Signature Inference for Functional Property Discovery

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

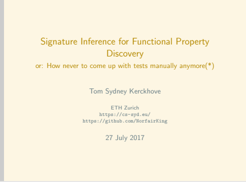
Tom Sydney Kerckhove

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

27 July 2017

2017-07-25

Signature Inference for Functional Property Discovery



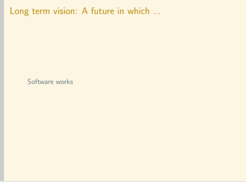
1. The presentation should take about one hour.
2. I have been working on this for the last four and a half months, so if I forget to explain anything, please ask me immediately.

Long term vision: A future in which ...

1. I am not happy with the state of software today.
2. Maybe I'm just an annoying user, but I find that software very often doesn't work.
3. The reason, I think, is that it is often cheaper to make software that only sort of works, at least in the short term.

Long term vision: A future in which ...

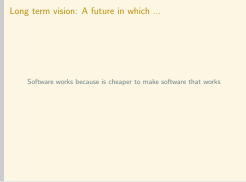
Software works


Long term vision: A future in which ...
Software works

1. I am not happy with the state of software today.
2. Maybe I'm just an annoying user, but I find that software very often doesn't work.
3. The reason, I think, is that it is often cheaper to make software that only sort of works, at least in the short term.

Long term vision: A future in which ...

Software works because is cheaper to make software that works

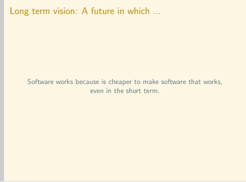

Long term vision: A future in which ...

Software works because is cheaper to make software that works

1. I am not happy with the state of software today.
2. Maybe I'm just an annoying user, but I find that software very often doesn't work.
3. The reason, I think, is that it is often cheaper to make software that only sort of works, at least in the short term.

Long term vision: A future in which ...

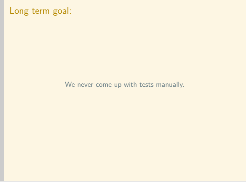
Software works because is cheaper to make software that works, even in the short term.


Long term vision: A future in which ...
Software works because is cheaper to make software that works, even in the short term.

1. I am not happy with the state of software today.
2. Maybe I'm just an annoying user, but I find that software very often doesn't work.
3. The reason, I think, is that it is often cheaper to make software that only sort of works, at least in the short term.

Long term goal:

We never come up with tests manually.



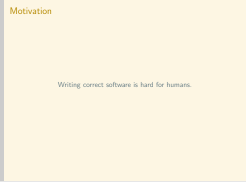
1. Spoiler: we are well on our way, and I'm going to show you a significant step in that direction.

Motivation

1. So why would we want to not want to come up with tests manually?

Motivation

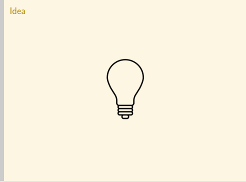
Writing correct software is hard for humans.


Motivation

Writing correct software is hard for humans.

1. So why would we want to not want to come up with tests manually?

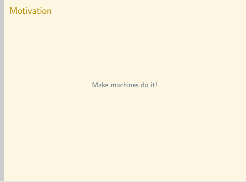
Idea



1. Here is an idea:

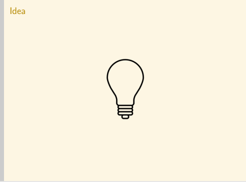
Motivation

Make machines do it!



1. It turns out that making machines write software is hard.
2. I read on hacker news: One day we will only have to give the machine a precise description of what we want code to do, and the machine will write it for us.
3. Well, we are already there. This precise description is called the code.

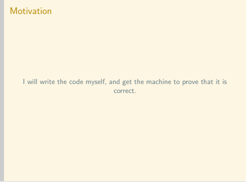
Idea



1. Alright, so maybe we cannot make machines write the code. New idea then.

Motivation

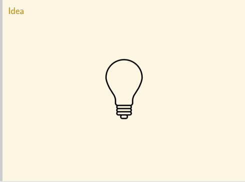
I will write the code myself, and get the machine to prove that it is correct.



1. There are a few problems with this.
2. First of all, you will run into Rice's theorem at some point.
3. Second, you have to already know exactly what it means for your code to be correct.
4. I argue that, in practice, formal methods will not solve the problem that writing correct code is expensive in the short term.

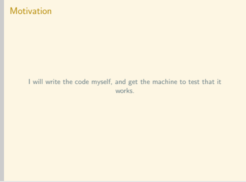
Idea

2017-07-25



Motivation

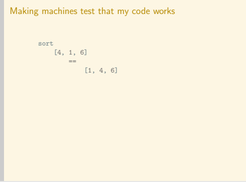
I will write the code myself, and get the machine to test that it works.



1. When formal methods are too expensive, what do we turn to? Testing!

Making machines test that my code works

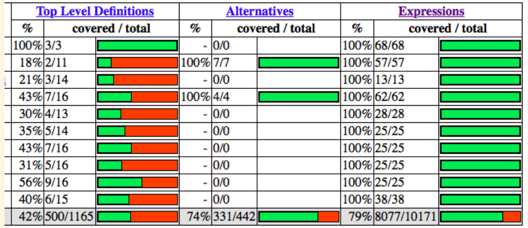
```
sort
    [4, 1, 6]
        ==
            [1, 4, 6]
```


Making machines test that my code works

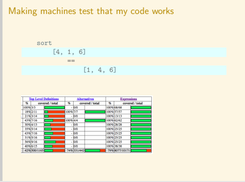
```
sort
    [4, 1, 6]
        ==
            [1, 4, 6]
```

Making machines test that my code works

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

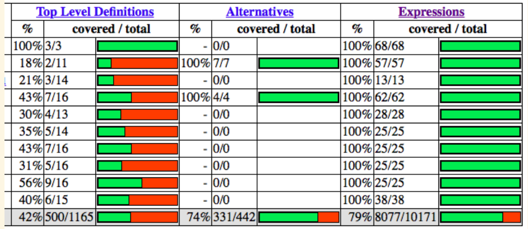


Top Level Definitions			Alternatives			Expressions		
%	covered / total		%	covered / total		%	covered / total	
100%	3/3	🟩	-	0/0		100%	68/68	🟩
18%	2/11	🟩🟥	100%	7/7	🟩	100%	57/57	🟩
21%	3/14	🟩🟥	-	0/0		100%	13/13	🟩
43%	7/16	🟩🟥	100%	4/4	🟩	100%	62/62	🟩
30%	4/13	🟩🟥	-	0/0		100%	28/28	🟩
35%	5/14	🟩🟥	-	0/0		100%	25/25	🟩
43%	7/16	🟩🟥	-	0/0		100%	25/25	🟩
31%	5/16	🟩🟥	-	0/0		100%	25/25	🟩
56%	9/16	🟩🟥	-	0/0		100%	25/25	🟩
40%	6/15	🟩🟥	-	0/0		100%	38/38	🟩
42%	500/1165	🟩🟥	74%	331/442	🟩🟥	79%	8077/10171	🟩🟥



Making machines test that my code works

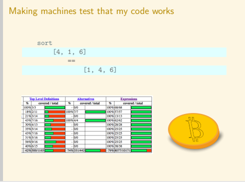
```
    sort
        [4, 1, 6]
            ==
                [1, 4, 6]
```



