Signature Inference for Functional Property Discovery

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

Tom Sydney Kerckhove

FP Complete
https://cs-syd.eu/
https://github.com/NorfairKing
https://fpcomplete.com

2018-02-22

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

```
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

```
{-# 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 -> Bool) -> [A] -> [A])
```

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

Property Discovery with EasySpec

Step 1: Automation

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
```

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],
    [...]
    background          :: [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

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

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

```
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

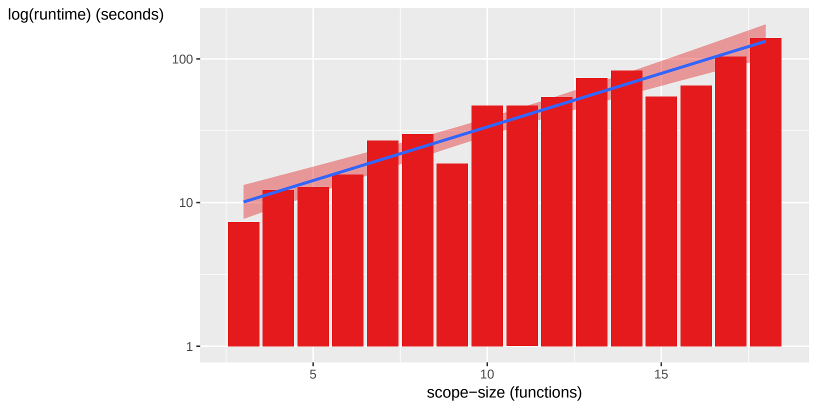
```
$ cat Reverse.hs
```

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

module Reverse where

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

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

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

Automated, but still slow



Definition: 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

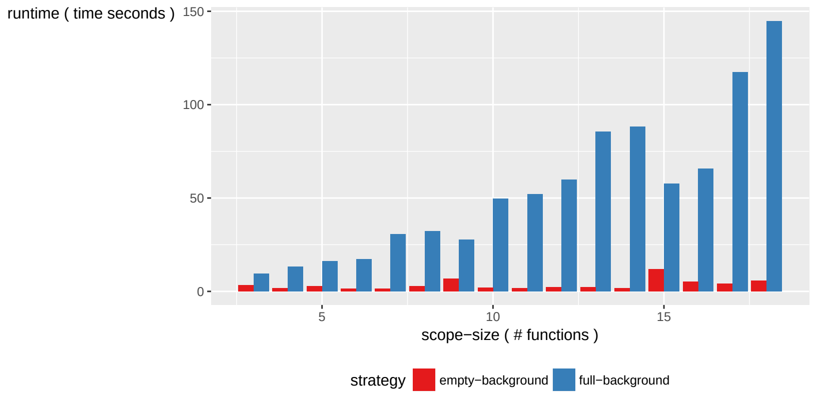
```
    inferFullBackground _ scope = scope
```

```
    inferEmptyBackground focus _ = focus
```

Full Background and Empty Background

```
inferFullBackground _ scope = scope
```

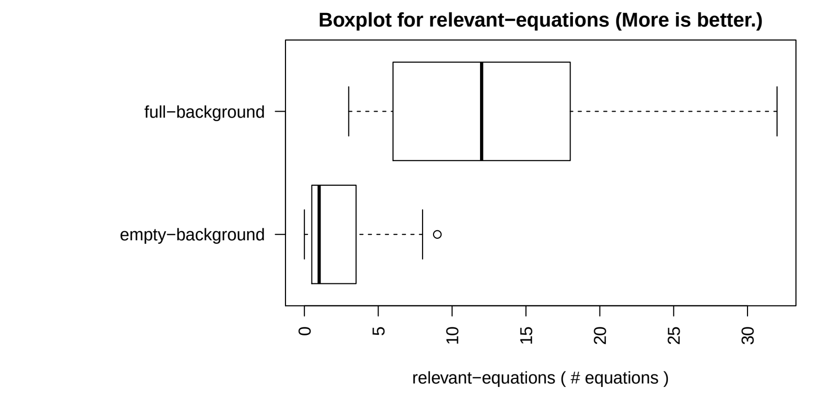
```
inferEmptyBackground focus _ = focus
```



Full Background and Empty Background

