Master thesis

Signature Inference for Functional Property Discovery

Tom Sydney Kerckhove

Adviser	Dr Dmitriy Traytel
Supervisor	Prof. David Basin
Department	Information Security
Date	2017-09-09

Abstract

Property discovery is the process of discovering properties of code via automated testing. It has the potential to be a great tool for a practical approach to software correctness. Unfortunately current methods remain infeasible because of the immense search space. We contribute a new approach to taming the complexity of a state-of-the-art generate and test algorithm. Our approach runs this algorithm several times with carefully chosen inputs of smaller size: signature inference. We implement this approach in a new tool called EasySpec. The results suggest that this approach is several orders of magnitude faster, fast enough to be practical, and produces similar results.

Acknowledgements

I would first like to acknowledge Professor David Basin. He allowed me to work on a topic of my choosing and believed in me when I did.
I would also like to thank my thesis adviser Dr Dmitriy Traytel. His door was always open whenever I ran into a trouble spot or had a question about my research or writing. He consistently allowed this work to be my own, but steered me in the right the direction whenever he thought I needed it.
A big thank you to everyone who contributed to proofreading this work. Their extra eyes were most helpful.
Finally, I must express my very profound gratitude to my parents and to my brother for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them.
As such, I would like to dedicate this work to my father, and introduce this work with his summary of what it means to present a thesis:

> Ik heb ergens op gestudeerd en nu ga ik mijn vertellingske doen.

Thank you.
Tom Sydney Kerckhove

Contents

List of Figures

1 Introduction

Writing correct code is hard. In order to write correct code, a programmer first has to decide what it means for code to be correct, and then write code that matches these expectations. The process is further complicated by the fact that these two phases are rarely separated in practice.
The first step in writing correct code is to choose a programming language. Compilation, static typing, immutability, functional programming, purity, higher-order programming and laziness are just some examples of programming language features that should support the programmer in writing correct code.
Beyond what the language of choice can offer, tools that employ formal methods such as formal verification, model checking, various static analysis techniques, etc, can provide additional correctness guarantees. These usually suffer from one of two problems: Either the programmer already has to have decided what it means for their code to be correct, or the method is so specific in its scope that it cannot help in general. Moreover, they are also often too expensive in terms of engineering effort or computing power, or both, to be used in practice.
The most commonly adopted approach in practice, is testing. Testing suffers from the same issue that a programmer already has to know what it means for their code to be correct, but it is rather more generally applicable. The idea of automated testing is that programmers can tell if their code is faulty before running the code in production. In principle, the programmer writes extra code that will automatically test whether the subject code is faulty. It seems that testing currently has the best balance between cost and benefit, but it is still too often beneficial from a business perspective to skip writing tests in the short term.
Unit testing is the most widely adopted approach to testing. A unit test consists of a piece of code, without arguments, and possibly some assertions about the results. We say that a unit test passes if, when run, the code does not crash and no assertions fail. Unit tests have two main problems.
Edsger W. Dijkstra famously said "Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence. " [3]. Indeed, unit tests often fail to address all possible aspects of the code under test. One would have to write at least one unit test for every possible code path of the code under test to have a chance to cover all possible aspects of the code under test. In what follows, we will call this problem the *coverage problem*.
Unit testing is quite expensive when it comes to developer time. Developers have to come up with enough examples of inputs and their corresponding outputs to cover the subject code. Then they have to implement the corresponding unit testing code. Not only does this process take a lot of time, it is also a mentally taxing task since the process is repetitive and often unrewarding. The problem of cost is exacerbated by the sad reality that management often does not see the value in testing. After all, if the code works, then testing looks like an extra cost that could be cut. If the code contains mistakes, then spending time testing (instead of ignoring the mistakes) just makes the development process seem slower. As a result, tests are all too often omitted from a software project.
Property testing [12] [22] differs from unit testing in one key aspect: The test code now takes an argument as input. For any argument, the test code now

tests a property of this argument, namely the property that the test passes when given this input. We say that the property test passes if it passes for a given number of randomly generated input values.
Property testing solves the coverage problem of unit testing probabilistically, but exacerbates the cost problem of unit testing. As the number of executions of a property test increases, the probability that all possible aspects of the code under test are covered, should tend to one. This means that a programmer now only needs to write a limited number of property tests, instead of a large number of unit tests. However, coming up with general properties of code is often considered more difficult than coming up with example test cases. Consequently, the developer time costs of property testing is even higher and, as a result, property testing is rarely ever done in practice.
Property discovery [14], [16], [21] is a technique to produce property tests for subject code automatically. By eliminating the human effort from conceiving tests, this approach can be combined with property testing to solve both of the problems with unit testing. By discovering properties automatically, the programmer is relieved from having to think of examples or properties, and now only has to select the properties that they think should hold. Property discovery is a relatively new technique, that is not ready for use in practical software engineering yet.
Claessen et al. [14] have explored automatic discovery of equational properties, and have shown that property discovery is a complex problem. The first attempt failed to discover large properties or properties of a large codebase in a reasonable amount of time. Subsequent research [16] has improved upon these limitations, but remains unable to discover properties of realistic codebases.
The input to the property discovery algorithm developed [14] and [16] is called a *signature* of functions. A signature is a set of functions that are defined to be relevant in property discovery. All functions that are not in the signature are completely ignored. This means that, to discover the properties of a single function, the programmer has to specify which functions are relevant. The process of figuring out which functions are relevant in property discovery is often only marginally easier than to think of the properties manually.
This signature is defined by a piece of code that the programmer has to write manually. It contains the names, types, and references to the implementations, of all the functions in the signature upon which one wishes to run the property algorithm. Writing this code represents a significant burden on the programmer that ensures that property discovery is not a practical method in practice from a developer's perspective.
We contribute a new approach to taming the computational complexity of property discovery: signature inference. The premise is that a programmer should not have to specify all relevant functions in order to perform property discovery. Ideally, the programmer should also not have to write an extra piece of code just to run the property discovery algorithm. Signature inference consists of inferring appropriate input for the current property discovery mechanism by using compile time information about the subject code. This approach therein simultaneously solves the "code as input" problem. Altogether, signature inference has the potential to solve the problems with property discovery and, by extension, property testing and unit testing. We propose a domain specific language to define what we have named *signature inference strategies*. These describe how to automatically supply input to a property discovery algorithm and

combine the results. Lastly, we contribute several signature inference strategies and a framework in which these signature inference strategies can be evaluated and compared to each other.

2 Background

In this section I will introduce the setting of our work.

2.1 Haskell

The code in this work is written in the Haskell programming language. Some familiarity is assumed, but the important details will be revisited here.
Haskell is a statically typed, purely functional language that evaluates lazily by default. The language lends itself well to concurrent programming and it uses type inference to ease programming [6].

2.1.1 Type Classes

Haskell famously has support for type classes [18], [19], [2]. Type classes are a way to support ad hoc polymorphism, as opposed to the parametric polymorphism that type parameters provide. Using type classes, the programmer can add constraints to type parameters. As an example, consider equality. Without type classes, the programmer would have to write a different equality function for each type. `eqInt :: Int -> Int -> Bool` for integers, `eqFloat :: Float -> Float -> Bool` for floating point numbers, etc. The `Eq` type class solves this problem by allowing the programmer to define what equality means for a type.

```
class Eq a where
    (==) :: a -> a -> Bool
```

For specific instances, equality is defined by instantiating the `Eq` type class using a type class instance. In this instance, the programmer implements the evidence that the given type indeed belongs to the `Eq` type class.

```
instance Eq Int where
    (==) :: Int -> Int -> Bool
    i == j = eqInt i j
```

Once equality is defined, the programmer can use the `==` function to test for equality, and the type checker will infer which equality will be used. Now that equality can be defined parametrically in which type of values is tested, the programmer can write functions that are parametric in a type, so long as that type supports equality. For example, the `elem` function can test if a list contains a given element, as long as the elements of the list support equality.

```
elem :: Eq a => a -> [a] -> Bool
```

When a type supports equality, we say that this type is in the type class `Eq`.

2.1.2 Type Class Evidence Dictionaries

The Glasgow Haskell Compiler implements type classes using a desugaring method called evidence dictionaries [9]. Take the previous example of a function with a type parameter that has a type class constraint.

```
elem :: Eq a => a -> [a] -> Bool
```

The compiler can compile this function without knowing which type will eventually inhabit the parameter `a`. However, in order to do that, the compiler needs to know which concrete function to use in place of the `==` function in the `Eq` type class. The type of the `elem` function is desugared to the following.

```
elem :: Dict (Eq a) -> a -> [a] -> Bool
```

In the above, the new `Dict Eq` argument is a dictionary that contains the concrete implementations of the functions in the `Eq` type class. This dictionary is called an evidence dictionary, because it contains the evidence that the corresponding type class inhabits the given type.

2.2 Property Testing with QuickCheck

QuickCheck [12], [11] is an implementation of the concept of property testing and it is written in Haskell. It is a particularly elegant example of a use case for type classes. Instead of requiring a programmer to specify a specific example of the workings of their code, the programmer now gets to specify a general property of their code. To perform property tests, a programmer needs to provide two pieces of code: a generator, and a function that takes the output of the generator and produces a Boolean value.

2.2.1 Generators and the Arbitrary Type Class

Generators are a central component of QuickCheck and property testing in general. In essence, a generator is a pure function that can use a pseudorandom generator to produce values of a certain type. For example, a value of type `Gen Int` provides evidence that it is possible to generate pseudorandom integers given a pseudorandom generator of bits. Users can write generators for their own data types such that values of those types can also be generated. Elements of the `Arbitrary` type class provide a generator called `arbitrary` that can generate random values of that type.

```
class Arbitrary a where
    arbitrary :: Gen a
```

QuickCheck will use the `Arbitrary` type class by default, but users can also specify which generator to use in the random testing.

2.2.2 Properties

A property, in this case, is loosely defined as anything that can produce a Boolean value from supplied pseudorandomness. An example is a function as follows.

```
f :: A -> Bool
```

This function is also usually called the property, even though it is pure. To produce random Boolean values using this function, QuickCheck will require a generator of the input type `A`.

2.2.3 Running Property Tests

When instructed to do so using the `quickCheck` function, the QuickCheck testing framework will generate random inputs to the supplied property. For each of the inputs, QuickCheck asserts that the resulting Boolean value is `True`. If all of the output values are `True` then we say that the property passes the property test. If the property evaluates any of the input values to `False` then the test fails. When a function fails a property test, a shrinking process is started. In this process, the input value for which the property does not hold is inspected further in order to find a smaller value that still fails the property. The shrinking process is not relevant to the work described in this thesis. For further details, please refer to the original QuickCheck paper [12] and the QuickCheck package on Hackage [11].

2.3 Property Discovery with QuickSpec

QuickSpec is a recently developed tool for property discovery [14], [16]. QuickSpec aims to simplify the process of writing property tests. It is based on the idea that properties could be discovered using a combination of intelligently looking at types, and using tests to validate the properties using QuickCheck. QuickSpec exposes the following function as its entry point.