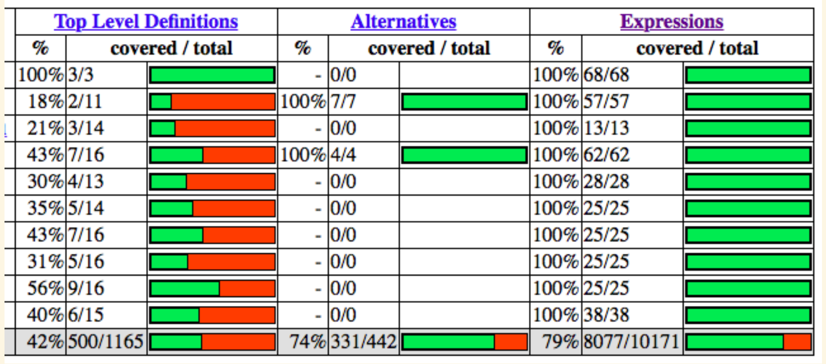
Top Level Definitions			Alternatives			Expressions		
%	covered / total		%	covered / total		%	covered / total	
100%	3/3	🟩	-	0/0		100%	68/68	🟩
18%	2/11	🟩🟥	100%	7/7	🟩	100%	57/57	🟩
21%	3/14	🟩🟥	-	0/0		100%	13/13	🟩
43%	7/16	🟩🟥	100%	4/4	🟩	100%	62/62	🟩
30%	4/13	🟩🟥	-	0/0		100%	28/28	🟩
35%	5/14	🟩🟥	-	0/0		100%	25/25	🟩
43%	7/16	🟩🟥	-	0/0		100%	25/25	🟩
31%	5/16	🟩🟥	-	0/0		100%	25/25	🟩
56%	9/16	🟩🟥	-	0/0		100%	25/25	🟩
40%	6/15	🟩🟥	-	0/0		100%	38/38	🟩
42%	500/1165	🟩🟥	74%	331/442	🟩🟥	79%	8077/10171	🟩🟥

2017-07-25

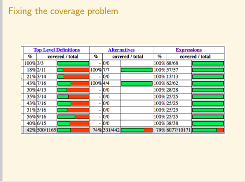
Signature Inference for Functional Property Discovery
└─Motivation
  └─Making machines test that my code works



Fixing the coverage problem

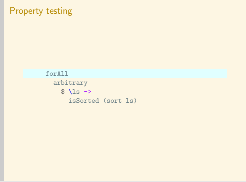


Top Level Definitions			Alternatives			Expressions		
%	covered / total		%	covered / total		%	covered / total	
100%	3/3	🟩	-	0/0		100%	68/68	🟩
18%	2/11	🟩🟥	100%	7/7	🟩	100%	57/57	🟩
21%	3/14	🟩🟥	-	0/0		100%	13/13	🟩
43%	7/16	🟩🟥	100%	4/4	🟩	100%	62/62	🟩
30%	4/13	🟩🟥	-	0/0		100%	28/28	🟩
35%	5/14	🟩🟥	-	0/0		100%	25/25	🟩
43%	7/16	🟩🟥	-	0/0		100%	25/25	🟩
31%	5/16	🟩🟥	-	0/0		100%	25/25	🟩
56%	9/16	🟩🟥	-	0/0		100%	25/25	🟩
40%	6/15	🟩🟥	-	0/0		100%	38/38	🟩
42%	500/1165	🟩🟥	74%	331/442	🟩🟥	79%	8077/10171	🟩🟥



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 testing

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

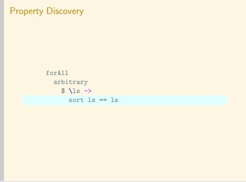
Property testing

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

Fixing the cost problem

Property Discovery

```
forAll
  arbitrary
    $ \ls ->
      sort ls == ls
```

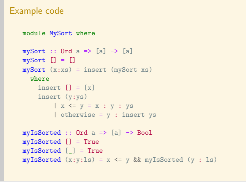


Property Discovery with QuickSpec

Property Discovery with QuickSpec

Example code

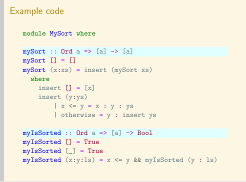
```
module MySort where

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)
```


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)
```

Example code

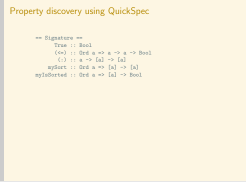
```haskell
module MySort where

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

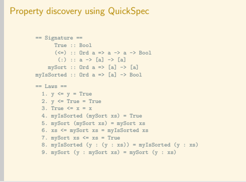
```
== Signature ==
      True :: Bool
      (<=) :: Ord a => a -> a -> Bool
       (:) :: a -> [a] -> [a]
    mySort :: Ord a => [a] -> [a]
myIsSorted :: Ord a => [a] -> Bool
```



1. Explain how would you use this
2. Before I go on:
3. This is Really cool!
4. Really good at what it does
5. Great foundation for what comes next

Property discovery using QuickSpec

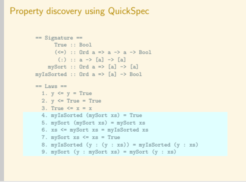
```
== Signature ==
      True :: Bool
      (<=) :: Ord a => a -> a -> Bool
       (:) :: a -> [a] -> [a]
    mySort :: Ord a => [a] -> [a]
myIsSorted :: Ord a => [a] -> Bool

== 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)
```



1. Explain how would you use this
2. Before I go on:
3. This is Really cool!
4. Really good at what it does
5. Great foundation for what comes next

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)
```

1. Explain how would you use this
2. Before I go on:
3. This is Really cool!
4. Really good at what it does
5. Great foundation for what comes next

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
```

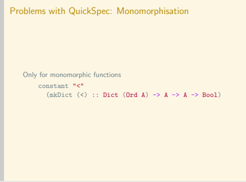
QuickSpec Code

```
{-# 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 "<"
  (mkDict (<) :: Dict (Ord A) -> A -> A -> Bool)
```



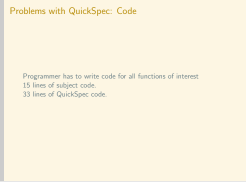
Problems with QuickSpec: Monomorphisation

Only for monomorphic functions

```
constant "<"
  (mkDict (<) :: Dict (Ord A) -> A -> A -> 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.

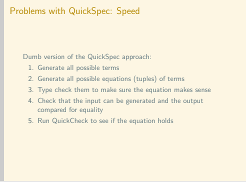

Problems with QuickSpec: Code

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

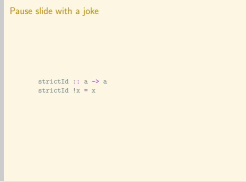

Problems with QuickSpec: Speed

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
```


Pause slide with a joke

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

Property Discovery with EasySpec

Step 1: Automation

Step 1: Automation

Signatures

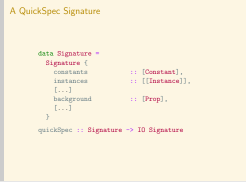
```haskell
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE ConstraintKinds #-}
{-# LANGUAGE RankNTypes #-}
{-# LANGUAGE FlexibleContexts #-}

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 {
    constants          :: [Constant],
    instances          :: [[Instance]],
    [...]
    background         :: [Prop],
    [...]
  }

quickSpec :: Signature -> IO Signature
```


A QuickSpec Signature

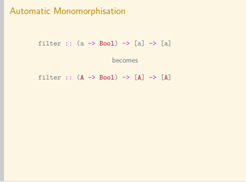
```
data Signature =
  Signature {
    constants          :: [Constant],
    instances          :: [[Instance]],
    [...]
    background         :: [Prop],
    [...]
  }

quickSpec :: Signature -> IO Signature
```

