Signature Inference for Functional Property Discovery

or: How never to come up with tests manually anymore(*)

Tom Sydney Kerckhove

ETH Zurich
https://cs-syd.eu/
https://github.com/NorfairKing

11 October 2017

Motivation

Writing correct software is hard for humans.

Unit Testing

```
sort
    [4, 1, 6]
        ==
            [1, 4, 6]
```

Unit Testing

```
    sort
        [4, 1, 6]
            ==
                [1, 4, 6]
```

Property testing

```
      forAll
        arbitrary
          $ \ls ->
            isSorted (sort ls)
```

Property testing

```
forAll
  arbitrary
    $ \ls ->
      isSorted (sort ls)
```

Property testing

```
      forAll
        arbitrary
          $ \ls ->
            isSorted (sort ls)
```

Property Discovery

```
      forAll
        arbitrary
          $ \ls ->
            isSorted (sort ls)
```

Property Discovery with QuickSpec

Example Code

```haskell
module MySort where

mySort :: Ord a => [a] -> [a]
mySort [] = []
mySort (x:xs) = insert (mySort xs)
  where
    insert [] = [x]
    insert (y:ys)
        | x <= y = x : y : ys
        | otherwise = y : insert ys

myIsSorted :: Ord a => [a] -> Bool
myIsSorted [] = True
myIsSorted [_] = True
myIsSorted (x:y:ls) = x <= y && myIsSorted (y : ls)
```

Example Code

```haskell
module MySort where

mySort :: Ord a => [a] -> [a]
mySort [] = []
mySort (x:xs) = insert (mySort xs)
  where
    insert [] = [x]
    insert (y:ys)
        | x <= y = x : y : ys
        | otherwise = y : insert ys

myIsSorted :: Ord a => [a] -> Bool
myIsSorted [] = True
myIsSorted [_] = True
myIsSorted (x:y:ls) = x <= y && myIsSorted (y : ls)
```

Property Discovery using QuickSpec

```
== Signature ==
      True :: Bool
      (<=) :: Ord a => a -> a -> Bool
       (:) :: a -> [a] -> [a]
    mySort :: Ord a => [a] -> [a]
myIsSorted :: Ord a => [a] -> Bool
```

Property Discovery using QuickSpec

```
== Signature ==
      True :: Bool
      (<=) :: Ord a => a -> a -> Bool
       (:) :: a -> [a] -> [a]
    mySort :: Ord a => [a] -> [a]
myIsSorted :: Ord a => [a] -> Bool
```

```
== Laws ==
  1. y <= y = True
  2. y <= True = True
  3. True <= x = x
  4. myIsSorted (mySort xs) = True
  5. mySort (mySort xs) = mySort xs
  6. xs <= mySort xs = myIsSorted xs
  7. mySort xs <= xs = True
  8. myIsSorted (y : (y : xs)) = myIsSorted (y : xs)
  9. mySort (y : mySort xs) = mySort (y : xs)
```

Property Discovery using QuickSpec

```
== Signature ==
      True :: Bool
      (<=) :: Ord a => a -> a -> Bool
       (:) :: a -> [a] -> [a]
    mySort :: Ord a => [a] -> [a]
myIsSorted :: Ord a => [a] -> Bool
```

```
== Laws ==
  1. y <= y = True
  2. y <= True = True
  3. True <= x = x
  4. myIsSorted (mySort xs) = True
  5. mySort (mySort xs) = mySort xs
  6. xs <= mySort xs = myIsSorted xs
  7. mySort xs <= xs = True
  8. myIsSorted (y : (y : xs)) = myIsSorted (y : xs)
  9. mySort (y : mySort xs) = mySort (y : xs)
```

QuickSpec Code

```haskell
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE ConstraintKinds #-}
{-# LANGUAGE RankNTypes #-}
{-# LANGUAGE FlexibleContexts #-}

module MySortQuickSpec where

import Control.Monad
import MySort
import QuickSpec

main :: IO ()
main =
    void $
    quickSpec
        signature
        { constants =
              [ constant "True" (True :: Bool)
              , constant "<=" (mkDict (<=) :: Dict (Ord A) -> A -> A -> Bool)
              , constant ":" ((:) :: A -> [A] -> [A])
              , constant "mySort" (mkDict mySort :: Dict (Ord A) -> [A] -> [A])
              , constant
                    "myIsSorted"
                    (mkDict myIsSorted :: Dict (Ord A) -> [A] -> Bool)
              ]
        }

mkDict ::
       (c =>
            a)
    -> Dict c
    -> a
mkDict x Dict = x
```