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

To run QuickSpec, a programmer is required to write a piece of code that calls this function in order to discover any properties. To call the `quickSpec` function, the programmer has to define a value of the type `Signature`.
The following is a simplified definition of `Signature`.

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

A signature mainly consists of a list of functions (`constants`) that are called constants because they must be in the `Typeable` type class. Specifically, a type with a `Typeable` instance allows a concrete representation of a type to be calculated [24]. A `Typeable` instance can automatically be generated with the `DeriveDataTypeable` GHC language extension or written manually, but only if the type in question is monomorphic. For example, the type `Int -> Bool` is in `Typeable` but `a -> Double` is not, because the latter is polymorphic in a type variable: `a`. It is important to note that generating values of a type with type parameters is impractical. A value of type `Constant` also contains a

given name for the function that it represents. This allows QuickSpec to output the discovered properties in a human readable manner.
A signature also contains a set of type class instances (`instances`) specifically the evidence dictionaries or information about how to construct them. For example, if a signature is said to contain the instance `Eq Int` then it contains a record with the following two functions.

```
(==) :: Int -> Int -> Int
(/=) :: Int -> Int -> Int
```

QuickSpec only knows about those instances that it gets supplied with via the signature.
The last part of a signature that is important for this work is a set of background properties (`background`). These are properties that we can tell the signature about if we already know them, and will use in its discovery. One way for a signature to contain background properties, is by using the properties that a previous run of QuickSpec discovered.
A signature also contains several configuration settings such as the maximum size of the discovered properties. The default value for this setting sets the maximum size of discovered properties to be 7 and we do not change it throughout this work. The discovery algorithm will discover all properties smaller than or equal to this maximum size. The size of a property is the maximum of the sizes of each of the sides of the equation.
Using a signature, QuickSpec will enumerate all properties that relate the functions in that signature, up to a given size. A property, in QuickSpec, is defined as an equation of the following form.

```
leftTerm = rightTerm
```

Here, `leftTerm` and `rightTerm` must be of the same type, and that type must be of the following form.

```
A -> B
```

In this type, `A` must be in `Arbitrary` such that arbitrary values can be generated as input. Note that it is not a restriction to only allow properties where the input types of both sides are the same. Indeed, for any property where the input types of the sides are different, there exists a property where this is not the case and where it expresses the same equation.

```
\a      -> f a = \b      -> g b -- different input type
\(a, b) -> f a = \(a, b) -> g b -- same input type
```

Furthermore, `B` must be in `Eq` such that the results of the respective functions may be compared for equality. Note that the shape of these properties is not a limitation with respect to their expressiveness. Indeed, any general property `p :: A -> Bool` can be expressed in the above form as follows.

```
\x -> p x = \_ -> True
```

Also note that QuickSpec does not show properties in this form. In fact, QuickSpec leaves out the left side of the lambda expression, so it would show the above equation as follows.

```
p x =       True
```

When QuickSpec is run using the `quickSpec` function, the discovered properties are contained in the `background` field of the resulting `Signature`. In fact, this resulting `Signature` is equal to the input `Signature` in all other respects. For further details, please refer to the QuickSpec papers [14], [16] and the QuickSpec repository on GitHub [17].

3 Signature Inference with EasySpec

In this section we will introduce the concept of signature inference and how we implemented it in the EasySpec tool.

3.1 Premise

QuickSpec requires a programmer to specify all of the context that they are interested in. If one were to automate the process of generating QuickSpec code from a codebase, the result would discover all properties that relate any properties in that codebase. This work, however, asserts that usually a programmer is not necessarily interested in all the equations relating the functions in the entire codebase. The assumption is that a programmer is more interested in the properties that involve a very small number of functions, say one. We call these functions the *focus functions*. The new goal is to find the properties that relate the focus functions to the rest of the codebase. We will call these *relevant equations* or *relevant properties*. With respect to this new goal, QuickSpec has at least two problems.

- QuickSpec will most likely find (a lot) of properties that do not relate the focus functions with the rest of the code base at all.
- QuickSpec will take an immense amount of time to run, with respect to the size of the signature it is given.

The question that this work tries to answer is "How do we choose the inputs to give to QuickSpec intelligently, such that it will find properties of that one focus function in a practically feasible amount of time?". Ideally the inputs will be chosen such that:

- QuickSpec finds all of the properties that relate the focus function with the rest of the codebase.
- QuickSpec does not waste any time finding the properties of the codebase that do not involve the focus function.

3.2 Automation

The first step in making property discovery feasible for practical use, is ensuring that a programmer never has to write extra code to discover properties of their code. This automation involves inspecting the subject code and generating a `Signature` to run `quickSpec` on using an automated interactive evaluator.

3.2.1 Generating a Signature

By interfacing with the GHC API [5] one can find all functions that are in scope in a given module. We will refer to these functions as values of type `Function` for easy reference, because these values have several names in the GHC API. To automatically supply QuickSpec with a `Signature` we need to generate a Haskell expression that describes a `Signature` and interactively evaluate it. Recall that a signature contains a list of `Constant` values. To generate an expression that evaluates to a `Signature`, we need to generate subexpressions

that each evaluate to a `Constant`. This involves monomorphising the functions in scope, and generating a `Constant` expression for each of the `Function` values by giving each function a name, and specifying their type explicitly.

```
not :: Bool -> Bool
   -- becomes --
constant "not" (not :: Bool -> Bool)
```

Next, we need to generate an expression that describes the type class instances that are in scope. QuickSpec already knows about some instances by default, and these were comprehensive enough to perform our research, so the full instance resolution work has not been implemented in EasySpec.
Lastly, we need to generate an expression that describes the properties that we already know about. Previously discovered properties usually come from previous runs of QuickSpec, so we leave this field empty for now and get back to it in section 3.6.1. We will call this initial automation `full-background` for reasons that will become clear in section 3.3.

3.2.2 Monomorphisation

Monomorphisation consists of instantiating any type variables in the type of a function with monomorphic types so that no type variables remain.
For type parameters of kind `*` in types without any constraints on the type parameters, QuickSpec has support in the form of placeholders. QuickSpec exposes five types `A`, `B`, `C`, `D`, `E` that represent type parameters. In reality these placeholders are just `newtype`s over `Integer` [1]. This means that monomorphising such a type is as simple as turning the type parameters into the QuickSpec placeholders. The following translation is an example of such a monomorphisation.

```
map :: (a -> b) -> [a] -> [b]
       -- becomes --
map :: (A -> B) -> [A] -> [B]
```

Monomorphisation is a bit more complicated in types that have type class constraints. Consider the following type.

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

Given type class constraints, the previous translation would be unsound. Consider the translation as follows.

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

There is in fact no guarantee that `A` is in fact in the `Ord` type class even though the parameter that it represents should be in the `Ord` type class. The way QuickSpec solves this problem is by translating type class constraints to argument evidence dictionaries. The previous type would be translated to the following.

[1] Note that this is sound because values of a parametric type cannot be inspected at all.

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

Here `Dict (Ord A)` is the evidence dictionary for the `Ord A` constraint, and `mkDict` the function that can turn a type class constraint into an evidence dictionary argument.
Haskell supports type parameters that cannot be instantiated with a type, but rather with a type constructor. These type parameters are called higher kinded. An example of a higher kinded type parameter is `t` in the length function.

```
length :: Foldable t => t a -> Int
```

Monomorphisation of higher kinded types is not trivial in general, but could be performed manually for type constructors in scope, given that type constraints can be resolved. This transformation has not been implemented in EasySpec yet.

3.2.3 Complexity

The `full-background` automation already offers many benefits, but it does not solve the problem of computational complexity that QuickSpec exhibits.

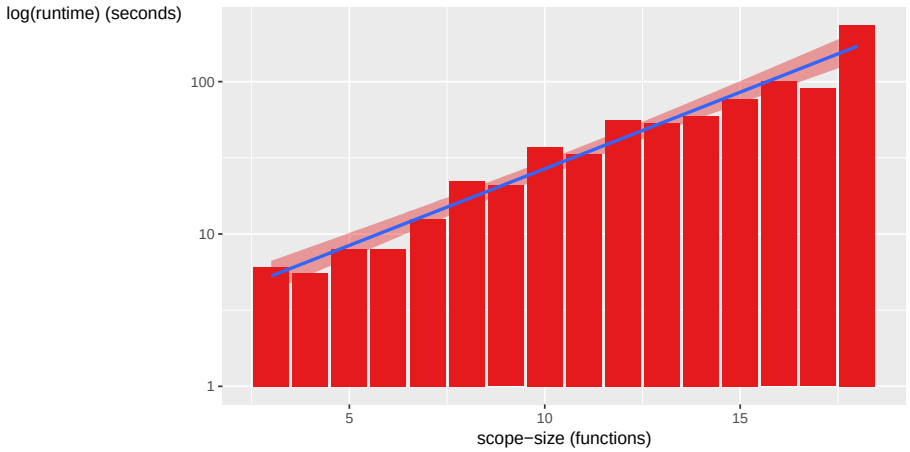


Figure 1: Runtime of `full-background`

In Figure 1 we see the runtime of the automated version of QuickSpec with respect to the size of the scope that it was run on. The maximum size of the discovered properties was set to the same default as QuickSpec. In fact, the complexity is of the order $O\left(N^{2P}\right)$ where N is the size of the scope, and P is the maximum size of the discovered properties. This is still polynomial time, since we fix P to be 7, but it is not practical for real codebases. For a more detailed breakdown of this complexity, we refer to section 4.1.

3.3 Reducing Signature Inference Strategies

The first attempt at speeding up the above process is to shrink the signature that is given to QuickSpec by omitting functions. This means that we

looked for ways to select a smaller subset of the scope. The first definition of a `SignatureInferenceStrategy` looks as follows.

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

The first argument to a signature inference strategy is the list of focus functions, and the second argument is a list of all the functions in scope. The result is supposed to be a list of elements of the scope that is smaller than the entire scope. This kind of signature inference strategy is sometimes called a reducing signature inference strategy.
Running QuickSpec on the entire scope of functions can be described as a trivial reducing signature inference strategy as follows.

```
fullBackground :: SignatureInferenceStrategy
fullBackground focus scope = scope
```

The simplest reducing signature inference strategy is called `empty-background`. It consists of reducing the scope to the focus and its entire implementation can be written as follows.

```
emptyBackground :: SignatureInferenceStrategy
emptyBackground focus scope = focus
```

The `empty-background` signature inference strategy will only run QuickSpec on the focus functions. As a result, it will find only relevant equations. However, it will only find equations that exclusively relate the focus functions to each other. Examples of such equations are idempotency and involution.

```
reverse (reverse x) = x -- Involution
sort (sort x) = sort x  -- Idempotency
```

3.4 Distance-based Reducing Signature Inference Strategies

Given a general distance between functions, we can construct a reducing signature inference strategy. The idea is that we may be able to use a distance to predict whether functions will be relevant to each other. If this is true, then the functions that are closest to the focus functions should be chosen to run QuickSpec on. A general distance based signature inference strategy has one parameter, namely the number of functions to choose around the focus functions. We will call this parameter i. For each function in scope, the distance to the focus function is calculated and the i closed functions in scope are put together in the signature. The following piece of code implements this concept [2].