Automatic Monomorphisation

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

becomes

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


Automatic Monomorphisation

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

becomes

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

Automatic Monomorphisation

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

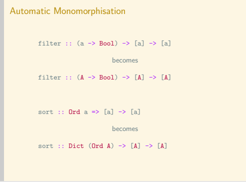
becomes

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

```
sort :: Ord a => [a] -> [a]
```

becomes

```
sort :: Dict (Ord A) -> [A] -> [A]
```


Automatic Monomorphisation

filter :: (a -> Bool) -> [a] -> [a]
becomes
filter :: (A -> Bool) -> [A] -> [A]

sort :: Ord a => [a] -> [a]
becomes
sort :: Dict (Ord A) -> [A] -> [A]

Signature Expression Generation

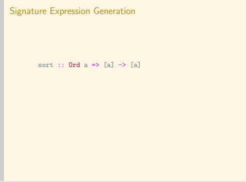
2017-07-25

Signature Inference for Functional Property Discovery
└─Automation
└─Signature Expression Generation

Signature Expression Generation

Signature Expression Generation

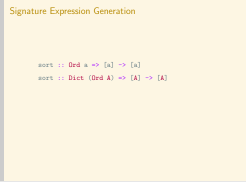
```
sort :: Ord a => [a] -> [a]
```


Signature Expression Generation

sort :: Ord a => [a] -> [a]

Signature Expression Generation

```
sort :: Ord a => [a] -> [a]
sort :: Dict (Ord A) => [A] -> [A]
```

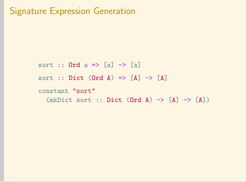


Signature Expression Generation

```
sort :: Ord a => [a] -> [a]
sort :: Dict (Ord A) => [A] -> [A]
```

Signature Expression Generation

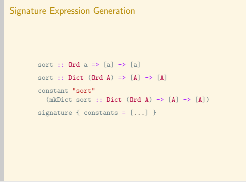
```
sort :: Ord a => [a] -> [a]

sort :: Dict (Ord A) => [A] -> [A]

constant "sort"
  (mkDict sort :: Dict (Ord A) -> [A] -> [A])
```


Signature Expression Generation

```
sort :: Ord a => [a] -> [a]

sort :: Dict (Ord A) => [A] -> [A]

constant "sort"
  (mkDict sort :: Dict (Ord A) -> [A] -> [A])
```

Signature Expression Generation

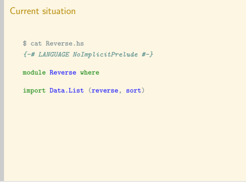
```
sort :: Ord a => [a] -> [a]

sort :: Dict (Ord A) => [A] -> [A]

constant "sort"
  (mkDict sort :: Dict (Ord A) -> [A] -> [A])

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


Signature Expression Generation

```
sort :: Ord a => [a] -> [a]
sort :: Dict (Ord A) => [A] -> [A]
constant "sort"
  (mkDict sort :: Dict (Ord A) -> [A] -> [A])
signature { constants = [...] }
```

Current situation

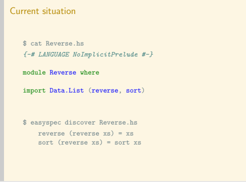
```
$ cat Reverse.hs
{-# LANGUAGE NoImplicitPrelude #-}

module Reverse where

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


Current situation

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

module Reverse where

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

Current situation

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

module Reverse where

import Data.List (reverse, sort)



$ easyspec discover Reverse.hs
    reverse (reverse xs) = xs
    sort (reverse xs) = sort xs
```

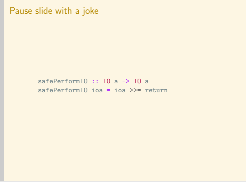


Pause slide with a joke

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

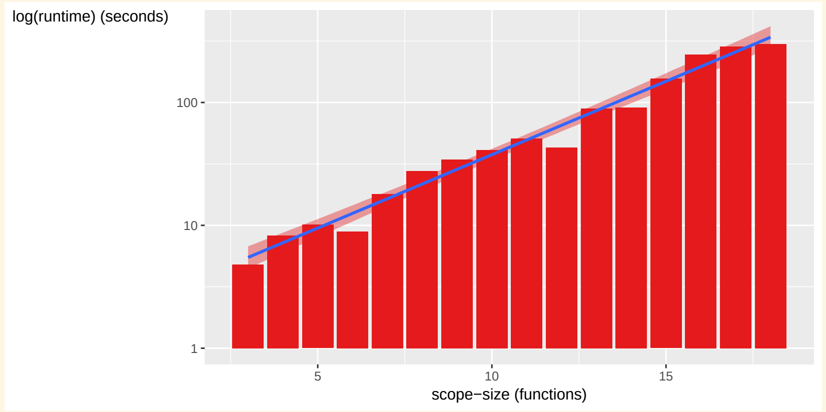
2017-07-25

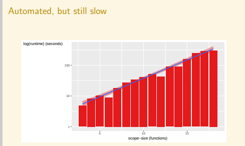
Signature Inference for Functional Property Discovery
└─Automation
└─Pause slide with a joke


Pause slide with a joke

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

Automated, but still slow





1. Now we have automated QuickSpec, but it still slow

Definitions

Definitions: Property

Example:

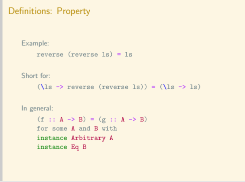
```
reverse (reverse ls) = ls
```

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
```



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
```

Definitions: Size of property

Example:

```
xs <= mySort xs = myIsSorted xs
```

Definitions: Size of property

Example:

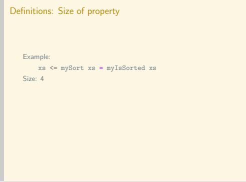
```
xs <= mySort xs = myIsSorted xs
```

Definitions: Size of property

Example:

```
xs <= mySort xs = myIsSorted xs
```

Size: 4


Definitions: Size of property

Example:

```
xs <= mySort xs = myIsSorted xs
```

Size: 4

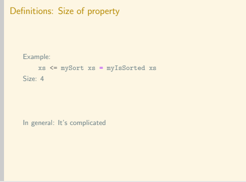
Definitions: Size of property

Example:

```
    xs <= mySort xs = myIsSorted xs
```

Size: 4

In general: It's complicated



Definitions: Size of property

Example:

```
    xs <= mySort xs = myIsSorted xs
```

Size: 4

In general: It's complicated

Definitions: Property of a function

Functions:

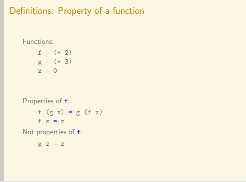
```
f = (* 2)
g = (* 3)
z = 0
```

Properties of `f`:

```
f (g x) = g (f x)
f z = z
```

Not properties of `f`:

```
g z = z
```



Definitions: Property of a function

Functions:

```
f = (* 2)
g = (* 3)
z = 0
```

Properties of f:

```
f (g x) = g (f x)
f z = z
```

Not properties of f:

```
g z = z
```

Definitions: Relevant function

Functions:

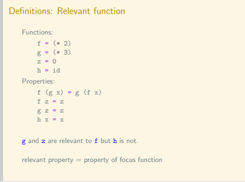
```
f = (* 2)
g = (* 3)
z = 0
h = id
```

Properties:

```
f (g x) = g (f x)
f z = z
g z = z
h x = x
```

`g` and `z` are relevant to `f` but `h` is not.

relevant property = property of focus function


Definitions: Relevant function

Functions:

```
f = (* 2)
g = (* 3)
z = 0
h = id
```

Properties:

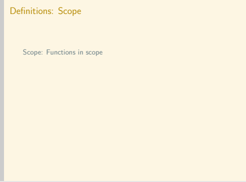
```
f (g x) = g (f x)
f z = z
g z = z
h x = x
```

`g` and `z` are relevant to `f` but `h` is not.

relevant property = property of focus function

Definitions: Scope

Scope: Functions in scope

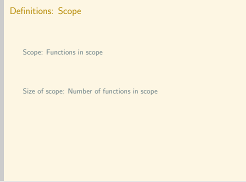

Definitions: Scope

Scope: Functions in scope

Definitions: Scope

Scope: Functions in scope

Size of scope: Number of functions in scope


Definitions: Scope

Scope: Functions in scope

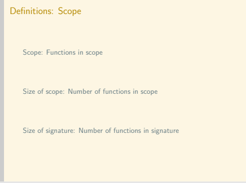
Size of scope: Number of functions in scope

Definitions: Scope

Scope: Functions in scope

Size of scope: Number of functions in scope

Size of signature: Number of functions in signature

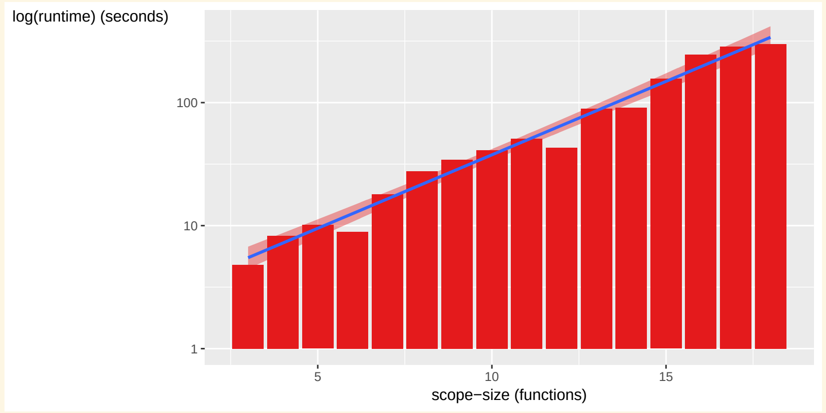

Definitions: Scope

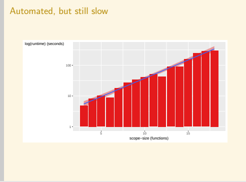
Scope: Functions in scope

Size of scope: Number of functions in scope

Size of signature: Number of functions in signature

Automated, but still slow

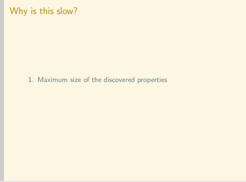




1. We set out to find eighty percent of the properties in twenty percent of the time.
2. Of course, later we realised that even twenty percent does not change the time complexity and therefore is too slow in practice.

Why is this slow?

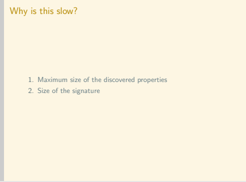
1. Maximum size of the discovered properties


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


Why is this slow?
1. Maximum size of the discovered properties
2. Size of the signature

Idea

2017-07-25

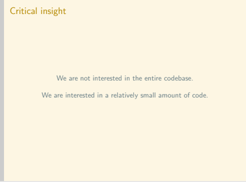
Signature Inference for Functional Property Discovery
└─Signature Inference
  └─Idea

Idea

Critical insight

We are not interested in the entire codebase.

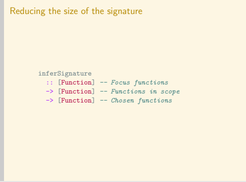
We are interested in a relatively small amount of code.



1. This means that we have an entirely different goal than QuickSpec
2. Comparisons with QuickSpec are not really fair, but we have nothing else to compare to

Reducing the size of the signature

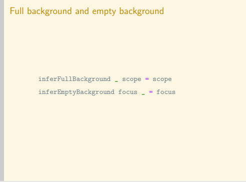
```
inferSignature
  :: [Function] -- Focus functions
  -> [Function] -- Functions in scope
  -> [Function] -- Chosen functions
```


Reducing the size of the signature

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

Full background and empty background

```
inferFullBackground _ scope = scope

inferEmptyBackground focus _ = focus
```

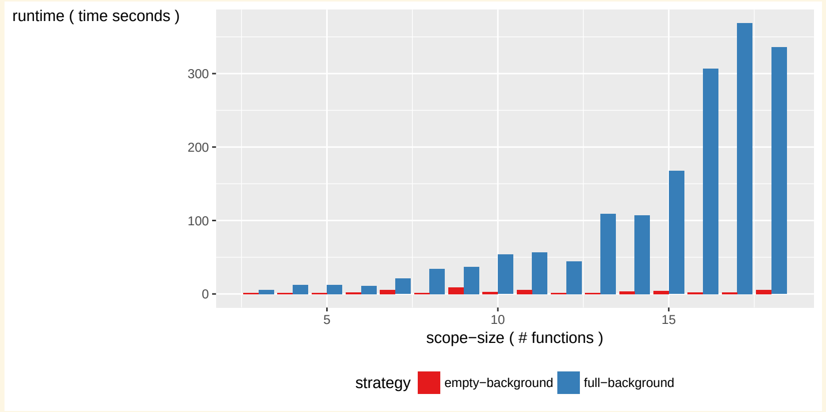


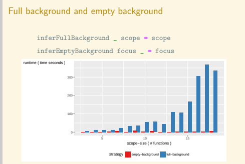
Full background and empty background

```
inferFullBackground _ scope = scope
inferEmptyBackground focus _ = focus
```

Full background and empty background

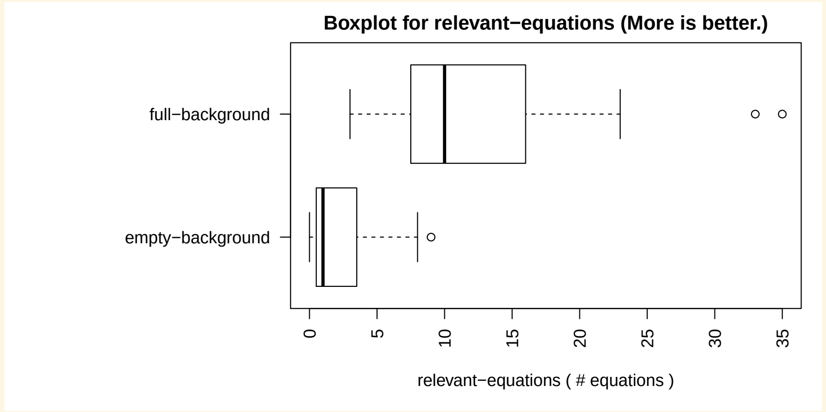
```
inferFullBackground _ scope = scope
inferEmptyBackground focus _ = focus
```

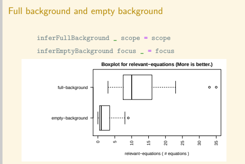




Full background and empty background

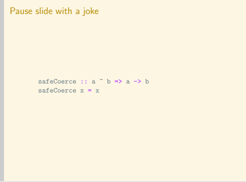
```
inferFullBackground _ scope = scope