Problems with QuickSpec: Monomorphisation

Only for monomorphic functions

```
    constant "filter"
      (filter :: (A -> B) -> [A] -> [B] -> Bool)
```

Problems with QuickSpec: Code

Programmer has to write code for all functions of interest
15 lines of subject code.
33 lines of QuickSpec code.

Problems with QuickSpec: Speed

Dumb version of the QuickSpec approach:

1. Generate all possible terms
2. Generate all possible equations (tuples) of terms
3. Type check them to make sure the equation makes sense
4. Check that the input can be generated and the output compared for equality
5. Run QuickCheck to see if the equation holds

Pause slide with a joke

```
strictId :: a -> a
strictId !x = x
```

Property Discovery with EasySpec

Step 1: Automation

Signatures

```
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE ConstraintKinds #-}
{-# LANGUAGE RankNTypes #-}
{-# LANGUAGE FlexibleContexts #-}

module MySortQuickSpec where

import Control.Monad
import MySort
import QuickSpec

main :: IO ()
main =
    void $
    quickSpec
        signature
        { constants =
              [ constant "True" (True :: Bool)
              , constant "<=" (mkDict (<=) :: Dict (Ord A) -> A -> A -> Bool)
              , constant ":" ((:) :: A -> [A] -> [A])
              , constant "mySort" (mkDict mySort :: Dict (Ord A) -> [A] -> [A])
              , constant
                    "myIsSorted"
                    (mkDict myIsSorted :: Dict (Ord A) -> [A] -> Bool)
              ]
        }

mkDict ::
       (c =>
            a)
    -> Dict c
    -> a
mkDict x Dict = x
```

Signatures

```haskell
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE ConstraintKinds #-}
{-# LANGUAGE RankNTypes #-}
{-# LANGUAGE FlexibleContexts #-}

module MySortQuickSpec where

import Control.Monad
import MySort
import QuickSpec

main :: IO ()
main =
    void $
    quickSpec
        signature
        { constants =
              [ constant "True" (True :: Bool)
              , constant "<=" (mkDict (<=) :: Dict (Ord A) -> A -> A -> Bool)
              , constant ":" ((:) :: A -> [A] -> [A])
              , constant "mySort" (mkDict mySort :: Dict (Ord A) -> [A] -> [A])
              , constant
                    "myIsSorted"
                    (mkDict myIsSorted :: Dict (Ord A) -> [A] -> Bool)
              ]
        }

mkDict ::
       (c =>
            a)
    -> Dict c
    -> a
mkDict x Dict = x
```

A QuickSpec Signature

```
data Signature =
  Signature {
    functions           :: [Function],
    [...]
    properties          :: [Prop],
  }
```

```
quickSpec :: Signature -> IO Signature
```

Signature Expression Generation

Signature Expression Generation

```haskell
filter :: (a -> Bool) -> [a] -> [a]
```

Signature Expression Generation

```
    filter :: (a -> Bool) -> [a] -> [a]
```

```
    filter :: (A -> Bool) -> [A] -> [A]
```

Signature Expression Generation

```haskell
filter :: (a -> Bool) -> [a] -> [a]
```

```haskell
filter :: (A -> Bool) -> [A] -> [A]
```

```haskell
function "filter"
  (filter :: (A -> Bool) -> [A] -> [A])
```

Signature Expression Generation

```
filter :: (a -> Bool) -> [a] -> [a]
```

```
filter :: (A -> Bool) -> [A] -> [A]
```

```
function "filter"
  (filter :: (A -> Bool) -> [A] -> [A])
```

```
signature { constants = [...] }
```

Current situation

```
$ cat Reverse.hs
{-# LANGUAGE NoImplicitPrelude #-}

module Reverse where

import Data.List (reverse, sort)
```

Current situation

```
$ cat Reverse.hs
```

```haskell
{-# LANGUAGE NoImplicitPrelude #-}

module Reverse where

import Data.List (reverse, sort)
```

