CHALMERS
UNIVERSITY OF TECHNOLOGY

Conjectures, tests and proofs: An overview of theory exploration

Citation for the original published paper (version of record):
Johansson, M., Smallbone, N. (2021). Conjectures, tests and proofs: An overview of theory exploration. Electronic Proceedings in Theoretical Computer Science, EPTCS, 341: 1-16. http://dx.doi.org/10.4204/EPTCS.341.1

Conjectures, Tests and Proofs: An Overview of Theory Exploration

Moa Johansson and Nicholas Smallbone
Chalmers University of Technology, Gothenburg, Sweden
moa.johansson@chalmers.se, nicsma@chalmers.se

A key component of mathematical reasoning is the ability to formulate interesting conjectures about a problem domain at hand. In this paper, we give a brief overview of a theory exploration system called QuickSpec, which is able to automatically discover interesting conjectures about a given set of functions. QuickSpec works by interleaving term generation with random testing to form candidate conjectures. This is made tractable by starting from small sizes and ensuring that only terms that are irreducible with respect to already discovered conjectures are considered. QuickSpec has been successfully applied to generate lemmas for automated inductive theorem proving as well as to generate specifications of functional programs. We give an overview of typical use-cases of QuickSpec, as well as demonstrating how to easily connect it to a theorem prover of the user's choice.

1 Introduction

What makes a mathematical conjecture interesting and worth trying to prove as a lemma? For a human mathematician the motivation might be that the statement would make another proof shorter, clearer and easier to understand, or that the lemma captures, for instance, some common algebraic property of a structure of interest. Coming up with the right lemma in the right situation is sometimes described as a *eureka step*: a sudden insight that makes a solution almost obvious.

What about conjectures in programming? For a programmer, an interesting conjecture about a program at hand should shed light on its intended use and form part of a specification, which in turn can be used to for example generate test cases for the program, or even prove its correctness. Writing detailed formal specifications is however in practice something few programmers have the time or expertise to do.

Automating these kind of creative steps in automated reasoning and program analysis is difficult both for symbolic and data-driven methods: Interesting lemmas might not fall out from simply applying deductive rules to axioms in a theorem prover; and when considering a new mathematical theory or program, we might not have a lot of previous knowledge in the domain, which makes it difficult to directly apply traditional machine learning techniques.

In the context of automation of inductive proofs, there has been work on patching failed proof attempts by speculating lemmas using e.g. various *proof critics*, ranging from simply generalising the goal by replacing common subterms with new variables, to more complex methods requiring specialised heuristics and search [2, 16, 9]. In this article we will describe a different approach, *theory exploration*, which instead of trying to construct lemmas from specific proof attempts, proceeds bottom-up: given a mathematical theory or program, what interesting conjectures can we come up with?

We will demonstrate various use-cases of theory exploration centred around our tool QuickSpec [26]. QuickSpec generates conjectures about a set of functions and datatypes given by the user. It works by enumerating terms starting from small sizes up to a user specified limit, and then evaluating them at random values to determine if any terms appear to be equal. To avoid uninteresting conjectures, and

Alexei Lisitsa and Andrei Nemytykh (Eds.): 9th International Workshop on Verification and Program Transformation (VPT 2021)
EPTCS 341, 2021, pp. 1–16, doi:10.4204/EPTCS.341.1

to manage the search space, only terms that have a unique normal form with respect to what has been discovered so far are kept. To give the reader a flavour of what QuickSpec can do, Figure 1 shows the *complete* output of QuickSpec on the theory of natural numbers with addition, multiplication and greatest common divisor, implemented as functions in Haskell. QuickSpec takes about 5 seconds to run on this example. Note that QuickSpec has no built-in knowledge of the laws of arithmetic, so everything in Figure 1 is invented from scratch. These conjectures include common properties such as commutativity,

```
 1. x+y = y+x                 13. gcd(x,0) = x
 2. x+0 = x                   14. gcd(x,1) = 1
 3. (x+y)+z=x+(y+z)           15. gcd(x,x*y) = x
 4. x*y = y*x                 16. gcd(x,x+y) = gcd(x,y)
 5. x*0 = 0                   17. gcd(gcd(x,y),z) = gcd(x,gcd(y,z))
 6. x*1 = x                   18. gcd(x*y,x*z) = x*gcd(y,z)
 7. (x*y)*z = x*(y*z)         19. gcd(x*x,y*y) = gcd(x,y)*gcd(x,y)
 8. x*(y+y) = y*(x+x)         20. gcd(x*y,z+y) = gcd(x*z,z+y)
 9. x*(y+1) = x+(x*y)         21. gcd(x+x,y+y) = gcd(x,y)+gcd(x,y)
10. (x*y)+(x*z) = x*(y+z)     22. gcd(x+y,y+y) = gcd(x+x,x+y)
11. gcd(x,y) = gcd(y,x)       23. gcd(x*x,1+1) = gcd(x,1+1)
12. gcd(x,x) = x
```

Figure 1: Complete QuickSpec output on a small theory about arithmetic.

associativity, distributivity and identity properties, as well as more specific facts about `gcd`, such as property 18 (about numbers having common factors), property 19 (about squares), the curious interchange property 20, and property 23 (which holds because 2 is prime). Notice that after discovering for instance $x*0=0$ (property 5), QuickSpec will not generate any terms where $x*0$ is a subterm, as such a term would be reducible by applying property 5 as a rewrite rule, and therefore is not considered interesting. This is one of the key observations that makes term generation tractable (see [26] for full technical details and optimisations of the term generation algorithm). Note that QuickSpec itself does not prove any properties, but tests them thoroughly on randomly generated values, using Haskell's QuickCheck tool [4]. Depending on the theory we are exploring we can of course pass the properties discovered by QuickSpec on to a suitable automated or interactive theorem prover.