inferEmptyBackground focus _ = focus
```





Pause slide with a joke

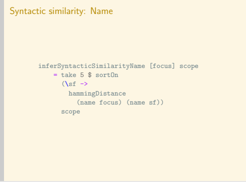
```
safeCoerce :: a ~ b => a -> b
safeCoerce x = x
```


Pause slide with a joke

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

Syntactic similarity: Name

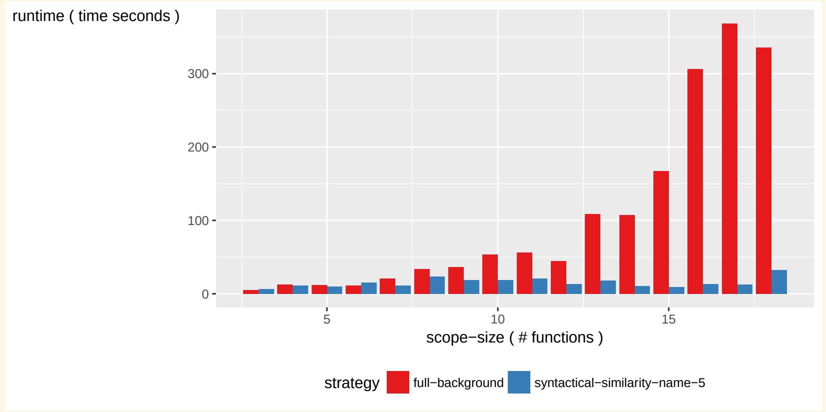
```
inferSyntacticSimilarityName [focus] scope
    = take 5 $ sortOn
      (\sf ->
        hammingDistance
          (name focus) (name sf))
      scope
```


Syntactic similarity: Name

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

Syntactic similarity: Name

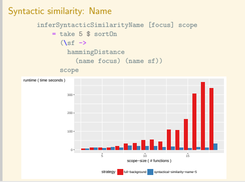
```
inferSyntacticSimilarityName [focus] scope
    = take 5 $ sortOn
      (\sf ->
        hammingDistance
          (name focus) (name sf))
      scope
```



2017-07-25

Signature Inference for Functional Property Discovery
└Signature Inference

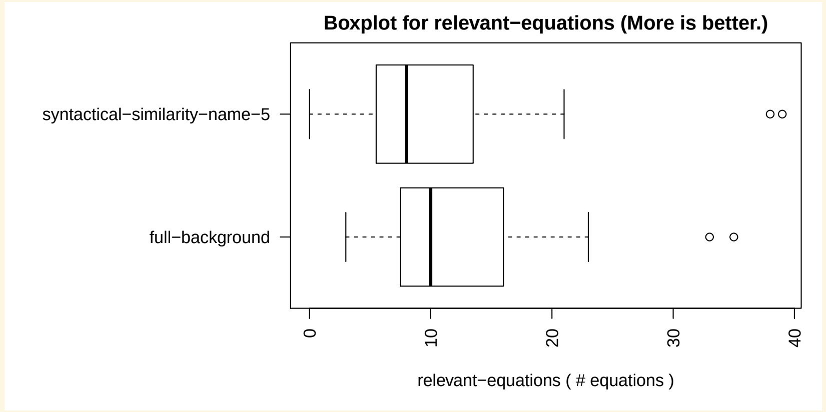
└Syntactic similarity: Name

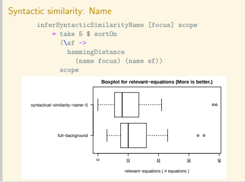

Syntactic similarity: Name

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

Syntactic similarity: Name

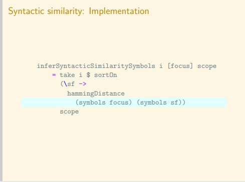
```
inferSyntacticSimilarityName [focus] scope
    = take 5 $ sortOn
      (\sf ->
        hammingDistance
          (name focus) (name sf))
      scope
```





Syntactic similarity: Implementation

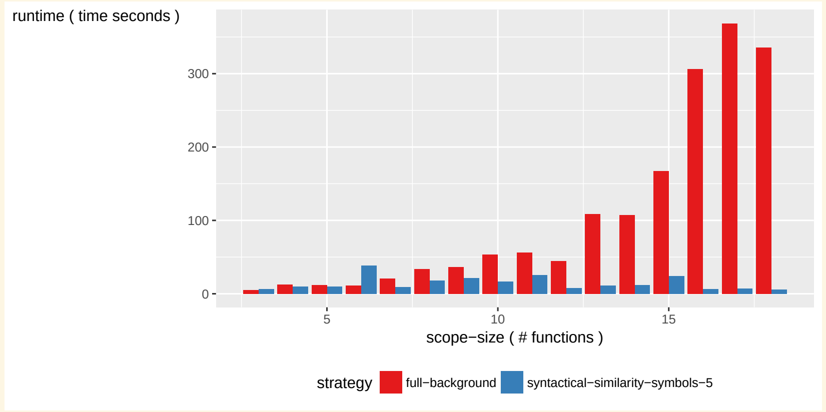
```
inferSyntacticSimilaritySymbols i [focus] scope
    = take i $ sortOn
      (\sf ->
        hammingDistance
          (symbols focus) (symbols sf))
      scope
```

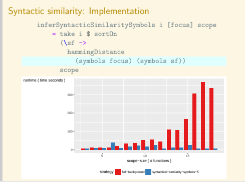

Syntactic similarity: Implementation

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

Syntactic similarity: Implementation

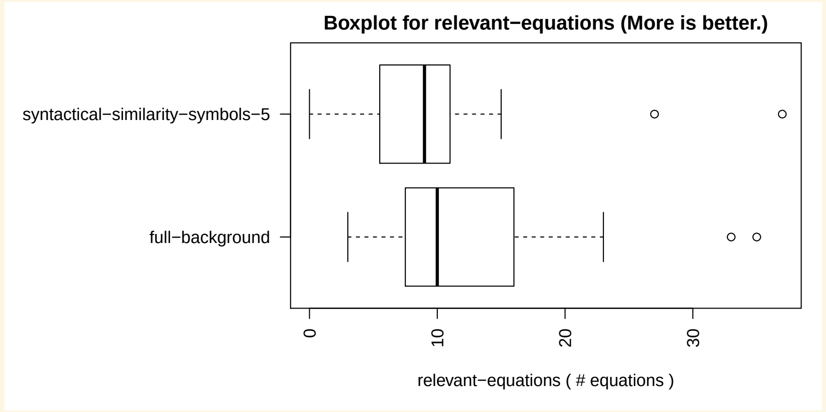
```
inferSyntacticSimilaritySymbols i [focus] scope
    = take i $ sortOn
      (\sf ->
        hammingDistance
          (symbols focus) (symbols sf))
      scope
```

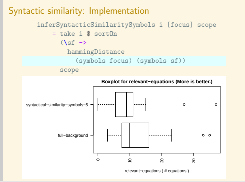




Syntactic similarity: Implementation

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



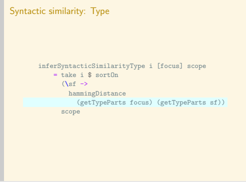


Syntactic similarity: Type

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

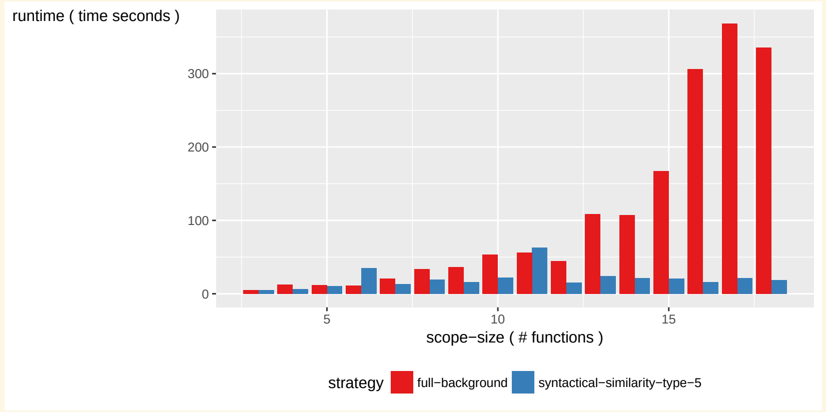
2017-07-25

Signature Inference for Functional Property Discovery
└Signature Inference
  └Syntactic similarity: Type



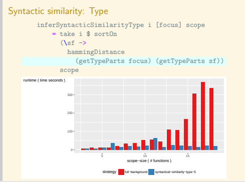
Syntactic similarity: Type

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



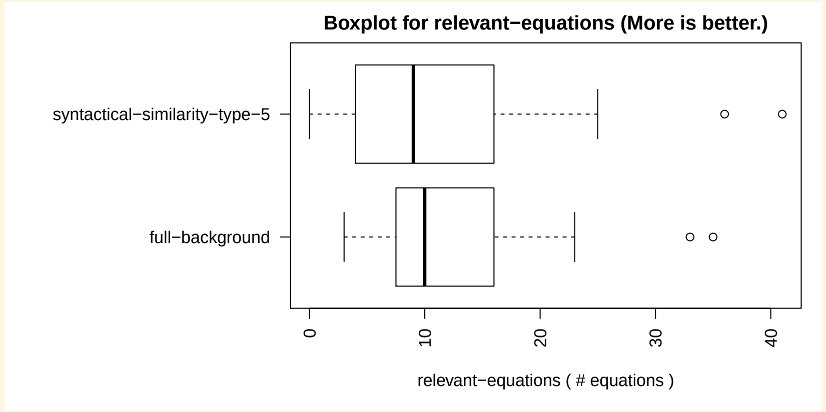
2017-07-25

Signature Inference for Functional Property Discovery
└─Signature Inference
  └─Syntactic similarity: Type



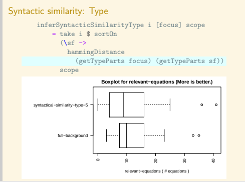
Syntactic similarity: Type

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



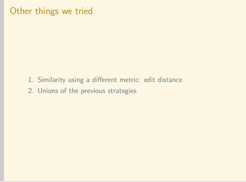
2017-07-25

Signature Inference for Functional Property Discovery
└Signature Inference
└Syntactic similarity: Type

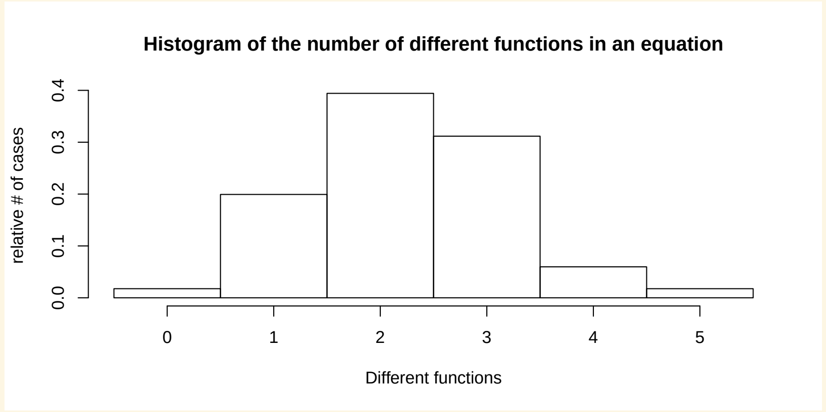


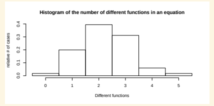
Other things we tried

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

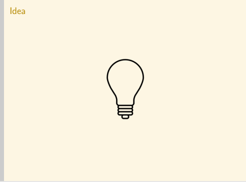


Breakthrough





Idea



Idea

We can run QuickSpec more than once!

We can run QuickSpec more than once!

Inferred Signature

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

2017-07-25

Signature Inference for Functional Property Discovery
└─Signature Inference
    └─Inferred Signature

Inferred Signature

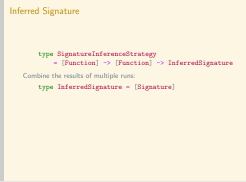
```
type SignatureInferenceStrategy
    = [Function] -> [Function] -> InferredSignature
```

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]
```