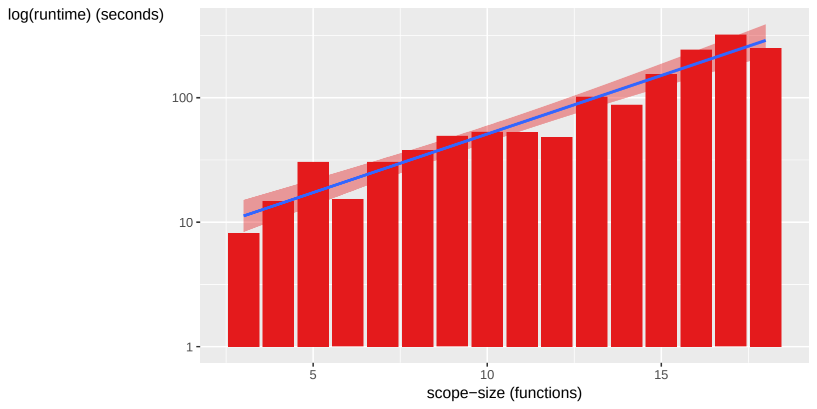
```
$ easyspec discover Reverse.hs
```

```
    reverse (reverse xs) = xs
    sort (reverse xs) = sort xs
```

Pause slide with a joke

```
safePerformIO :: IO a -> IO a
safePerformIO ioa = ioa >>= return
```

Automated, but still slow



Definitions

Definitions: Property

Example:

```
    reverse (reverse ls) = ls
```

Short for:

```
    (\ls -> reverse (reverse ls)) = (\ls -> ls)
```

In general:

```
    (f :: A -> B) = (g :: A -> B)
    for some A and B with
    instance Arbitrary A
    instance Eq B
```

Why is this slow?

1. Maximum size of the discovered properties

Why is this slow?

1. Maximum size of the discovered properties
2. Size of the signature

Idea

Critical insight

We are not interested in the entire codebase.

We are interested in a relatively small amount of code.

Reducing the size of the signature

```haskell
inferSignature
  :: [Function] -- Focus functions
  -> [Function] -- Functions in scope
  -> [Function] -- Chosen functions
```

Full background and empty background

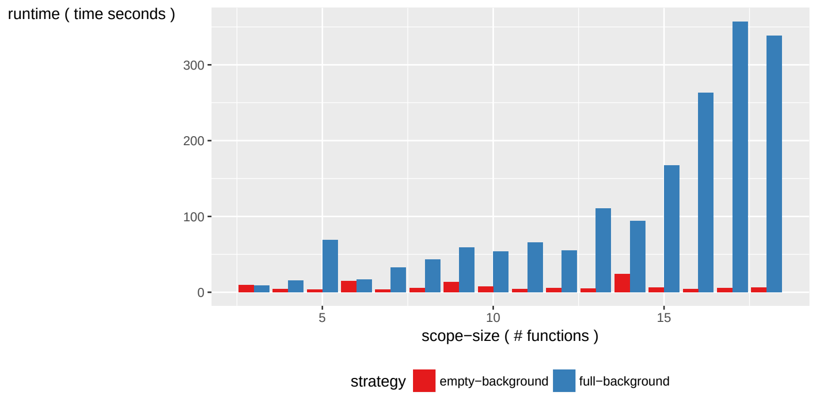
```
    inferFullBackground _ scope = scope
```

```
    inferEmptyBackground focus _ = focus
```

Full background and empty background

```
inferFullBackground _ scope = scope
```

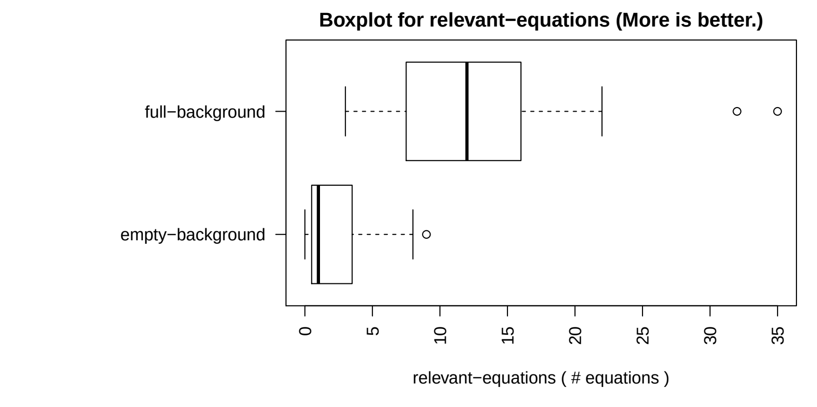
```
inferEmptyBackground focus _ = focus
```