```
inferFullBackground _ scope = scope
```

```
inferEmptyBackground focus _ = focus
```

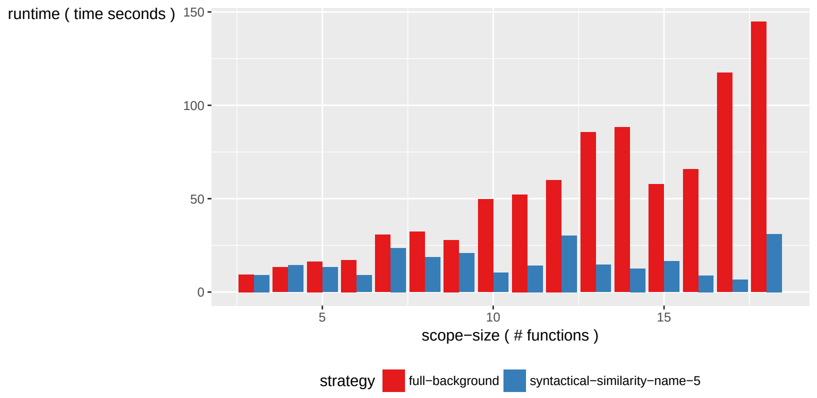


Syntactic Similarity: Name

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

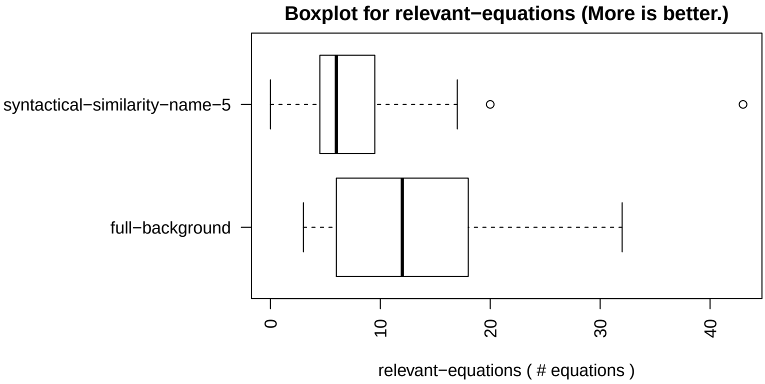
Syntactic Similarity: Name

```
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
```

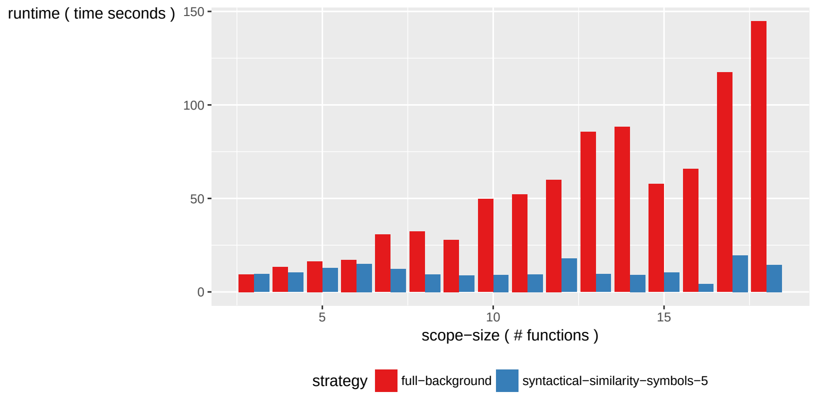


Syntactic Similarity: Implementation

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

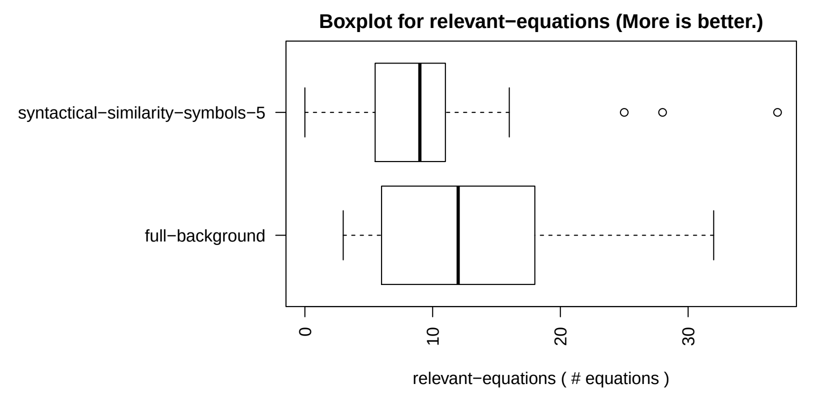
Syntactic Similarity: Implementation