[2]This piece of code assumes that there is only a single focus function. In practice this is a valid assumption, but this code could also be extended to work on a larger focus using a summation.

```
distanceBased
    :: (Function -> Function -> Double)
    -> Int -> SignatureInferenceStrategy
distanceBased dist i [focus] scope
    = take i
    $ sortOn
        (\sf -> dist focus sf)
        scope
```

Figure 2: General distance based signature inference strategy

3.4.1 Syntactic Similarity Name

The first distance based reducing signature inference strategy is called `syntactic-similarity-name`. It is based on the assumption that mutual relevance of functions can be predicted by the similarity of their names. For example, the functions `isPrime :: Int -> Bool` and `primeAtIndex :: Int -> Int` are most likely relevant to each other, and we may be able to predict this fact because the names both mention the word "prime". Because EasySpec has access to compile time information about code, it can introspect the name of functions. The following is pseudocode to define this signature inference strategy.

```
inferSyntacticSimilarityName
    :: Int -> SignatureInferenceStrategy
inferSyntacticSimilarityName
    = distanceBased
        (\ff sf ->
            hammingDistance (name ff) (name sf))
```

Figure 3: `syntactic-similarity-name`

3.4.2 Syntactic Similarity Symbols

A second distance based reducing signature inference strategy is called `syntactic-similarity-symbols`. This strategy looks at the implementation of functions to determine a distance. It is based on the assumption that mutually relevant functions will have a similar looking implementation when it comes to the functions that they use. To use the same example, `isPrime :: Int -> Bool` and `primeAtIndex :: Int -> Int` both probably use `div :: Int -> Int -> Int` and `mod :: Int -> Int -> Int` and should therefore be judged to be close to each other. `syntactic-similarity-symbols` defines the distance between two functions as the hamming distance between the symbol vectors of these functions.

```
inferSyntacticSimilaritySymbols
    :: Int -> SignatureInferenceStrategy
inferSyntacticSimilaritySymbols
    = distanceBased
        (\ff sf ->
            hammingDistance (symbols ff) (symbols sf))
```

Figure 4: `syntactic-similarity-symbols`

3.4.3 Syntactic Similarity Type

Our final distance based reducing signature inference strategy is called `syntactic-similarity-type`. This strategy is based on the assumption that functions that are relevant to each other will have similar types. In the example of `isPrime :: Int -> Bool` and `primeAtIndex :: Int -> Int` both of the types of these functions mention the type `Int`. For each type, we define a multiset of parts of that type. For example, the type `[a] -> [a]` has the following parts multiset.

```
[a] -> [a] : 1
([a] ->)   : 1
(-> [a])   : 1
(->)       : 2
[a]        : 2
[]         : 2
a          : 2
```

Figure 5: The multiset of parts of the type `[a] -> [a]`

This multiset is interpreted as a vector in an infinitely dimensional vector space. In this space, the hamming distance between two vectors is used as the distance between two functions. The resulting signature inference strategy looks as follows.

```
inferSyntacticSimilarityType
    :: Int -> SignatureInferenceStrategy
inferSyntacticSimilarityType
    = distanceBased
        (\ff sf ->
            hammingDistance (typeParts ff) (typeParts sf))
```

Figure 6: `syntactic-similarity-type`

3.5 Type Reachability

A more sophisticated signature inference strategy is based on how functions can be composed. This signature inference strategy is called `type-reachability` and it is a reducing signature inference strategy. It uses type information to try to rule out parts of the scope from being relevant to the focus.

3.5.1 Motivation

The idea is to determine which functions could possibly be composed with the focus functions to produce an equation. Consider the following example of a scope.

```
f :: Int -> Char
g :: () -> Int
h :: Char -> Double
i :: Bool -> Bool
j :: Double -> Double
```

Figure 7: An example scope in which `type-reachability` would work well

If the focus consists of the single function `f` then, without looking at their implementation, we can already say that `i` can never occur in an equation together with `f`. This is because there is no way to make terms on the left and right hand side of the equation that both mention `f` and have a sub term of type `Bool`. The intention of the `type-reachability` signature inference strategy is to eliminate functions from the scope that cannot occur together in an equation with any focus functions. To see why this situation occurs, we have to explore a concept that we have called type reachability.

3.5.2 Definition

Type reachability is defined recursively as follows.

1. Every function is type reachable from itself in zero steps.
2. If two functions can make up an equation by each occurring on either side of the equation, then they are type reachable from each other in one step. This means that the types of these two functions must be unifiable. For example, the functions `id :: a -> a` and `reverse :: [a] -> [a]` are type reachable from each other because the equation `id xs = reverse xs` typechecks. Note that this equation does not have to hold.
3. If the output of one function can be used as an input to a second function, then these functions are type reachable from each other in one step. For example, the functions `(+) :: Num a => a -> a -> a` and `take :: Int -> [a] -> [a]` are type reachable from each other because they can be composed as `take (a + b) xs`. Note that there can

be multiple possible input and output types of a function. The function `(+)` can have both `a` and `a -> a` as an output type due to currying. The function `take` can have both `Int` and `a` as an input type due to the fact that it has multiple arguments.

4. If a function `f` is type reachable from a function `g` in a steps, and a third function `h` is reachable from `g` in b steps, then we say that `f` is type reachable from `h` in $a + b$ steps.

Consider the example in Figure 3.5.1 again. From the function `f`, the functions `g` and `h` are type reachable in one step. The function `j` is type reachable in two steps, and `i` is not type reachable from `f` at all.

3.5.3 The Type Reachability Strategy

The idea of `type-reachability` is to find all functions in scope that are type reachable from the focus functions in less than i steps, where i is a parameter of the strategy. It is important to note that we only use an underapproximation for real type reachability. This underapproximation only deals with monomorphic types, and only considers the first argument and the last output of functions. This limitation allows us to implement the underapproximation function `typeReachableInOneStep` without integrating with any type checker.

```
inferTypeReachability :: Int -> SignatureInferenceStrategy
inferTypeReachability i focus scope = go i focus
  where
    go 0 acc = acc
    go n acc = go (n - 1) $
      concatMap
        (\af -> filter (typeReachableInOneStep af) focus)
        acc
```

Figure 8: `type-reachability`

In this work, we investigate this strategy with 7 as the parameter i, because this is the maximum size of properties that we use.

3.6 Graph Signature Inference Strategies

During experimentation, we searched for clues to build better strategies. One of the measures we looked at, was the number of different functions that occur in any given property. We ran `full-background` on many examples, and gathered the following data.

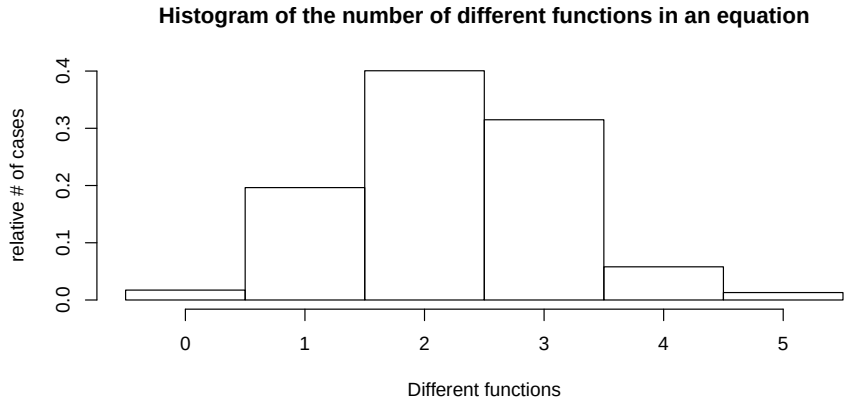


Figure 9: Number of different functions that occur in a property

In Figure 9 there is a histogram of the number of different functions in an equation. We notice that 60 and 70 per cent of all properties that were discovered by `full-background` only talk about two or fewer functions. Similarly, we find that almost 90 per cent of all properties only talk about three or fewer functions. This means that it may not always be worthwhile to run QuickSpec on large signatures. The runtime of running QuickSpec suggests that it is cheaper to run QuickSpec multiple times with different small signatures than to run QuickSpec on a large signature.

3.6.1 Definition

To allow for multiple runs of QuickSpec, the definition of a signature inference strategy needs to be changed. The first attempt involved defining a signature inference strategy as a reducing signature inference strategy that produces a list of signatures instead of a single signature.

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

type InferredSignature = [Signature]
```

A list signature inference strategy, as these were named, could be run very similarly to a regular reducing signature inference strategy. QuickSpec is run on each of the different signatures that is inferred, and the resulting equations are collected as the result. Note that any reducing signature inference strategy can be trivially converted to a reducing signature inference strategy.
In the next iteration of this idea, we observed that QuickSpec has a feature that allows it to learn from previous discoveries as alluded to in section 2.3. Recall that a signature contains a field called `background` that allows a user to specify previously discovered properties. The result of running QuickSpec is a signature in which this field has been filled with the properties that were discovered in that run. In subsequent runs, the user can then specify these discovered properties in the signatures to run QuickSpec on, and QuickSpec

will take these properties into account for optimisation of further discovery. For more details, refer to the second QuickSpec paper [16].
We adapted our definition to allow for dependencies between signatures by arranging the resulting signatures in a forest.

```
type InferredSignature = Forest Signature
```

Now we could describe the idea that every signature had to be run before its parent could be run, and QuickSpec would perform the appropriate optimisation. Again, it is trivial to convert a list signature inference strategy into a forest signature inference strategy. Next, we noticed that we may as well allow signature inference strategies to share children in the forest. That's why we adapted our definition again, this time to arrange signatures in a directed acyclic graph.

```
type InferredSignature = DAG Signature
```

Note that any tree is a directed acyclic graph as well, so translations were trivial again. At this point, signature inference strategies were strictly more expressive, which allowed for more intricate signature inference strategies.

3.6.2 Chunks

Using the newfound knowledge that properties usually contain very few different functions, we set out to create a signature inference strategy that takes advantage of this fact. It sets out to find the properties that contain two or fewer different functions. As such, it first creates a node that only contains the focus functions. Next, it creates a signature for every tuple of one focus function and one scope function and adds the initial node as a dependency. The resulting directed acyclic graph of signatures is star shaped and only contains signatures with two or fewer functions. As an example, consider the scope in figure 10, and choose `sort` as the focus.