Full background and empty background

```
inferFullBackground _ scope = scope
```

```
inferEmptyBackground focus _ = focus
```



Pause slide with a joke

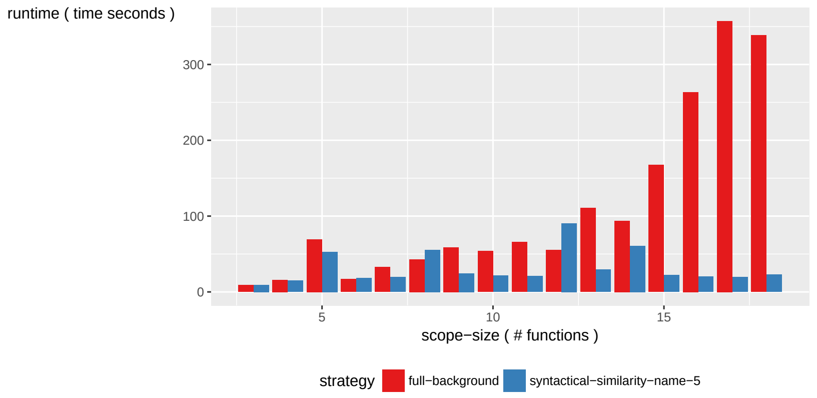
```haskell
safeCoerce :: a ~ b => a -> b
safeCoerce x = x
```

Syntactic similarity: Name

```
inferSyntacticSimilarityName [focus] scope
    = take 5 $ sortOn
      (\sf ->
        distance
          (name focus) (name sf))
      scope
```

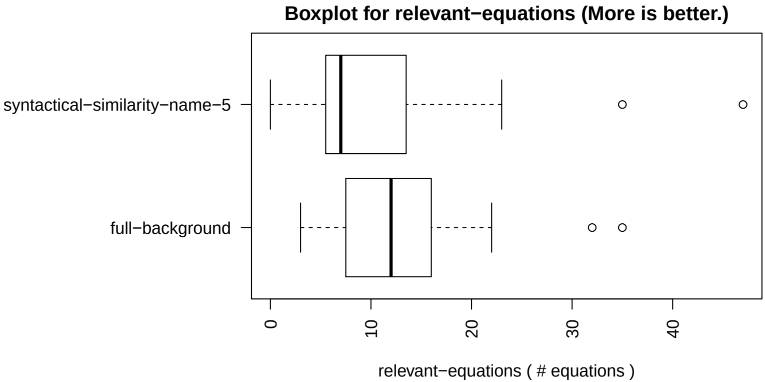
Syntactic similarity: Name

```
inferSyntacticSimilarityName [focus] scope
    = take 5 $ sortOn
      (\sf ->
        distance
          (name focus) (name sf))
      scope
```



Syntactic similarity: Name

```
inferSyntacticSimilarityName [focus] scope
    = take 5 $ sortOn
      (\sf ->
        distance
          (name focus) (name sf))
      scope
```

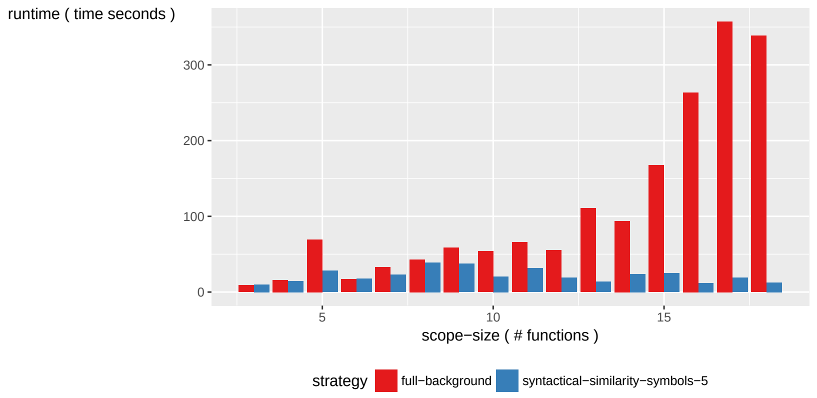

Boxplot for relevant−equations (More is better.)