```
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
```

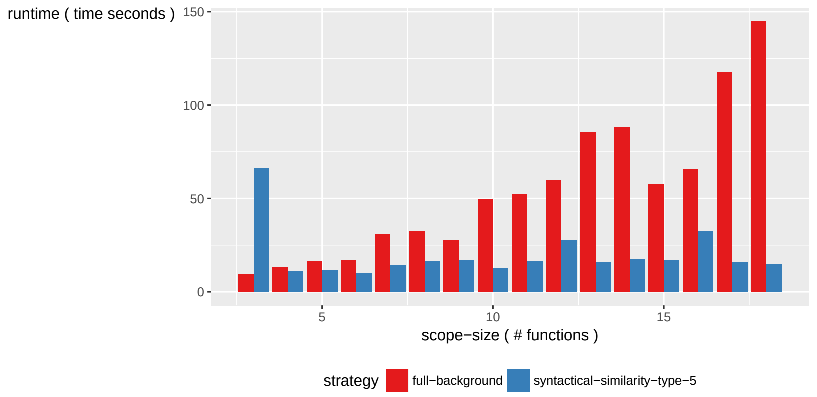


Syntactic Similarity: Type

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

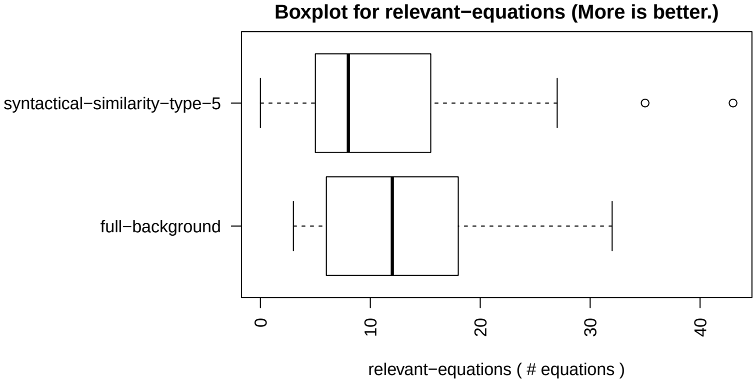
Syntactic Similarity: Type

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



Syntactic Similarity: Type

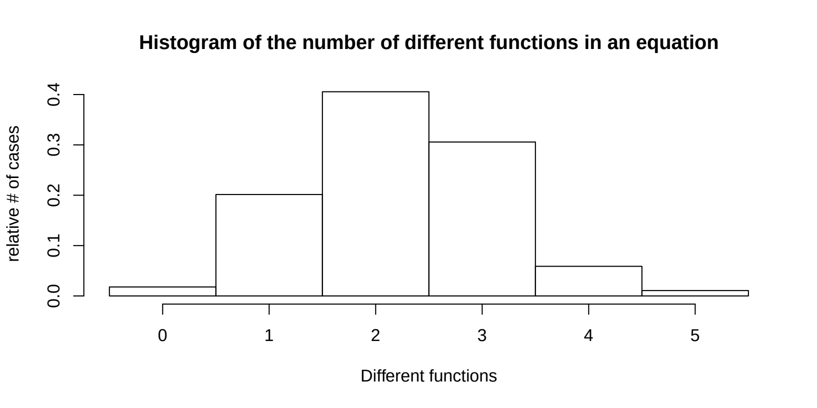
```
inferSyntacticSimilarityType i [focus] scope
    = take i $ sortOn
      (\sf ->
        distance
          (getTypeParts focus) (getTypeParts sf))
      scope
```



Other Things we Tried

1. Similarity using a different metric: edit distance
2. Unions of the previous strategies

Breakthrough



Idea

We can run QuickSpec more than once!

Inferred Signature

Combine the results of multiple runs:

```
    [Signature]
```

Inferred Signature

Combine the results of multiple runs:

```
    [Signature]
```

User previous results as background properties:

```
    Forest Signature
```

Inferred Signature

Combine the results of multiple runs:

```
    [Signature]
```

User previous results as background properties:

```
    Forest Signature
```

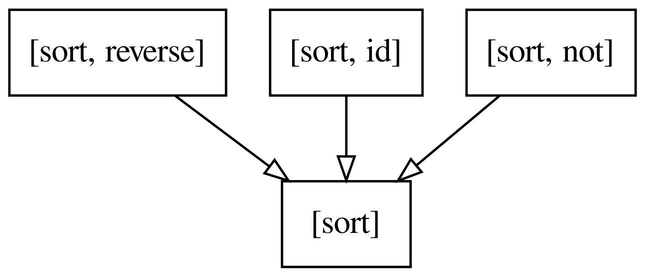
Share previous runs:

```
    DAG Signature
```

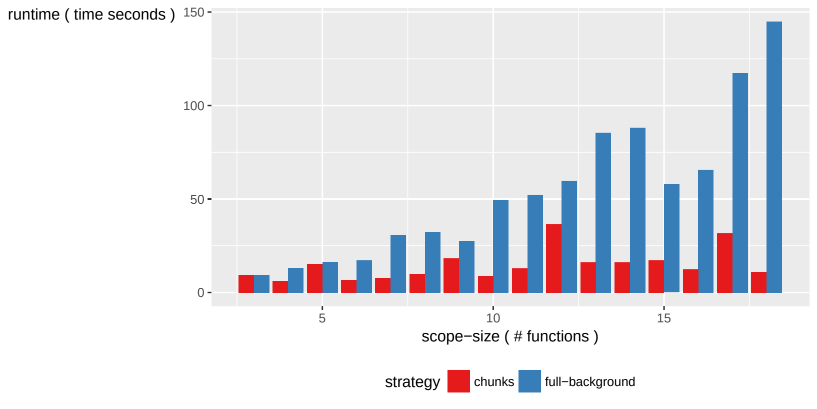
Chunks

```
    chunks :: SignatureInferenceStrategy
```

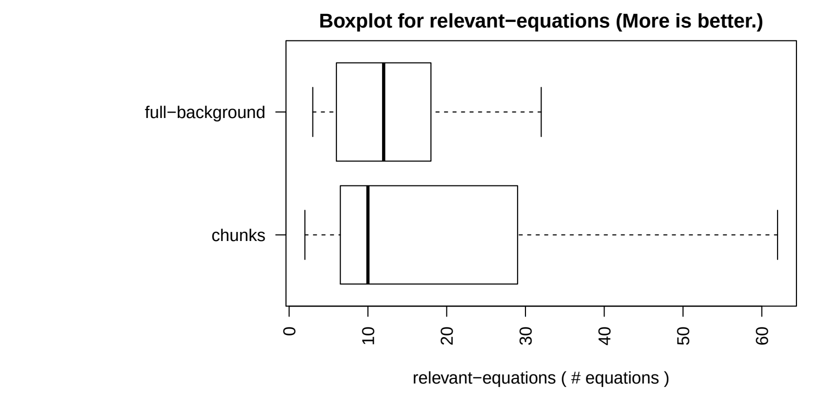
```
> chunks
>     [sort :: Ord a => [a] -> [a]]
>     [reverse :: [a] -> [a], id :: a -> a]
```



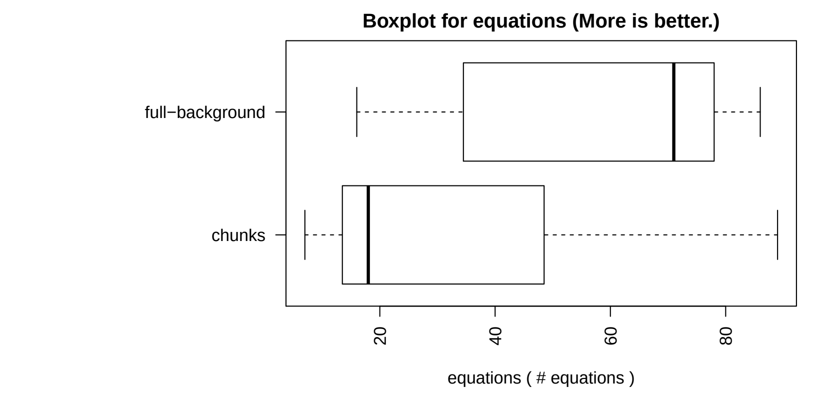
The Runtime of Chunks



The Outcome of Chunks: Relevant equations



Why does chunks find more relevant equations?



Why does chunks find more relevant equations?

Scope:

```
    a = (+ 1)
    b = (+ 2)
    c = (+ 3)
    d = (+ 4)
```

Why does chunks find more relevant equations?

Scope:

```
    a = (+ 1)
    b = (+ 2)
    c = (+ 3)
    d = (+ 4)
```

Full background:

```
    a (a x) = b x
    a (b x) = c x
    a (c x) = d x
```

Relevant to d:

```
    a (c x) = d x
```

Why does chunks find more relevant equations?

Scope:

```
a = (+ 1)
b = (+ 2)
c = (+ 3)
d = (+ 4)
```

Full background:

```
a (a x) = b x
a (b x) = c x
a (c x) = d x
```

Relevant to d:

```
a (c x) = d x
```

Chunks for d:

```
b (b x) = d x
a (a (a (a x))) = d x
```

All relevant

Inferred Signature

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



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

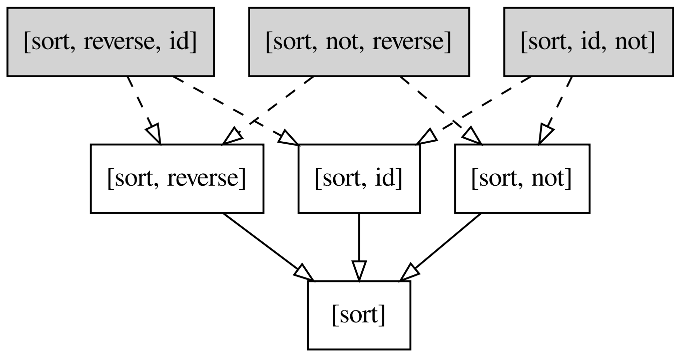
Inferred Signature

```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])
```

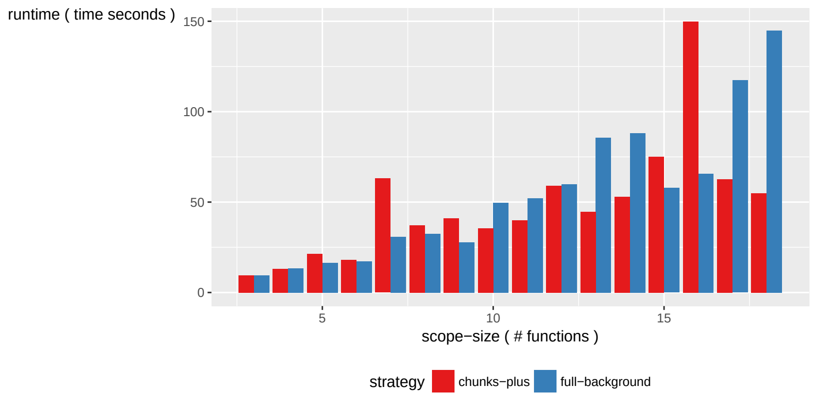
Chunks Plus

```
    chunksPlus :: SignatureInferenceStrategy
```

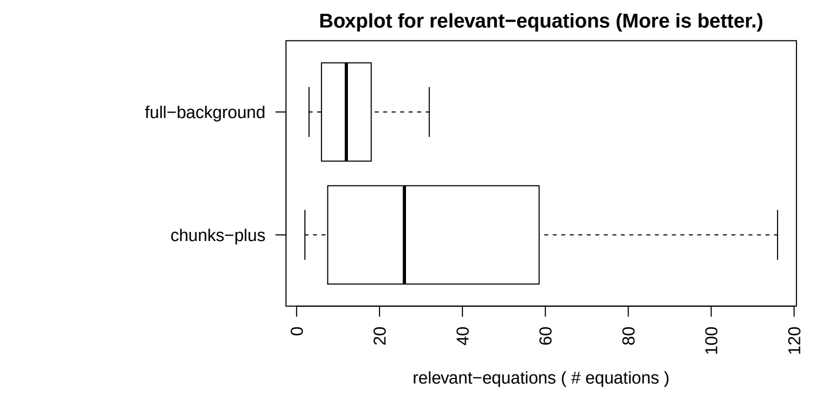
```
    > chunksPlus
    >     [sort :: Ord a => [a] -> [a]]
    >     [reverse :: [a] -> [a], id :: a -> a]
```



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
```