```
sort :: Ord a => [a] -> [a]
reverse :: [a] -> [a]
id :: a -> a
not :: Bool -> Bool
```

Figure 10: An example scope

The `chunks` signature inference strategy will first run QuickSpec on a signature that only contains `sort`, and then on the signatures with two functions as depicted in figure 13.

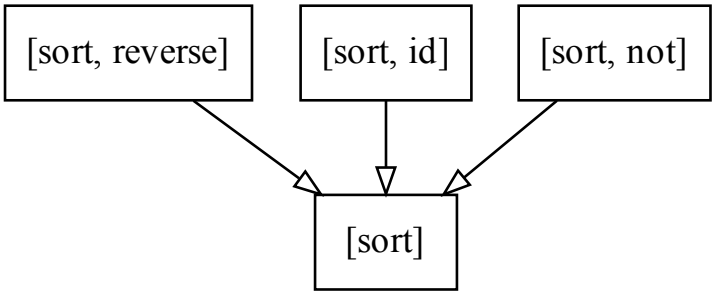


Figure 11: The graph that `chunks` builds for the example

3.7 Monadic Signature Inference Strategies

Graph signature inference strategies are expressive enough to declare dependencies between QuickSpec runs, but they cannot use information from previous runs in subsequent runs. One hypothesis suggests that the equations that are discovered in QuickSpec runs may teach more about the functions at hand, and what we can expect when we subsequently run QuickSpec. The definition of a signature inference strategy would have to be adapted again, to make it even more expressive.

3.7.1 Definition

One standard abstraction that allows for the expression of composition is the monad. We adapted the result of a signature inference strategy to be defined as a monadic piece of data that expresses how and when QuickSpec should be run.

```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])
```

To allow for monadic computation, but not for arbitrary input or output, the `InferM` monad represents a syntax tree that describes a computation. The following instances then enable syntactic sugar to construct such a syntax tree.

```haskell
instance Functor InferM where
    fmap = InferFmap

instance Applicative InferM where
    pure = InferPure
    (<*>) = InferApp

instance Monad InferM where
    (>>=) = InferBind
```

The special constructor `InferFrom` describes the intent to run QuickSpec on the given signature. For each run of QuickSpec, an `OptiToken` is generated that describes that run of QuickSpec, and can be given to subsequent runs to inform QuickSpec about the corresponding previous discoveries. As an example, `chunks` can now be described as follows.

```haskell
inferChunks :: [Function] -> [Function] -> InferM ()
inferChunks focus scope = do
    (l1t, _) <- InferFrom focus []
    forM_ [(f, s) | f <- focus, s <- scope] $ \(f, s) ->
        InferFrom [f, s] [l1t]
```

Figure 12: `chunks` as a monadic signature inference strategy

This monad only expresses the intent to run QuickSpec. One still requires a straightforward interpreter to actually discover any properties.

3.7.2 Chunks Plus

The increased expressiveness of monadic signature inference strategies opened the doors for a new strategy: `chunks-plus`. This strategy is built upon the assumption that if two functions are related by any properties that only relate these two properties, then they are more likely to also be related in properties with more different functions. The idea is that `chunks-plus` first runs QuickSpec in a similar way to `chunks` and then potentially runs QuickSpec some more based on the following procedure. For each set of two different scope functions, the corresponding nodes in the `chunks` strategy are considered. If QuickSpec finds relevant equations in both of these nodes, a new node is created that contains both of these scope functions and the focus function. This new node then points to appropriate two nodes as a dependant. As an example, consider the same example scope as in section 3.6.2. The `chunks-plus` signature inference strategy will first run QuickSpec exactly as `chunks` would, and then conditionally on the signatures with three functions as depicted in grey in figure 13.

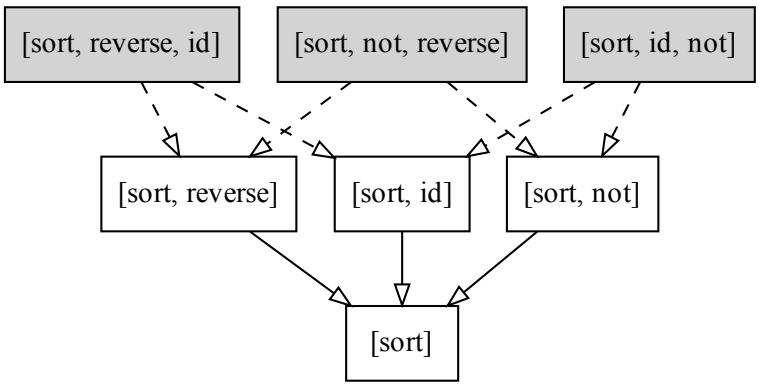


Figure 13: The graph that `chunks-plus` builds for the example

3.8 Composing strategies: Drilling and Reducing

In the previous sections, a larger pattern emerges. The idea that the entire codebase is too large to consider even a substantial part of it, leads us to signature inference strategies that try to reduce the scope to a smaller subset of relevant functions. On the other hand, we find signature inference strategies that find more relevant equations (but fewer irrelevant equations) and spend less time as a result. In this section we call this concept "drilling". Now we can combine these two ideas and make new signature inference strategies by composing previous signature inference strategies. A reducing corresponds to a

function as follows.

```
[Function] -> [Function] -> [Function]
```

We have already considered the following reducings.

- `empty-background`
- `syntactic-similarity-name`
- `syntactic-similarity-type`
- `syntactic-similarity-symbols`
- `type-reachability`

A drilling corresponds to a function as follows.

```
[Function] -> [Function] -> InferM ()
```

The idea here is to find as many relevant equations as possible for the given focus. We have already considered the following drillings.

- `full-background`
- `chunks`
- `chunks-plus`

3.8.1 Composing Two Reducings

Given two reducings, we can create a new reducing by composition.

```
composeReducings
  :: ([Function] -> [Function] -> [Function])
  -> ([Function] -> [Function] -> [Function])
  -> ([Function] -> [Function] -> [Function])
composeReducings s1 s2 focus = s2 focus . s1 focus
```

Note that this is only a useful idea in practice if the two reducings do not both predetermine the size of the result. For example, composing `syntactic-similarity-name` with `syntactic-similarity-type` is not useful because the result will be equivalent to the result of `syntactic-similarity-type`[3]. However, composing, for example, `type-reachability` with `syntactic-similarity-name` could be useful.

3.8.2 Composing Two Drillings

Given two drillings, we can create a new drilling by composition.

[3]This is modulo some details involving the parameter i for each reducing.

```
composeDrillings
  :: ([Function] -> [Function] -> InferM ())
  -> ([Function] -> [Function] -> InferM ())
  -> ([Function] -> [Function] -> InferM ())
composeDrillings d1 d2 focus scope = do
  d1 focus scope
  d2 focus scope
```

In theory composing two drillings could be useful, for example if the two drillings operate fundamentally differently and if they both do not take much time. In practice we only have very similar drillings, so we will not be looking at any compositions of drillings.

3.8.3 Composing a Reducing With a Drilling

Given a reducing and a drilling, we can compose them to make a new drilling as follows.

```
composeReducingAndDrilling
  :: ([Function] -> [Function] -> [Function]
  -> ([Function] -> [Function] -> InferM ())
  -> ([Function] -> [Function] -> InferM ())
composeReducingAndDrilling s d focus =
  d focus . s focus
```

This composition opens up opportunities for new trade-offs. For example, in such a composition, we could combine a reducing that reduces a scope to a constant size with a drilling that does not concern itself with the size of the scope.

3.8.4 Filling the Gaps

Consider all possible compositions of one reducing and one drilling. Any combination of a reducing and `full-background` as the drilling corresponds to that reducing by itself. Any combination of `empty-background` with a drilling will not be much different in practice from just using `empty-background` by itself. Now let us consider the combinations that we have not discussed yet. We can combine each of the reducings with both `chunks` and `chunks-plus` as a drilling, to define the following new strategies.

`chunks-similarity-name`	`chunks-plus-similarity-name`
`chunks-similarity-symbols`	`chunks-plus-similarity-symbols`
`chunks-similarity-type`	`chunks-plus-similarity-type`
`chunks-type-reachability`	`chunks-plus-type-reachability`

Figure 14: Composed signature inference strategies

3.8.5 Special Compositions

The last signature inference strategies that we consider are compositions of two reducings and a drilling. The first reducing is `type-reachability` the second is one of the distance based reducings, and the drilling is `chunks-plus`. At this point we did not consider similar combinations with `chunks` anymore, because the scope size would be constant after the distance based reducing, which meant that the drilling was allowed to be slow. We called these strategies `chunks-plus-reachability-name`, `chunks-plus-reachability-symbols` and `chunks-plus-reachability-type`.

4 Evaluation

In this section we evaluate the different signature inference strategies that were introduced in the previous section.

4.1 Discovery Complexity

The nature of signature inference strategies is that they perform some local computation interleaved with property discovery by QuickSpec. For this work, we conceived a type of complexity analysis that specifically applies to signature inference strategies. We assume that the local computation is negligible compared to the QuickSpec invocations. For the purpose of the analysis, we will consider the local computation free, and we will focus on the complexity of the property discovery. If we view QuickCheck as a "generate and test" method, then its runtime is upper bounded by the number of equations that are tested. This means that we will compute the discovery complexity of a signature inference strategy as the sum of those numbers, over the executions of property discovery.

4.1.1 Maxmimum Number of Discovered Equations

The number of properties of maximum size M that can be discovered in a scope of S functions is related to the number of terms T that can be built using those functions. Because QuickSpec only discovers properties that consist of function applications and variables, we can compute the number of possible terms of maximum size S as follows.
If we assume a fixed number of possible variables V that could occur in a term, the number of possible terms of size one T_1 is $S + V$. For terms of size two, the number of possible terms T_2 is equal to $(S + V)^2$ because every such term is of the form fg where f and g are both terms of size one. For terms of size three, there are two options. A term of size three is either of the form $f\,(gh)$ or of the form $(fg)\,h$. This means that there are $2(S + V)^3$ different terms of size three. In general, the number of possible terms of size n is equal to the number of possible binary trees with n leaves times the number of possible combinations of contents of those leaves $(S + V)^n$. The number of binary trees with n leaves, is well known to be the Catalan number C_{n-1} [10].

$$T_n = C_{n-1}(S+V)^n = \frac{1}{n}\binom{2n-2}{n-1}(S+V)^n$$

The number of terms of *maximum* size n is then a sum as follows.

$$\sum_{i=1}^{M} \frac{1}{n}\binom{2i-2}{i-1}(S+V)^i$$

To arrive at the maximum number of discovered equations of size M from a scope of S functions, we take all possible tuples of terms of maximum size M.

$$\left(\sum_{i=1}^{M} \frac{1}{n}\binom{2i-2}{i-1}(S+V)^i\right)^2$$

Now we only need to choose the number of distinct variables we allow in an equation. Variables are scoped over both sides of the equation, but not across different equations. This means that more than $2M$ different variables will never be used in the same equation, but there could be a (rather useless) equation consisting only of $2M$ different variables. To conclude, the maximum number of equations of maximum size M that can be discovered using a scope of S functions can be computed as follows.

$$\left(\sum_{i=1}^{M} \frac{1}{n}\binom{2i-2}{i-1}(S+2M)^i\right)^2$$

In our experiments, we have fixed the maximum size of discovered equations to be 7. This naturally limits the maximum size of discovered equations in a scope of S functions to the following number.

$$\left(\sum_{i=1}^{7} \frac{1}{n}\binom{2i-2}{i-1}(S+14)^i\right)^2$$

In general, this number is $O\left(S^{2M},\right)$ but because we have fixed M to be 7, this number is $O\left(S^{14}\right)$ for us.

4.1.2 Example

We will look at the discovery complexity of two different signature inference strategies in more detail here. The first is `full-background` as defined in section 3.3 and it is easy to analyse because it performs little local computation and only runs QuickSpec once. For a scope of size S, `full-background` runs QuickSpec once with a scope size of S. This means that the discovery complexity of `full-background` is $O\left(S^{14}\right)$.

Next, consider `chunks` as defined in section 3.6.2. This signature inference strategy performs some local computation to construct tuples of a scope function and a focus function, and runs QuickSpec as many times as there are such tuples. In a situation with S scope functions and F focus functions, there are SF such tuples. This means that `chunks` runs QuickSpec a total of SF times with a scope of constant size 2. The discovery complexity of `chunks` is therefore linear in the number of those tuples.

$$O\left(SF\right)$$

This means that the discovery complexity is linear in the scope size if the focus size is constant (usually it is one). Note that this is a worst case analysis. In practice types and advanced pruning by QuickSpec vastly decreases the number of equations tested [16].

4.2 Evaluators

It is hard to quantify which of two inference strategies is better. We developed an evaluation framework that, for every run of EasySpec, remembers the input, the equations that were discovered, and how long the run took. To define what 'better' means when it comes to inference strategies, we developed the concept

of an evaluator. An evaluator has a name and a way to create a `Maybe Double`, given this information about a run of EasySpec.