Inferred Signature

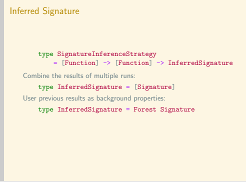
```
    type SignatureInferenceStrategy
        = [Function] -> [Function] -> InferredSignature
```

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
```

Inferred Signature

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

Combine the results of multiple runs:

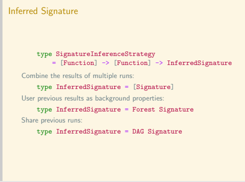
```
type InferredSignature = [Signature]
```

User previous results as background properties:

```
type InferredSignature = Forest Signature
```

Share previous runs:

```
type InferredSignature = DAG Signature
```



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
```

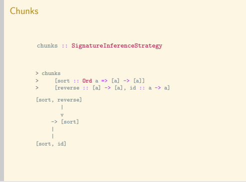
Share previous runs:

```
type InferredSignature = DAG Signature
```

Chunks

```
chunks :: SignatureInferenceStrategy
```

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

[sort, reverse]
        |
        v
     -> [sort]
     |
     |
[sort, id]
```

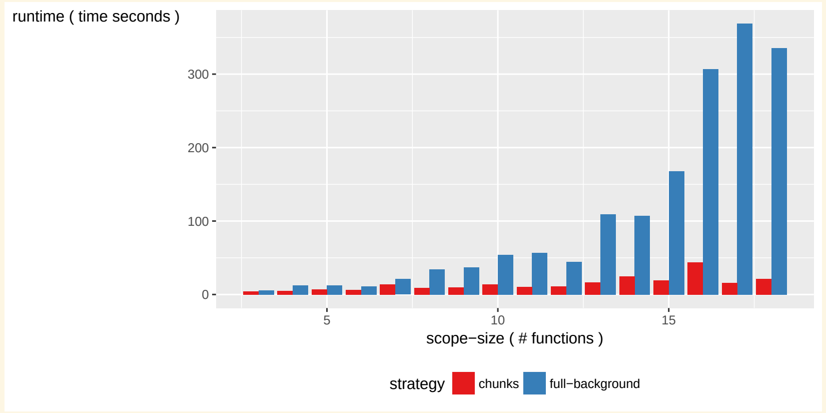

Chunks

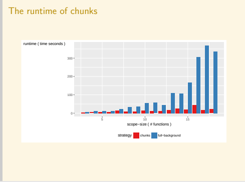
chunks :: SignatureInferenceStrategy

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

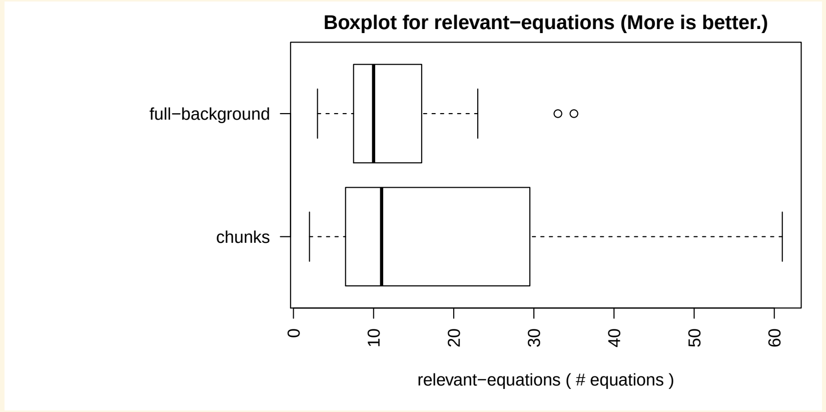
[sort, reverse]
|
v
-> [sort]
|
|
[sort, id]

The runtime of chunks

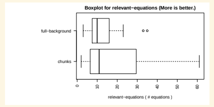




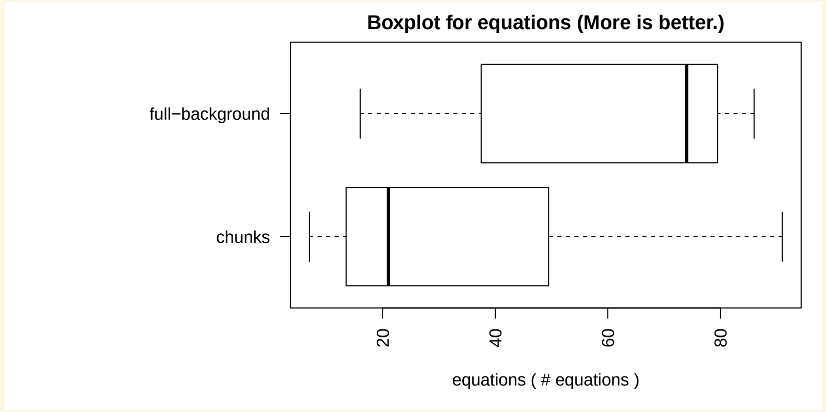
The outcome of chunks: Relevant equations

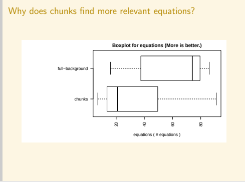


The outcome of chunks: Relevant equations



Why does chunks find more relevant equations?

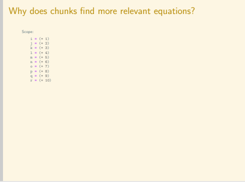




Why does chunks find more relevant equations?

Scope:

```
i = (+ 1)
j = (+ 2)
k = (+ 3)
l = (+ 4)
m = (+ 5)
n = (+ 6)
o = (+ 7)
p = (+ 8)
q = (+ 9)
r = (+ 10)
```


Why does chunks find more relevant equations?

Scope:

```
i = (+ 1)
j = (+ 2)
k = (+ 3)
l = (+ 4)
m = (+ 5)
n = (+ 6)
o = (+ 7)
p = (+ 8)
q = (+ 9)
r = (+ 10)
```

Why does chunks find more relevant equations?

Scope:

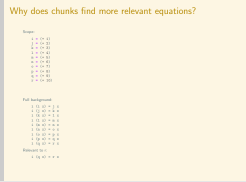
```
i = (+ 1)
j = (+ 2)
k = (+ 3)
l = (+ 4)
m = (+ 5)
n = (+ 6)
o = (+ 7)
p = (+ 8)
q = (+ 9)
r = (+ 10)
```

Full background:

```
i (i x) = j x
i (j x) = k x
i (k x) = l x
i (l x) = m x
i (m x) = n x
i (n x) = o x
i (o x) = p x
i (p x) = q x
i (q x) = r x
```

Relevant to r:

```
i (q x) = r x
```


Why does chunks find more relevant equations?

Scope:

```
i = (+ 1)
j = (+ 2)
k = (+ 3)
l = (+ 4)
m = (+ 5)
n = (+ 6)
o = (+ 7)
p = (+ 8)
q = (+ 9)
r = (+ 10)
```

Full background:

```
i (i x) = j x
i (j x) = k x
i (k x) = l x
i (l x) = m x
i (m x) = n x
i (n x) = o x
i (o x) = p x
i (p x) = q x
i (q x) = r x
```

Relevant to r:

```
i (q x) = r x
```

Why does chunks find more relevant equations?

Scope:

```
i = (+ 1)
j = (+ 2)
k = (+ 3)
l = (+ 4)
m = (+ 5)
n = (+ 6)
o = (+ 7)
p = (+ 8)
q = (+ 9)
r = (+ 10)
```

Full background:

```
i (i x) = j x
i (j x) = k x
i (k x) = l x
i (l x) = m x
i (m x) = n x
i (n x) = o x
i (o x) = p x
i (p x) = q x
i (q x) = r x
```

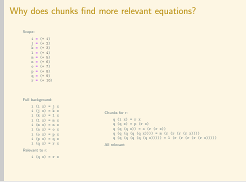
Relevant to r:

```
i (q x) = r x
```

Chunks for r:

```
q (i x) = r x
q (q x) = p (r x)
q (q (q x)) = o (r (r x))
q (q (q (q (q x)))) = m (r (r (r (r x))))
q (q (q (q (q (q x))))) = l (r (r (r (r (r x)))))
```

All relevant


Why does chunks find more relevant equations?
Scope:
i = (+ 1) j = (+ 2) k = (+ 3) l = (+ 4) m = (+ 5) n = (+ 6) o = (+ 7) p = (+ 8) q = (+ 9) r = (+ 10)
Full background:
i (i x) = j x i (j x) = k x i (k x) = l x i (l x) = m x i (m x) = n x i (n x) = o x i (o x) = p x i (p x) = q x i (q x) = r x
Relevant to r:
i (q x) = r x
Chunks for r:
q (i x) = r x q (q x) = p (r x) q (q (q x)) = o (r (r x)) q (q (q (q (q x)))) = m (r (r (r (r x)))) q (q (q (q (q (q x))))) = l (r (r (r (r (r x)))))
All relevant

Inferred Signature

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

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

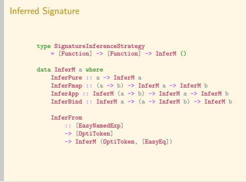
2017-07-25

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
        :: [EasyNamedExp]
        -> [OptiToken]
        -> InferM (OptiToken, [EasyEq])
```



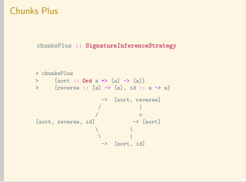
Inferred Signature

```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
        :: [EasyNamedExp]
        -> [OptiToken]
        -> InferM (OptiToken, [EasyEq])
```

Chunks Plus

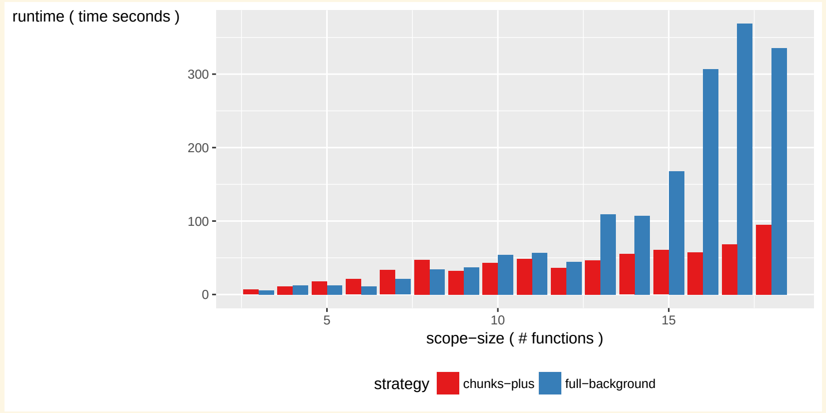
```
chunksPlus :: SignatureInferenceStrategy
```

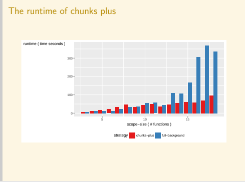
```
> chunksPlus
>     [sort :: Ord a => [a] -> [a]]
>     [reverse :: [a] -> [a], id :: a -> a]

                      ->  [sort, reverse]
                     /             |
                    /              v
[sort, reverse, id]             -> [sort]
                    \           |
                     \          |
                      ->  [sort, id]
```