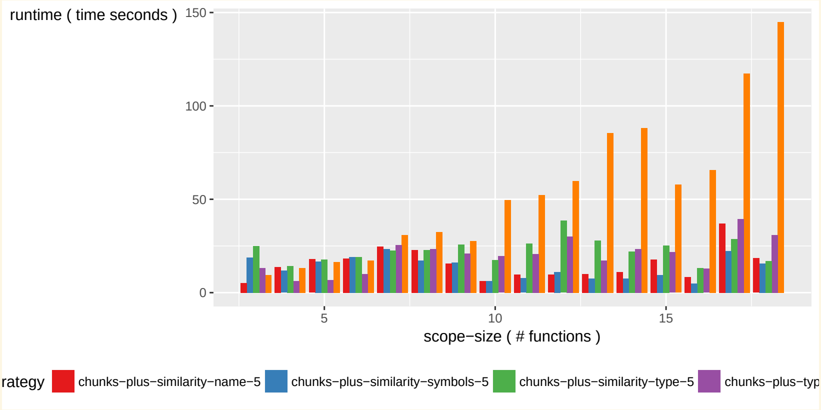
Composing Strategies

```
type Reducing
    = [Function] -> [Function] -> [Function]

type Drilling
    = [Function] -> [Function] -> InferM ()
```

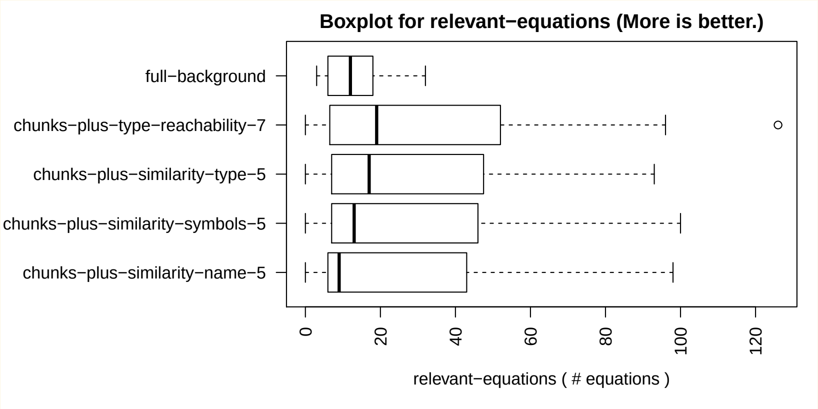
Composing Strategies

```haskell
composeReducings :: Reducing -> Reducing -> Reducing
composeReducings r1 r2 focus = r2 focus . r1 focus

composeDrillings :: Drilling -> Drilling -> Drilling
composeDrillings d1 d2 focus scope = do
    d1 focus scope
    d2 focus scope

composeReducingWithDrilling
    :: Reducing -> Drilling -> Drilling
composeReducingWithDrilling r d focus scope
    = d focus $ r focus scope
```

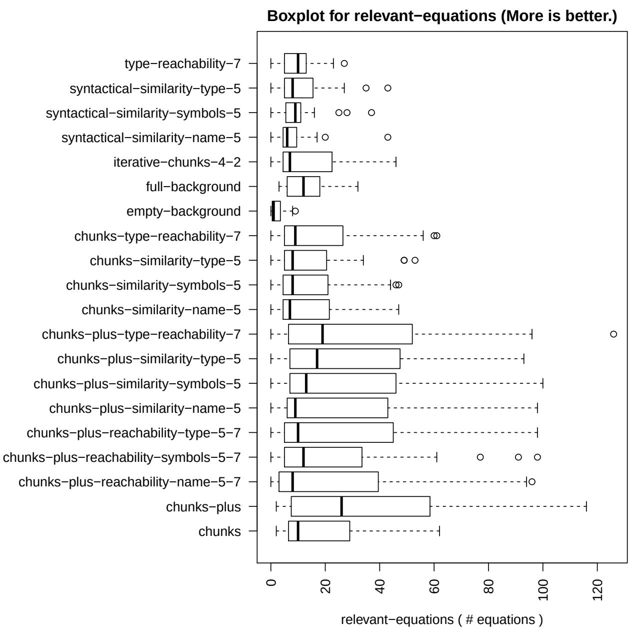
The runtime of chunks plus composed with reducings



The outcome of chunks plus composed with reducings: Relevant equations



All strategies



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

Signature Inference for Functional Property Discovery

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

Tom Sydney Kerckhove

FP Complete
https://cs-syd.eu/
https://github.com/NorfairKing
https://fpcomplete.com

2018-02-22