```
type Evaluator = EvaluationInput -> Maybe Double
```

We define the following evaluators.

- `equations`: The number of equations that were found
- `runtime`: The amount of time that the run took
- `relevant-equations`: The number of relevant equations that were found
- `relevant-functions`: The number of relevant functions that were found
- `equations-minus-relevant-equations`: The number of irrelevant equations that were found
- `relevant-equations-divided-by-runtime`: The number of relevant equations found per unit of time

In practice, what we are looking for is a strategy that runs in a practical amount of time on a programmer's development machine. What we mean by practical depends on the use case. We had two use cases in mind.
In the first use case, EasySpec would be run at night, to find gaps in a test suite. In this use case, we envision that a linear discovery complexity would be feasible, but anything worse than that would be impractical.
In the second use case, EasySpec would be an interactive assistant that can suggest tests for code that is being written as it is being written. For this purpose, anything slower than a logarithmic discovery complexity would most likely be impractical, and even a logarithmic discovery complexity may be infeasible for large codebases. Because of the difference between complexity and actual runtime, testing out a particular signature inference strategy will still be the best way to tell if it is practical.
A signature inference strategy should not just be fast, but it should also find relevant properties. In practice we only care about properties that are relevant to the focus functions, but it is important to note that all properties take time to discover. This means that we look for a strategy that finds many relevant properties, and ideally few irrelevant properties and therefore few properties beyond the relevant properties.
While a measure like `relevant-equations-divided-by-runtime` can be a useful tool for evaluation, we have not found a single evaluator to rank signature inference strategies by [4]. The best way to evaluate signature inference strategies that we have found is to first decide if the runtime is practical, and among practical signature inference strategies, choose the signature inference strategy with the highest score for `relevant-equations`. We chose to compare other strategies with `full-background` mostly, because the output of `full-background` most accurately represents what was already possible with QuickSpec.

[4] For example, `empty-background` scores very well on the `relevant-equations-divided-by-runtime` evaluator, but is hardly the best signature inference strategy.

4.3 Experiments

To evaluate EasySpec and the different signature inference strategies, we performed multiple experiments. In a perfect world, we would run EasySpec on real world Haskell code. However, Haskell is a large language when it comes to syntax, and the GHC compiler extensions make the language bigger. Due to time constraints, EasySpec could not be readied for use on real world code. As a result, experiments had to be performed on subject code that was limited to the part of the language that is currently supported. We put together example code with as large a scope as `full-background` could handle with a diverse mixture of functions and types. For each of these examples, for each signature inference strategy and for each function in the module, EasySpec is run using that function as the focus. The code samples can be found in the EasySpec repository, [23] and consist of the following major groups.

- `Bools.hs` consists of Boolean functions, operators, and list functions that use Boolean filter functions.
- `Monoid.hs` consists of integer functions, operators and constants.
- `DNA.hs` consists of functions that deal with strings characters.

These examples are together comprehensive in the sense that they contain all the types that QuickSpec supports by default. We made a separate group of code samples to evaluate the runtime aspect of signature inference strategies. These files consist of three locally defined functions that resemble `id`, `++` and `reverse` and an increasing scope of prelude functions around them.

4.4 Strategies

4.4.1 Empty Background

In Figure 15, the runtime of `empty-background` is plotted compared to the runtime of `full-background`. Because the signature that QuickSpec is run on in `empty-background` is of constant size, `empty-background` will always run in constant time. This means that `empty-background` is a practical signature inference strategy to use.

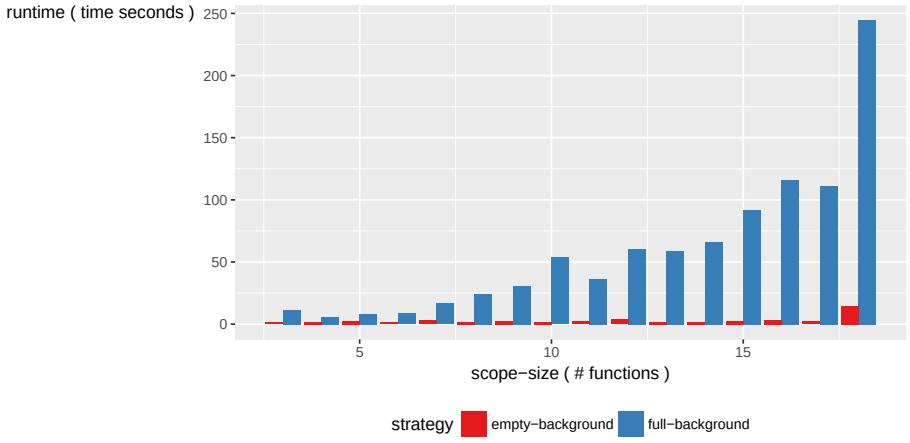


Figure 15: Runtime of `empty-background` .

However, when we look at Figure 16, we see that `empty-background` finds almost no relevant equations. Figure 16 shows a box plot of the number of relevant equations found, comparing `empty-background` and `full-background` .

