xs ++ ys) == reverse xs ++ reverse ys? ✗
reverse (xs ++ ys) == reverse ys ++ reverse xs? ✓
```

```
xs == ys? ✗
reverse xs == xs? ✗
reverse xs == ys? ✗
xs ++ xs == xs? ✗
reverse (reverse xs) == xs? ✓
reverse (reverse (reverse xs)) == xs?
```

Output generalized law

```
reverse (xs ++ xs) == reverse xs ++ reverse xs? ✓
```



Spectacular: Phasing

Iterative Discovery

↓

Earlier Pruning

↓

Generalize Findings

↓

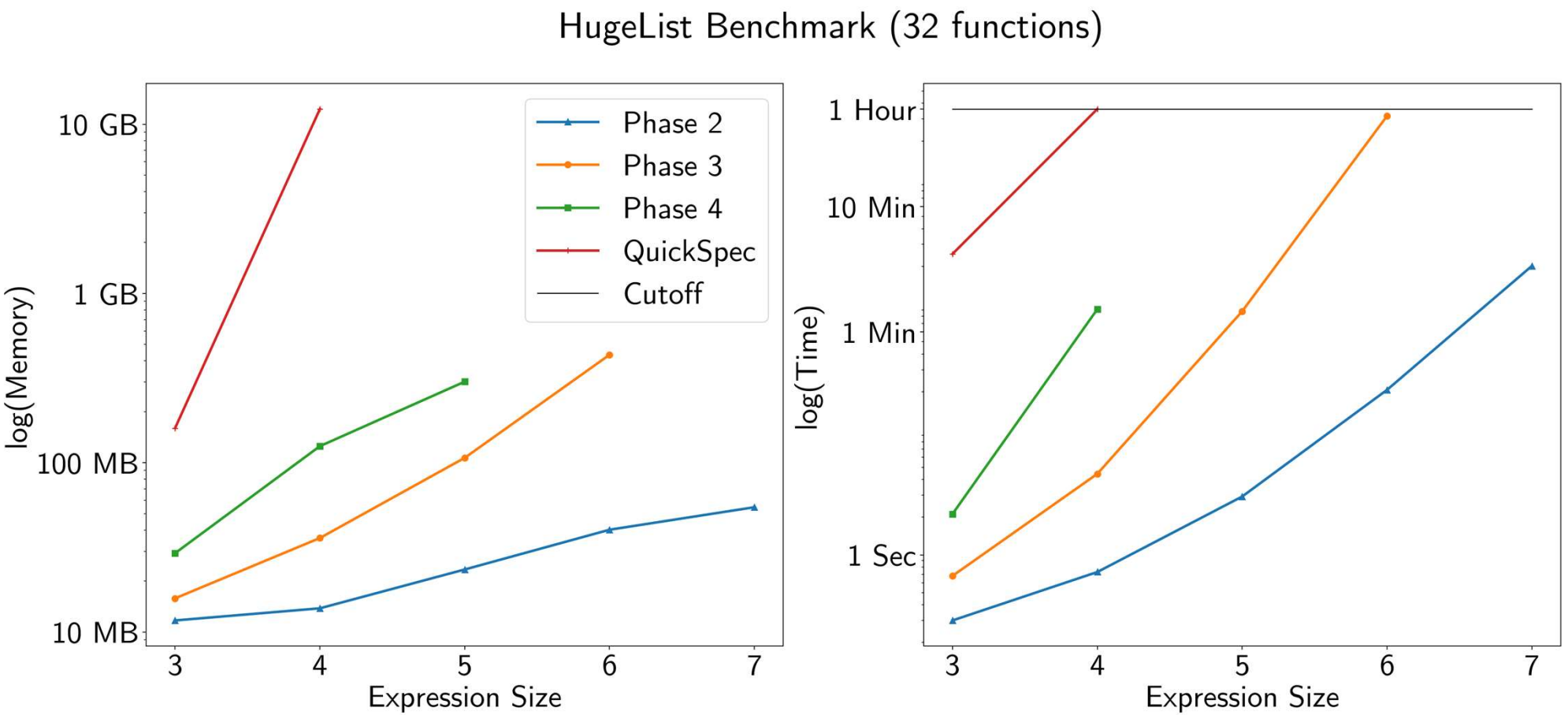
More Efficiency

Spectacular: HugeList

```
con "length" (length :: [A] -> Int),
con "sort" (sort :: [Int] -> [Int]),
con "scanr"
  (scanr :: (A -> B -> B) -> B -> [A] -> [B]),
con "succ" (succ :: Int -> Int),
con ">>="  ((>>=) :: [A] -> (A -> [B]) -> [B]),
con "snd" (snd :: (A, B) -> B),
con "reverse" (reverse :: [A] -> [A]),
con "0" (0 :: Int),
con "," ((,) :: A -> B -> (A, B)),
con ">=>"
 ((>=>) :: (A -> [B]) -> (B -> [C]) -> A -> [C]),
con ":" ((:) :: A -> [A] -> [A]),
con "break"
 (break :: (A -> Bool) -> [A] -> ([A], [A])),
con "filter" (filter :: (A -> Bool) -> [A] -> [A]),
con "scanl"
 (scanl :: (B -> A -> B) -> B -> [A] -> [B]),
con "zipWith"
 (zipWith :: (A -> B -> C) -> [A] -> [B] -> [C]),
con "concat" (concat :: [[A]] -> [A]),
```

```
con "zip"  (zip :: [A] -> [B] -> [(A, B)]),
con "usort" (usort :: [Int] -> [Int]),
con "sum" (sum :: [Int] -> Int),
con "++"  ((++) :: [A] -> [A] -> [A]),
con "map"  (map :: (A -> B) -> [A] -> [B]),
con "foldl"
 (foldl :: (B -> A -> B) -> B -> [A] -> B),
con "takeWhile"
 (takeWhile :: (A -> Bool) -> [A] -> [A]),
con "foldr"
 (foldr :: (A -> B -> B) -> B -> [A] -> B),
con "drop" (drop :: Int -> [A] -> [A]),
con "dropWhile"
 (dropWhile :: (A -> Bool) -> [A] -> [A]),
con "span"
 (span :: (A -> Bool) -> [A] -> ([A], [A])),
con "unzip" (unzip :: [(A, B)] -> ([A], [B])),
con "+" ((+) :: Int -> Int -> Int),
con "[]" ([] :: [A]),
con "partition"
 (partition :: (A -> Bool) -> [A] -> ([A], [A])),
con "fst" (fst :: (A, B) -> A),
con "take" (take :: Int -> [A] -> [A]) ]
```

Spectacular: Results



Thank You!

Iterative Discovery
↓
Early Pruning
↓
Generalize Findings
↓
More Efficiency

Questions?

pallm@chalmers.se