Syntactic similarity: Implementation

```haskell
inferSyntacticSimilaritySymbols i [focus] scope
    = take i $ sortOn
      (\sf ->
        distance
          (symbols focus) (symbols sf))
      scope
```

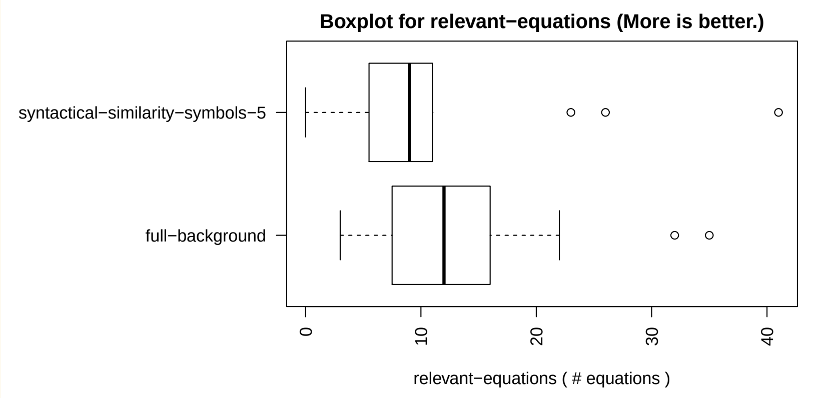
Syntactic similarity: Implementation

```
inferSyntacticSimilaritySymbols i [focus] scope
    = take i $ sortOn
      (\sf ->
        distance
          (symbols focus) (symbols sf))
      scope
```



Syntactic similarity: Implementation

```
inferSyntacticSimilaritySymbols i [focus] scope
    = take i $ sortOn
      (\sf ->
        distance
          (symbols focus) (symbols sf))
      scope
```

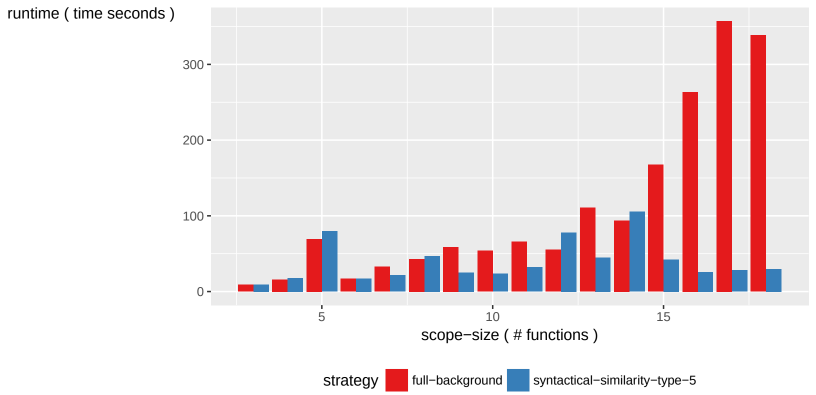


Syntactic similarity: Type

```
inferSyntacticSimilarityType i [focus] scope
    = take i $ sortOn
      (\sf ->
        distance
          (typeParts focus) (typeParts sf))
      scope
```

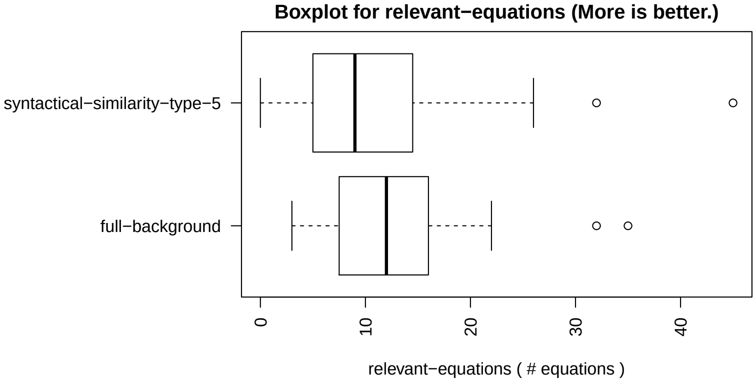
Syntactic similarity: Type

```
inferSyntacticSimilarityType i [focus] scope
  = take i $ sortOn
   (\sf ->
    distance
     (typeParts focus) (typeParts sf))
   scope
```