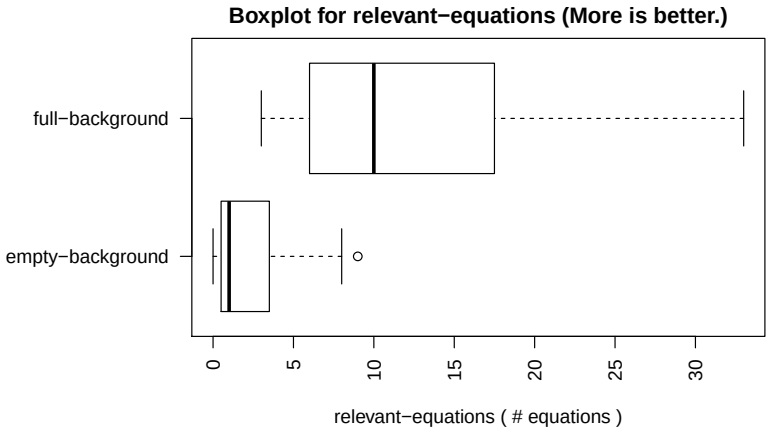


Figure 16: Relevant equations of `empty-background`

Because the `empty-background` signature inference strategy only finds equations that only consist of functions in focus, it misses out on most of the relevant equations.

4.4.2 Syntactic Similarity

All distance based signature inference strategies have a parameter i that determines the maximum size of the signature to select. We chose to fix this parameter to 5 such that the runtime of each of these signature inference strategies would remain constant as well.

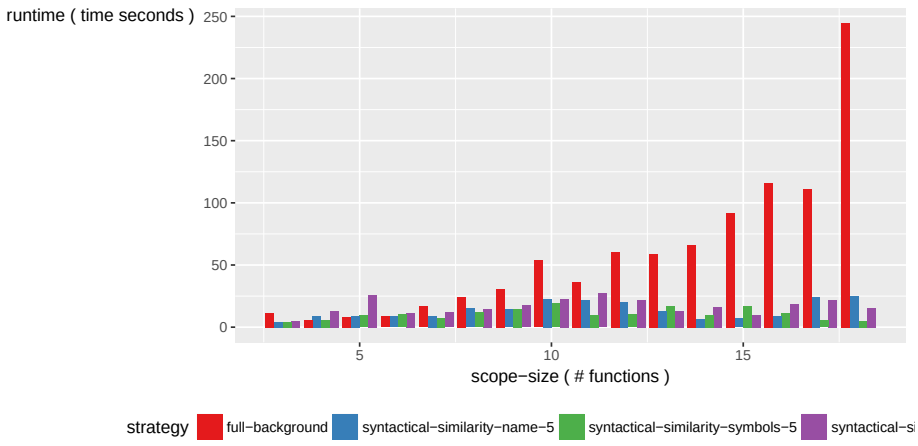


Figure 17: Runtime of the syntactic similarity signature inference strategies: `syntactic-similarity-name`, `syntactic-similarity-symbols` and `syntactic-similarity-type`

In Figure 17, we find the runtimes of `full-background`, `syntactic-similarity-name`, `syntactic-similarity-symbols` and `syntactic-similarity-type`. As is to be expected, all of these signature inference strategies run in constant time and are practical as such.

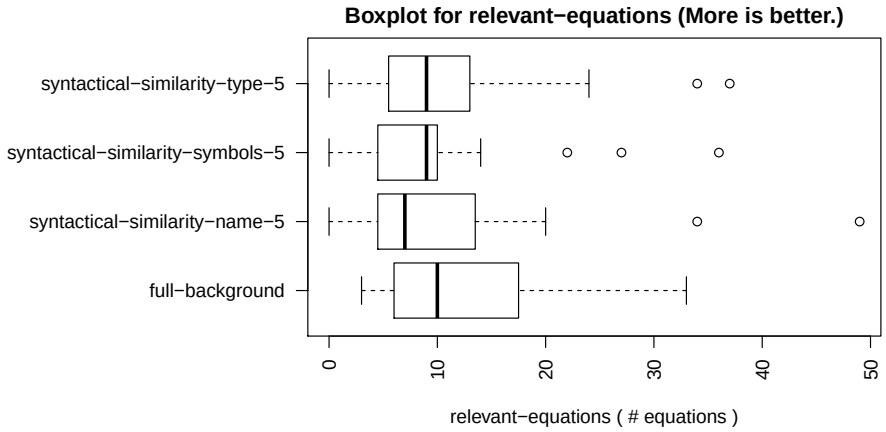


Figure 18: Relevant equations of the syntactic similarity signature inference strategies: `syntactic-similarity-name`, `syntactic-similarity-symbols` and `syntactic-similarity-type`

In Figure 18, we find a box plot of the number of relevant equations that each of these strategies discover. As it turns out, these strategies already find a good number of equations. However, the way we have set up our experiments may have skewed these numbers because the number we fixed for the parameter i: 5

is already a significant fraction of the size of the scope. This would not be the case in practice. It seems that choosing smaller signatures to run QuickSpec on is a good idea, but that these strategies are not ideal in deciding which functions to put in the smaller signature.

4.4.3 Type Reachability

The `type-reachability` signature inference strategy is different from the distance based signature inference strategies because it does not guarantee that the reduced scope is any smaller than the original scope. As such, it is not guaranteed to be any faster than `full-background`. This could make `type-reachability` infeasible for use in certain situations, but as we can see in Figure 19, the experiments that we used do not cause `type-reachability` to exhibit this problem.

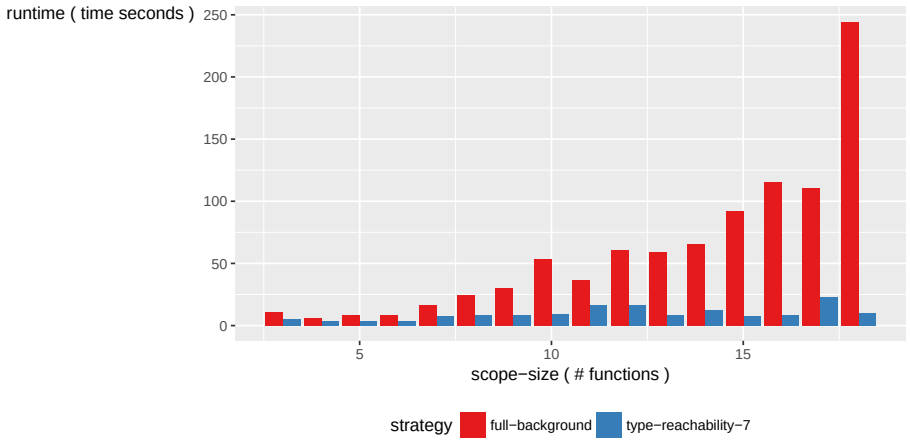


Figure 19: The runtime of `type-reachability`

In practice, we see that `type-reachability` reduces the scope to a sufficiently small subset such that the runtime is subsequently small enough to be practically feasible. As for the discovered equations, in Figure 20, we find that `type-reachability` is not better than `full-background`, but it is at least as promising as the distance based signature inference strategies.

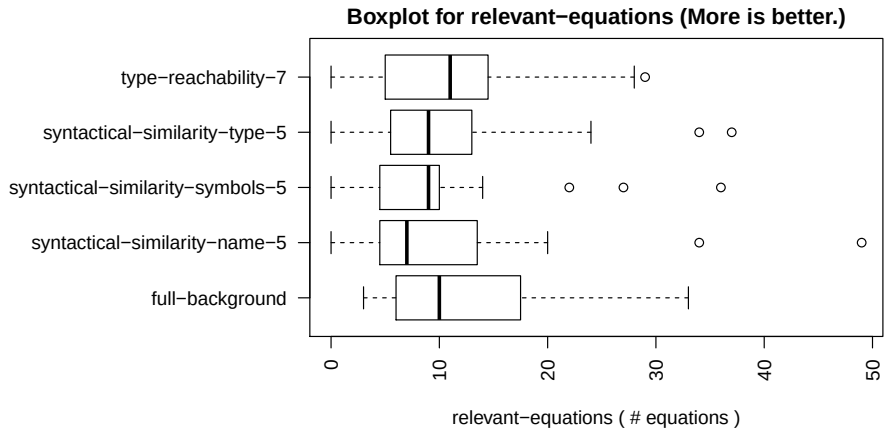


Figure 20: The number of relevant equations of the reducing signature inference strategies

4.4.4 Chunks

The `chunks` signature inference strategy runs QuickCheck on many signatures of constant size. To be precise, `chunks` runs QuickSpec on exactly SF signatures of size 2 and one more signature of size F. Here, S is the size of the scope, and F is the size of the focus. As a result, we expect `chunks` to have a linear discovery complexity with respect to the size of the scope.

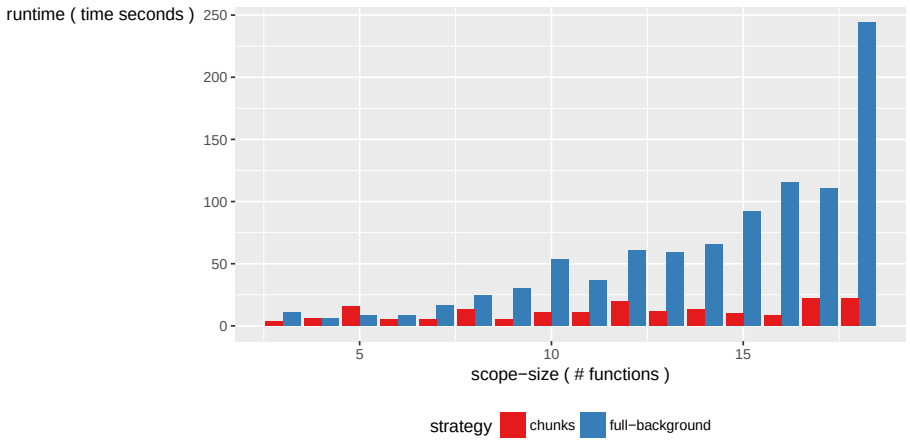


Figure 21: The number of relevant equations of `chunks`

Figure 21 confirms this expectation. For the interactive use case, `chunks` may not be practical, but for the nightly use case, it could be. A user would have to evaluate whether the strategy is fast enough for their use case. Moreover, the `chunks` signature inference strategy may still be a useful building block for developing better signature inference strategies.
For the `chunks` signature inference strategy, we expected to find the equations

that `full-background` finds, but only the ones that mention two or fewer distinct functions.

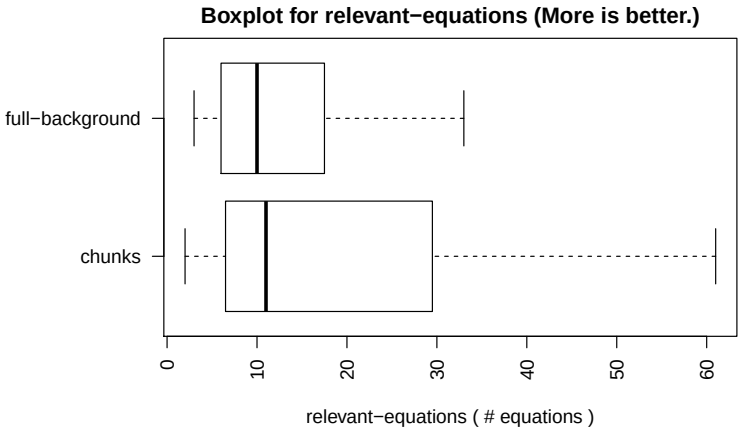


Figure 22: The number of relevant equations of `chunks`

However, Figure 22, shows that `chunks` often even finds even more relevant properties than `full-background` does. This may seem curious at first, but it is explained by the fact that QuickSpec outputs only the most general properties it discovers. With a larger scope, general properties that subsume others are more likely to be discovered. With more scope, properties are more likely to have a more general property that will be discovered. If the context is reduced, such as in the case of `chunks`, QuickSpec is more likely to find multiple different relevant equations that could generalise to fewer equations if QuickSpec had more context. As an example, consider the following scope.