Chunks Plus

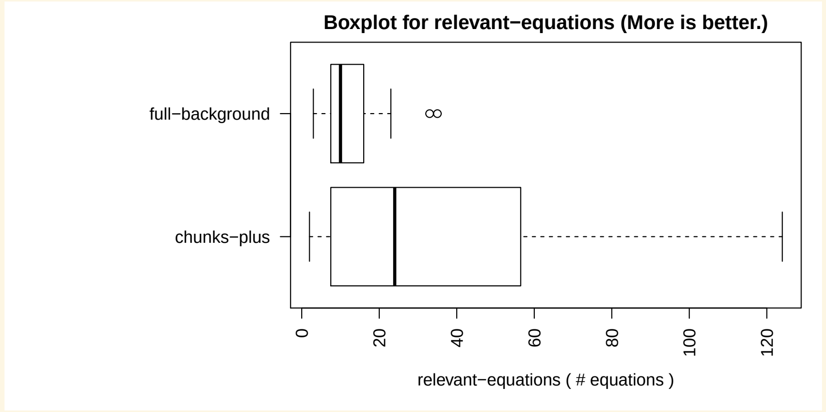
```
chunksPlus :: SignatureInferenceStrategy

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

                      ->  [sort, reverse]
                     /             |
                    /              v
[sort, reverse, id]             -> [sort]
                    \           |
                     \          |
                      ->  [sort, id]
```

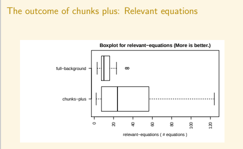
The runtime of chunks plus



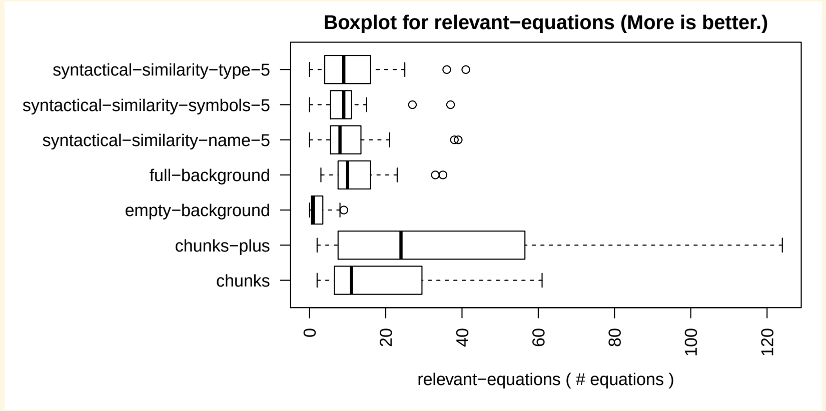


The outcome of chunks plus: Relevant equations



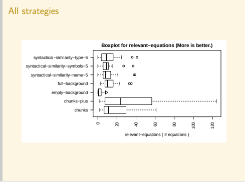


All strategies

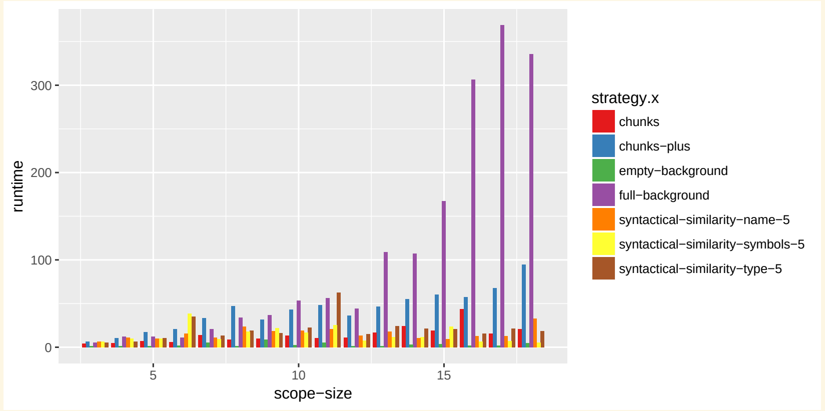


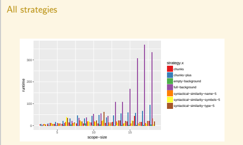
2017-07-25

Signature Inference for Functional Property Discovery
└─Signature Inference
└─All strategies



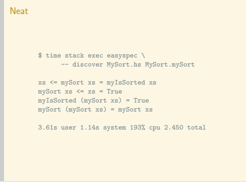
All strategies





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
```


Neat

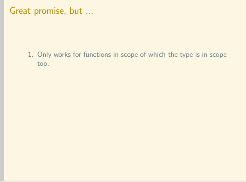
```
$ time stack exec easyspec \
      -- discover MySort.hs MySort.mySort

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 ...

Great promise, but ...

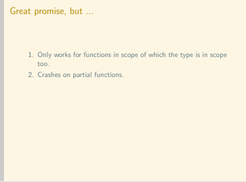
1. Only works for functions in scope of which the type is in scope too.


Great promise, but ...

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

Great promise, but ...

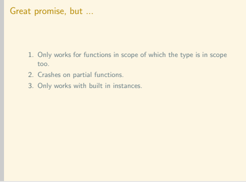
1. Only works for functions in scope of which the type is in scope too.
2. Crashes on partial functions.


Great promise, but ...

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

Great promise, but ...

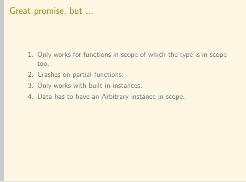
1. Only works for functions in scope of which the type is in scope too.
2. Crashes on partial functions.
3. Only works with built in instances.


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.

Great promise, but ...

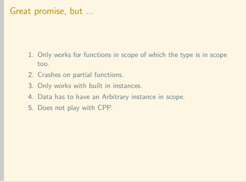
1. Only works for functions in scope of which the type is in scope too.
2. Crashes on partial functions.
3. Only works with built in instances.
4. Data has to have an Arbitrary instance in scope.


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.

Great promise, but ...

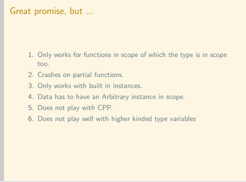
1. Only works for functions in scope of which the type is in scope too.
2. Crashes on partial functions.
3. Only works with built in instances.
4. Data has to have an Arbitrary instance in scope.
5. Does not play with CPP.


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.

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

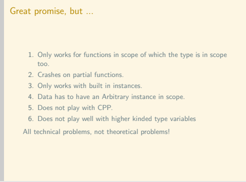

Great promise, but ...

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

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!


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

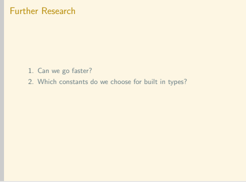
All technical problems, not theoretical problems!

Further Research

1. Can we go faster?

Further Research

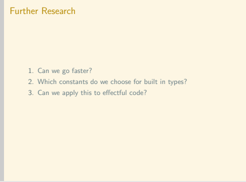
1. Can we go faster?
2. Which constants do we choose for built in types?


Further Research

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

Further Research

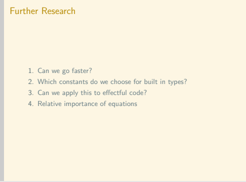
1. Can we go faster?
2. Which constants do we choose for built in types?
3. Can we apply this to effectful code?


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?

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


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!

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

Student at ETH
This is my master thesis
Wrote Haskell in open source
Taught Haskell at ETH
Wrote Haskell in industry
Looking for a job!

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

About Me

Student at ETH
This is my master thesis
Wrote Haskell in open source
Taught Haskell at ETH
Wrote Haskell in industry
Looking for a job!

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