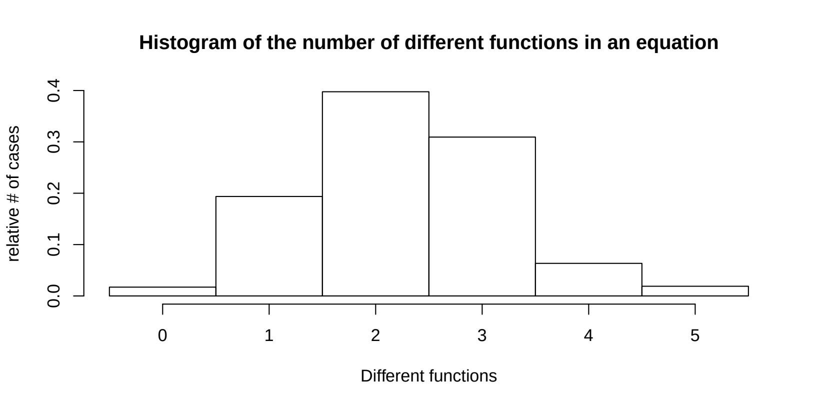


Syntactic similarity: Type

```
inferSyntacticSimilarityType i [focus] scope
    = take i $ sortOn
      (\sf ->
        distance
          (typeParts focus) (typeParts sf))
      scope
```



Breakthrough



Idea

We can run QuickSpec more than once!

Inferred Signature

```
    type SignatureInferenceStrategy
        = [Function] -> [Function] -> InferredSignature
```

Inferred Signature

```
type SignatureInferenceStrategy
    = [Function] -> [Function] -> InferredSignature
```

Combine the results of multiple runs:

```
type InferredSignature = [Signature]
```

Inferred Signature

```
type SignatureInferenceStrategy
    = [Function] -> [Function] -> InferredSignature
```

Combine the results of multiple runs:

```
type InferredSignature = [Signature]
```

User previous results as background properties:

```
type InferredSignature = Forest Signature
```

Inferred Signature

```
type SignatureInferenceStrategy
    = [Function] -> [Function] -> InferredSignature
```

Combine the results of multiple runs:

```
type InferredSignature = [Signature]
```

User previous results as background properties:

```
type InferredSignature = Forest Signature
```

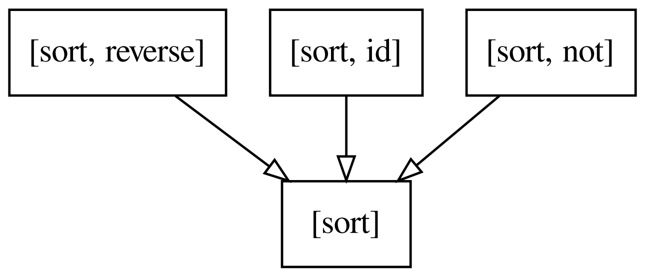
Share previous runs:

```
type InferredSignature = DAG Signature
```

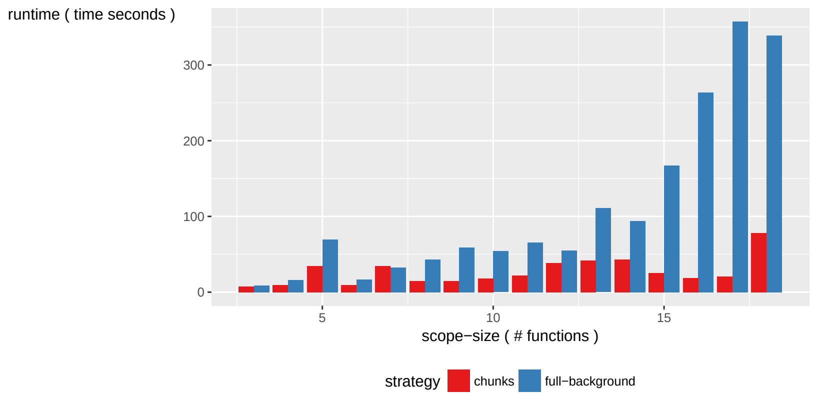
Chunks

```
    chunks :: SignatureInferenceStrategy
```