```
a :: Int -> Int
a = (+1)
b :: Int -> Int
b = (+2)
c :: Int -> Int
c = (+3)
d :: Int -> Int
d = (+4)
```

When we run `full-background` on this scope, we find the following equations.

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

If we chose `d` as the focus, only the last equation would be considered relevant. This means that `full-background` only finds one relevant equation. When we run `chunks` on this scope with focus `d` we find the following relevant equations.

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

Note that neither of these equations are found in the `full-background` results, because they are more specific than some of the equations that `full-background` finds. The reduction of the scope caused `chunks` to find more relevant equations than `full-background` did.

4.4.5 Chunks Plus

We expect `chunks-plus` to have a quadratic discovery complexity. As such, `chunks-plus` is most likely not a practical signature inference strategy to use by itself in the use cases that we had in mind. However, as a building block, `chunks-plus` may still be useful.

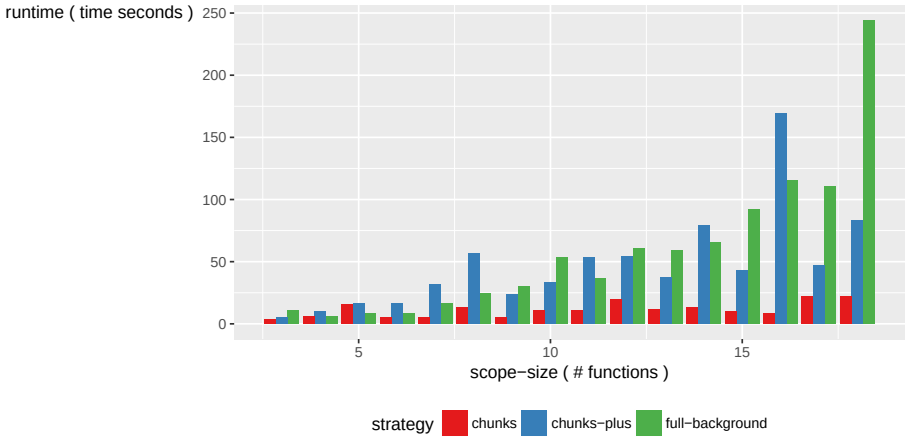


Figure 23: The number of relevant equations of `chunks-plus`

In Figure 23, we see that `chunks-plus` has the complexity that we expected. This plot also confirms that `chunks-plus` would be impractical to use by itself. As for the relevant equations that `chunks-plus` finds, we expect that it finds at least as many as `chunks` does.

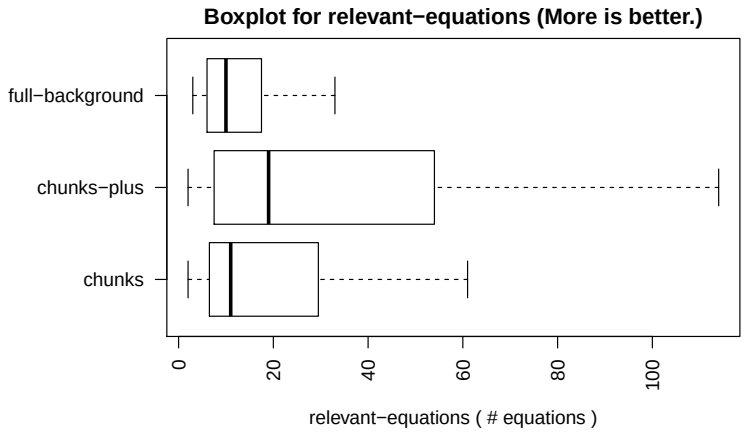
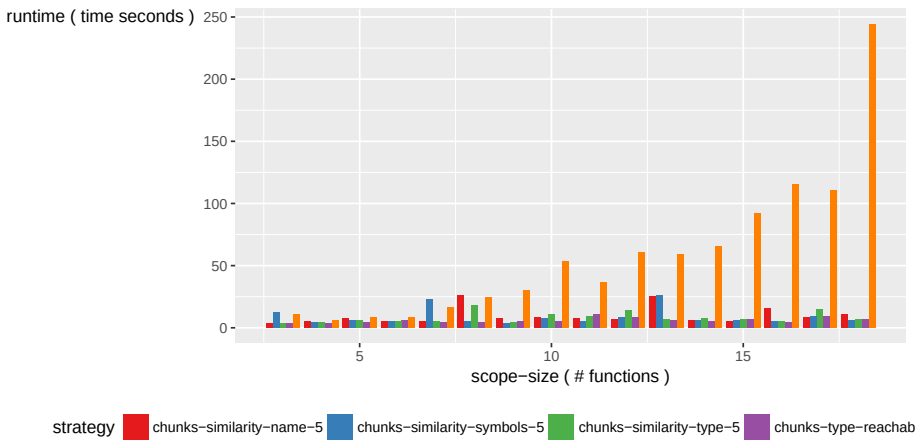


Figure 24: The number of relevant equations of `chunks-plus`

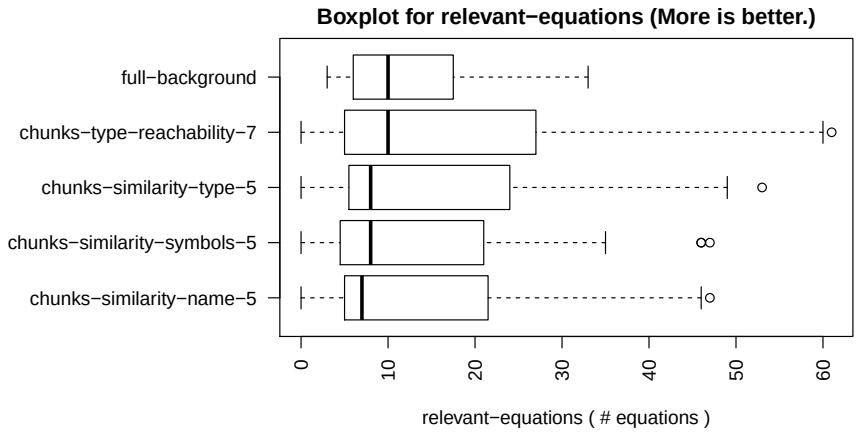
In Figure 24, we see that this is indeed the case. While the runtime of `chunks-plus` prohibits it from being used in practice, its time complexity is still twelve factors of the scope size faster than `full-background` is, and it regularly finds more relevant equations.

4.4.6 Compositions

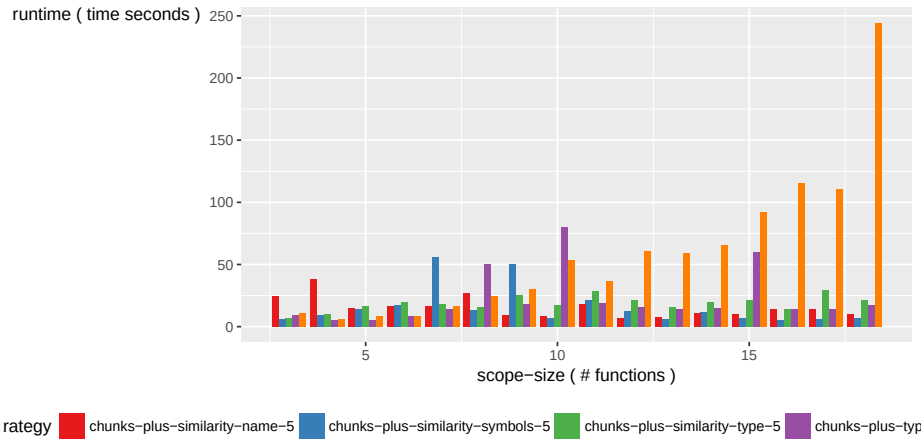
In Figure 4.4.6, we see that all the compositions of a reducing with `chunks` yield a signature inference strategy that runs in a practically feasible amount of time, even `chunks-type-reachability`.



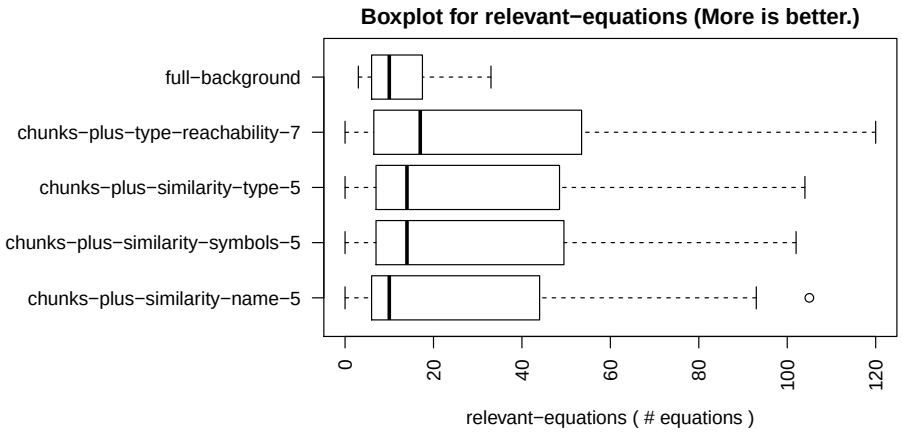
In Figure , we see that these composed strategies all approach `full-background` in terms of how many relevant equations they find.



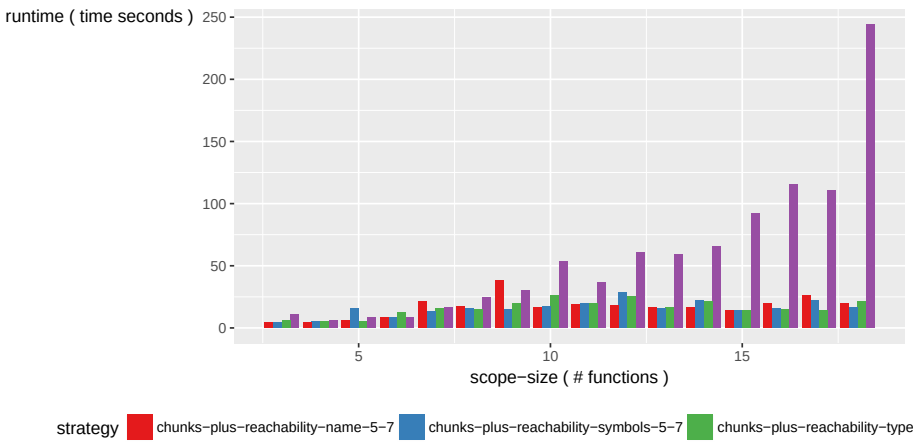
As we can see in Figure 4.4.6, composing the same reducings with a more intensive drilling: `chunks-plus` produces signature inference strategies that still run in a practical amount of time, but still a greater amount of time, even `chunks-plus-type-reachability`.



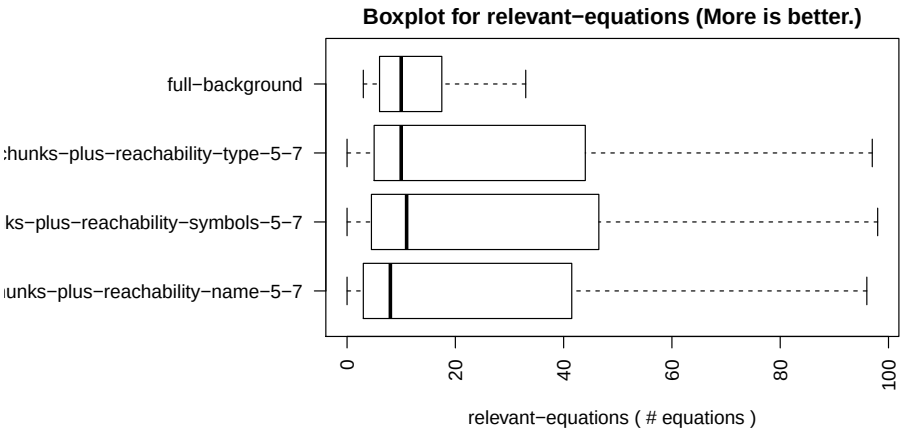
In Figure 4.4.6 we find that these compositions regularly outperform `full-background`.



The compositions of two reducings with a drilling produced less successful signature inference strategies than expected. As we can see in Figure 4.4.6, these strategies still ran in a practical amount of time.

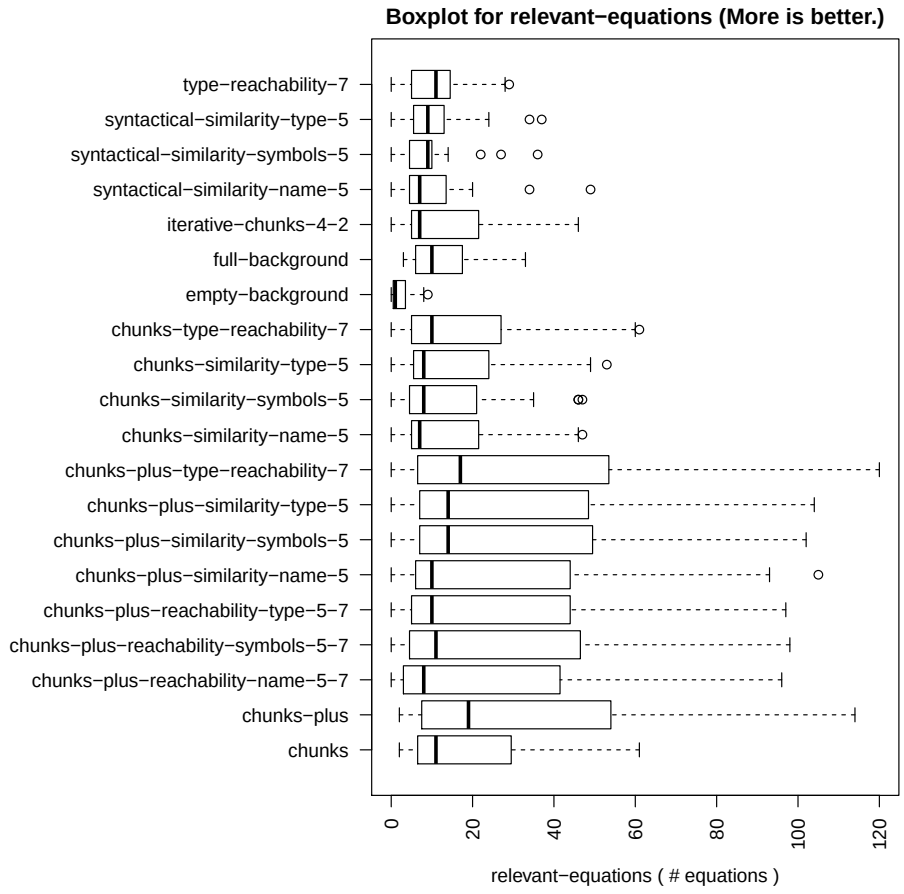


However, if we look at Figure 4.4.6, we see that these more complex compositions do not outperform their simpler variants.



4.4.7 Overview

In Figure 4.4.7, there is an overview of the performance, with respect to the number of relevant equations found, of all the signature inference strategies that we studied.



The best performing signature inference strategy is clearly `chunks-plus` and, while it is still significantly faster than `full-background`, it may not run in a practically feasible amount of time. Of the signature inference strategies that run in a feasible amount of time, `chunks-plus-type-reachability` finds the most relevant equations. If one needs a constant time guarantee, then `chunks-plus-similarity-type` or `chunks-plus-similarity-symbols` are viable alternatives.

5 Discussion

This work has explored several different signature inference strategies. A summary can be found in Figure 25.

Strategy	DC	DC'
`full-background`	$O\left(S^{2M}\right)$	$O\left(S^{14}\right)$
`empty-background`	$O\left(F^{2M}\right)$	$O\left(1\right)$
`syntactic-similarity-name`	$O\left(i^{2M}\right)$	$O\left(1\right)$
`syntactic-similarity-symbols`	$O\left(i^{2M}\right)$	$O\left(1\right)$
`syntactic-similarity-type`	$O\left(i^{2M}\right)$	$O\left(1\right)$
`type-reachability`	$O\left(S^{2M}\right)$	$O\left(S^{14}\right)$
`chunks`	$O\left(SF2^{2M}\right)$	$O\left(S\right)$
`chunks-plus`	$O\left(S^2F3^{2M}\right)$	$O\left(S^2\right)$
`chunks-similarity-name`	$O\left(Fi2^{2M}\right)$	$O\left(1\right)$
`chunks-similarity-symbols`	$O\left(Fi2^{2M}\right)$	$O\left(1\right)$
`chunks-similarity-type`	$O\left(Fi2^{2M}\right)$	$O\left(1\right)$
`chunks-type-reachability`	$O\left(SF2^{2M}\right)$	$O\left(S\right)$
`chunks-plus-similarity-name`	$O\left(Fi^23^{2M}\right)$	$O\left(1\right)$
`chunks-plus-similarity-symbols`	$O\left(Fi^23^{2M}\right)$	$O\left(1\right)$
`chunks-plus-similarity-type`	$O\left(Fi^23^{2M}\right)$	$O\left(1\right)$
`chunks-plus-type-reachability`	$O\left(S^2F3^{2M}\right)$	$O\left(S^2\right)$
`chunks-plus-reachability-name`	$O\left(Fi^23^{2M}\right)$	$O\left(1\right)$
`chunks-plus-reachability-symbols`	$O\left(Fi^23^{2M}\right)$	$O\left(1\right)$
`chunks-plus-reachability-type`	$O\left(Fi^23^{2M}\right)$	$O\left(1\right)$

Figure 25: A summary of the different signature inference strategies

The symbols in this table are defined as follows.

- S: size of the scope
- M: maximum size of the discovered properties
- F: size of the focus
- i: chosen size of the signature in a reducing signature inference strategy
- j: chosen depth of type reachability
- DC: worst case discovery complexity
- DC': worst case discovery complexity, when M is fixed to be 7, F is fixed to be 1, i is fixed to be 5 and j is fixed to be 7; This is the configuration that we chose.

5.1 Configurability

The exploration of different signature inference strategies has had the nice side effect that signature inference can now be configured to use a chosen signature inference strategy.
Indeed, different signature inference strategies make it possible to support multiple use cases. One potential user interface involves online property discovery and immediate feedback. This use case can be enabled by choosing a signature inference strategy that has an appropriate discovery complexity.
Another potential user interface involves running property discovery overnight. In this case there is a larger number of feasible signature inference strategies available.
Users are not forced to use any single signature inference strategy, and are invited to choose their own and evaluate whether it is appropriate for their use case.

5.2 Shortcomings

EasySpec suffers from most of the functional shortcomings that QuickSpec has. This includes false positives, which means that sometimes properties are discovered that do not hold. This is not a big problem since a human must still select the properties that they want to have hold, and the properties are still tested afterwards with different random input.
Both QuickSpec and EasySpec can only discover properties that already hold (modulo false positives). This means that properties that you may want to have hold about code will not be discovered if the code does not already satisfy those properties.
Higher kinded type variables are not supported in EasySpec because their monomorphisation still has to be done manually in QuickSpec and QuickSpec has no dummy higher kinded variables as it does for other variables.
Both QuickSpec and EasySpec use the `arbitrary` generators from the `Arbitrary` type class to generate random values. Custom generators could be of great value if certain properties only hold for a subset of a type, but neither QuickSpec nor EasySpec currently supports them. Furthermore, EasySpec currently does not find any instances that are in scope, so EasySpec will only operate on types of which QuickSpec already has the Arbitrary instance built-in.
Lastly, because EasySpec uses the interactive evaluator that built into GHC by interpolating Strings, there are many issues with respect to modules and unexported symbols. For example, EasySpec does not work well on modules that export functions of which the type contains unexported symbols, such as the function `error :: HasCallStack a => String -> a` wherein `HasCallStack` is not exported.
EasySpec uses the GHC API to type check code, and translates the resulting types to a representation defined in an external library. This translation allowed for quicker iteration because the translated representation was easier to work with than internal representation in GHC, but the translation is not lossless. Therefore the translation incurs several limitations. It further complicates certain common practical situations such as modules and unexported symbols. Moreover, it also prevents us from using the type checking mechanisms within GHC to implement type reachability.

6 Conclusion

QuickSpec has made great progress toward practical property discovery. It looked promising, until we had a look at the complexity of QuickSpec with respect to realistic codebases. Signature inference has proved to be a promising approach to making property discovery practical by taming the complexity of QuickSpec. We provided options to decrease the discovery complexity from $O\left(S^{14}\right)$ to $O\left(S\right)$ or $O\left(1\right)$. The progress that we achieved on making property discovery practical for codebases of realistic sizes promises that property discovery could become a real and very valuable tool in software development.

6.1 Future Work

Significant engineering effort is required to make property discovery realistically usable. Much of the necessary research is done, but significant corners had to be cut with respect to practical usability. Nevertheless, there is still room for more research on this topic.

The signature inference strategies discussed in this work only ever choose functions out of the scope and translate their types literally. These strategies never attempt to add anything to the signature that was not already in scope. It may be useful to consider adding entirely new functions or expressions to a signature. Literals are a good example of such expressions: it could be useful to add the zero constant to the signature, if the focus involves numbers. Similarly, it may be useful to add the empty list to the scope, if the focus involves lists, etc.

Automatic monomorphisation of higher kinded type variables has been glossed over in this work, because it is a nontrivial concept to implement. Higher kinded type variables cannot simply be translated in the same way that simple type variables can be translated because QuickSpec currently has no support for such a translation.

Type class instances should be discovered by the signature inference tool, but currently are not. This limitation has cut down the practical significance of EasySpec significantly, and requires engineering effort to lift. However, instance discovery could also aid signature inference.

References

[1] Colin Runciman and Matthew Naylor and Fredrik Lindblad. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008*, pages 37–48, 2008.

[2] Cordelia V. Hall and Kevin Hammond and Simon L. Peyton Jones and Philip Wadler. Type Classes in Haskell. *ACM Trans. Program. Lang. Syst.*, 18(2):109–138, 1996.

[3] Edsger W. Dijkstra. The Humble Programmer. *Commun. ACM*, 15(10):859–866, 1972.

[4] GHC Developers. Glasgow Haskell Compiler version 8.0.2, 2016.

[5] GHC Developers. Glasgow Haskell Compiler API version 8.0.2, 2017.

[6] haskell.org. Haskell Language Front Page.

[7] Jasmin Christian Blanchette and David Greenaway and Cezary Kaliszyk and Daniel Kühlwein and Josef Urban. A Learning-Based Fact Selector for Isabelle/HOL. *J. Autom. Reasoning*, 57(3):219–244, 2016.

[8] Jia Meng and Lawrence C. Paulson. Lightweight relevance filtering for machine-generated resolution problems. volume 7, pages 41–57, 2009.

[9] John Peterson and Mark P. Jones. Implementing Type Classes. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, pages 227–236, 1993.

[10] Knuth, Donald E. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[11] Koen Claessen. QuickCheck: Automatic testing of Haskell programs. `http://hackage.haskell.org/package/QuickCheck-2.9.2`, September 2016.

[12] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000.*, pages 268–279, 2000.

[13] Koen Claessen and Moa Johansson and Dan Rosén and Nicholas Smallbone. *Automating Inductive Proofs Using Theory Exploration*, pages 392–406. 2013.

[14] Koen Claessen and Nicholas Smallbone and John Hughes. QuickSpec: Guessing Formal Specifications Using Testing. In *Tests and Proofs, 4th International Conference, TAP 2010, Málaga, Spain, July 1-2, 2010. Proceedings*, pages 6–21, 2010.

[15] Lee Pike. SmartCheck: automatic and efficient counterexample reduction and generalization. In *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*, pages 53–64, 2014.

[16] Nicholas Smallbone and Moa Johansson and Koen Claessen and Maximilian Algehed. Quick specifications for the busy programmer. *J. Funct. Program.*, 27:e18, 2017.

[17] Nick Smallbone. QuickCheck: Automatic testing of Haskell programs. `https://github.com/nick8325/quickspec/tree/3c6e0105374bcf1ed0d4f8d2a1a1d2875764fa56`, 2016.

[18] Philip Lee Wadler. Letter to Haskell working group, February 1988.

[19] Philip Wadler and Stephen Blott. How to Make ad-hoc Polymorphism Less ad-hoc. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*, pages 60–76, 1989.

[20] Rudy Braquehais and Colin Runciman. FitSpec: refining property sets for functional testing. In *Proceedings of the 9th International Symposium on Haskell, Haskell 2016, Nara, Japan, September 22-23, 2016*, pages 1–12, 2016.

[21] Rudy Braquehais and Colin Runciman. Speculate: discovering conditional equations and inequalities about black-box functions by reasoning from test results. In *Haskell Symposium 2017*. ACM, 2017.

[22] Rudy Matela. LeanCheck: Cholesterol-free property-based testing for Haskell. `https://hackage.haskell.org/package/leancheck-0.6.2`, March 2017.

[23] Tom Sydney Kerckhove. EasySpec: Signature Inference for Functional Property Discovery. `https://github.com/NorfairKing/easyspec/tree/ec1c933e7c647a010e941ca36662dc23ded3c511`, 2017.

[24] unknown. Typeable @ base-4.9.1.0. `http://hackage.haskell.org/package/base-4.9.1.0/Data-Typeable.html#t:Typeable`, January 2017.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Signature Inference for Functional Property Discovery

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Kerckhove	Tom Sydney

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zürich, 2017-09-08	**Tom Sydney Kerckhove**

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.