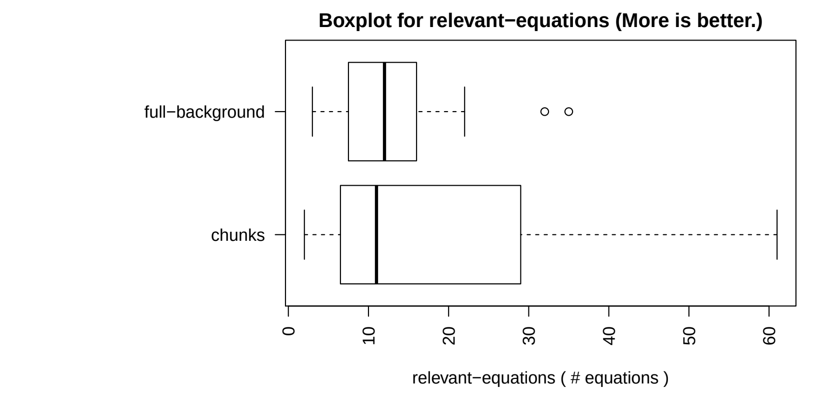
```
    > chunks
    >     [sort :: Ord a => [a] -> [a]]
    >     [reverse :: [a] -> [a], id :: a -> a]
```



The runtime of chunks



The outcome of chunks: Relevant equations



Inferred Signature

```
type SignatureInferenceStrategy
    = [Function] -> [Function] -> InferredSignature



type InferredSignature =
    DAG ([(Signature, [Equation])] -> Signature)
```

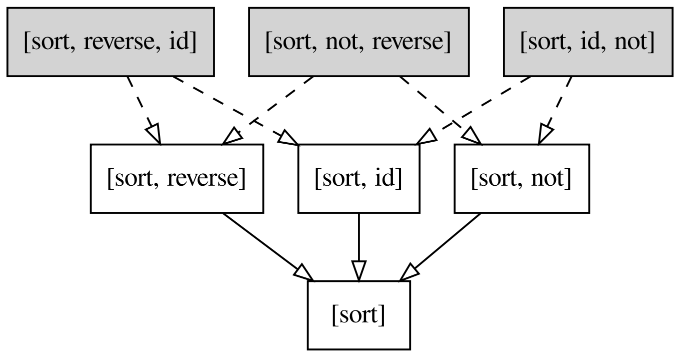
Inferred Signature

```haskell
type SignatureInferenceStrategy
    = [Function] -> [Function] -> InferM ()

data InferM a where
    InferPure :: a -> InferM a
    InferFmap :: (a -> b) -> InferM a -> InferM b
    InferApp :: InferM (a -> b) -> InferM a -> InferM b
    InferBind :: InferM a -> (a -> InferM b) -> InferM b

    InferFrom
        :: Signature
        -> [OptiToken]
        -> InferM (OptiToken, [Equation])
```

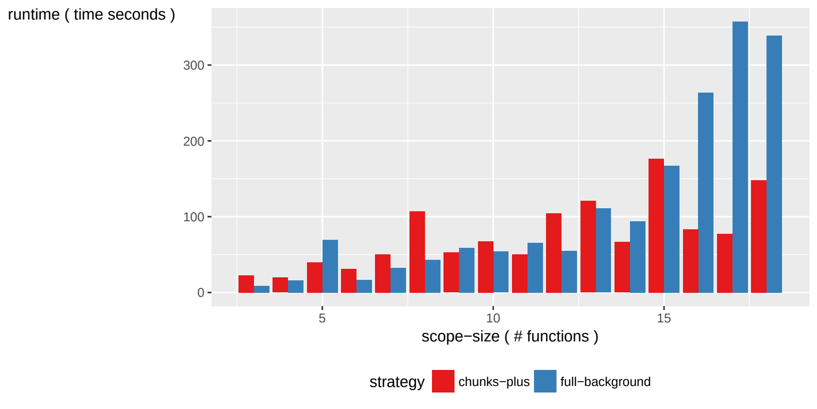
Chunks Plus

```
    chunksPlus :: SignatureInferenceStrategy
```

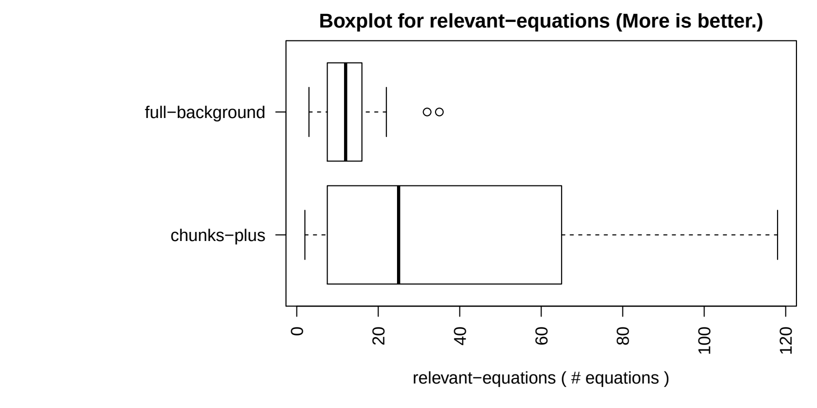
```
   > chunksPlus
   >     [sort :: Ord a => [a] -> [a]]
   >     [reverse :: [a] -> [a], id :: a -> a]
```



The runtime of chunks plus



The outcome of chunks plus: Relevant equations



Neat

```
$ time stack exec easyspec \
      -- discover MySort.hs MySort.mySort

xs <= mySort xs = myIsSorted xs
mySort xs <= xs = True
myIsSorted (mySort xs) = True
mySort (mySort xs) = mySort xs

3.61s user 1.14s system 193% cpu 2.450 total
```

Great promise, but ...

Great promise, but ...

1. Only works for functions in scope of which the type is in scope too.

Great promise, but ...

1. Only works for functions in scope of which the type is in scope too.
2. Crashes on partial functions.

Great promise, but ...

1. Only works for functions in scope of which the type is in scope too.
2. Crashes on partial functions.
3. Only works with built in instances.

Great promise, but ...

1. Only works for functions in scope of which the type is in scope too.
2. Crashes on partial functions.
3. Only works with built in instances.
4. Data has to have an Arbitrary instance in scope.

Great promise, but ...

1. Only works for functions in scope of which the type is in scope too.
2. Crashes on partial functions.
3. Only works with built in instances.
4. Data has to have an Arbitrary instance in scope.
5. Does not play with CPP.

Great promise, but ...

1. Only works for functions in scope of which the type is in scope too.
2. Crashes on partial functions.
3. Only works with built in instances.
4. Data has to have an Arbitrary instance in scope.
5. Does not play with CPP.
6. Does not play well with higher kinded type variables.

Great promise, but ...

1. Only works for functions in scope of which the type is in scope too.
2. Crashes on partial functions.
3. Only works with built in instances.
4. Data has to have an Arbitrary instance in scope.
5. Does not play with CPP.
6. Does not play well with higher kinded type variables.

All technical problems, not theoretical problems!

Further Research

1. Can we go faster?

Further Research

1. Can we go faster?
2. Which constants do we choose for built in types?

Further Research

1. Can we go faster?
2. Which constants do we choose for built in types?
3. Can we apply this to effectful code?

Further Research

1. Can we go faster?
2. Which constants do we choose for built in types?
3. Can we apply this to effectful code?
4. Relative importance of equations

Call to action

Proofs of concept:

```
    https://github.com/nick8325/quickcheck
    https://github.com/nick8325/quickspec

    https://github.com/NorfairKing/easyspec
```

Now we need to make it production ready!

About Me

This was my master thesis
Wrote Haskell in open source
Taught Haskell at ETH
Haskell and DevOps in industry

`https://cs-syd.eu/`
`https://cs-syd.eu/cv`
`https://github.com/NorfairKing`