2 A Small QuickSpec Tutorial

In this section we demonstrate QuickSpec on two examples taken from programming: a data structure implementing maps from keys to values, and a library for describing musical compositions. The section serves as a tutorial for the aspiring QuickSpec user: we describe in detail how we ran QuickSpec, how and why we fine-tuned its settings for each example, and what obstacles we encountered along the way.

QuickSpec can be downloaded from `https://github.com/nick8325/quickspec`. The examples in this section are found in the following files in that repository:

- Greatest common divisor: `examples/GCD.hs`.
- Maps: `examples/Map.hs`.
- Music: `examples/Music/MusicQS.hs`.

2.1 Greatest Common Divisor

We first show how the conjectures in Figure 1 were produced. To run QuickSpec, the user first defines a *signature*, which describes the set of functions we would like to explore (along with other configuration settings that we shall see later). The signature that we used for the arithmetic theory is shown in Figure 2.

```
exampleSig = series [arithSig, gcdSig]
  where
    arithSig =
      [con "0" (0 :: Natural),
       con "1" (1 :: Natural),
       con "+" ((+) :: Natural -> Natural -> Natural),
       con "*" ((*) :: Natural -> Natural -> Natural)]
    gcdSig =
      [con "gcd" (gcd :: Natural -> Natural -> Natural)]
```

Figure 2: A QuickSpec signature for the greatest common divisor function.

The function `con` is used in the signature to declare a Haskell constant or function. It takes two arguments: the first is the name of the constant, and the second is the constant itself. QuickSpec has no built-in constants that it automatically considers, so if we want to see conjectures involving `0` and `1`, we need to explicitly declare them. Type signatures are used to specialise our arithmetic functions to the natural numbers—any typeclass-polymorphic function must be specialised to a concrete type.

We also use the function `series`. The expression `series [arithSig, gcdSig]` tells QuickSpec that we would like to split the exploration into two parts: First we would like to explore the operations `0`, `1`, `+` and `*`, and once that is done we would like to add `gcd` to the mix. By doing this, we see conjectures about `gcd` only once all conjectures for the other operations have been generated.

Having written this signature, we can run QuickSpec by typing in `quickSpec exampleSig` at the Haskell prompt, and the conjectures in Figure 1 are produced.

2.2 Maps from Keys to Values

The Haskell standard library defines a module `Data.Map` which implements a map from keys to values. Figure 3 shows a small part of the map API, which we will now explore with QuickSpec.

The type `Map k v` represents a map from keys of type `k` to values of type `v`. The constant `empty` is an empty map; `insert` adds a key-value pair to a map; `delete` removes a key-value pair given a key; and `lookup` looks up a key to obtain a value, returning `Nothing` if the key is not found. The last three functions have an `Ord k` constraint, meaning that they require the key type to have a suitable total order.

```
type Map k v
empty :: Map k v
insert :: Ord k => k -> v -> Map k v -> Map k v
delete :: Ord k => k -> Map k v -> Map k v
lookup :: Ord k => k -> Map k v -> Maybe v
```

Figure 3: An excerpt from the `Map` module.

QuickSpec supports testing polymorphic functions, but its support for typeclass polymorphism is immature. To test the map API, we will therefore specialise the map functions to a specific type of keys and values. We start by defining two types `Key` and `Val`, which are thin wrappers around integers:

```
newtype Key = Key Int deriving (Eq, Ord, Arbitrary)
newtype Val = Val Int deriving (Eq, Ord, Arbitrary)
```

Then we can define a signature, which is shown in Figure 4. We note three things about this signature:

- When a function involves a user-defined type, we must declare that types in the signature. Here, we use the command `monoTypeWithVars` to declare the types `Key`, `Val` and `Map Key Val`. The string argument (`"k"`, `"v"`, `"m"`) specifies what names should be used for variables of that type when printing conjectures.
- Because `lookup` returns a `Maybe Val`, its laws may involve the `Maybe` constructors `Just` and `Nothing`, which we therefore add to the signature. However, we are not interested in `Just` and `Nothing` in themselves, but only in their interactions with the map operations. We therefore declare them as *background functions* using the `background` combinator. Background functions can appear in conjectures but any conjecture must involve at least one non-background function.
- The `A` in the type signatures for `Nothing` and `Just` is a polymorphic *type variable*. Polymorphic functions can be declared in QuickSpec by using the types `A`, `B`, `C` etc. in type declarations. QuickSpec will then take care of instantiating the functions at appropriate types.

```
mapsSig =
  [monoTypeWithVars ["k"] (Proxy :: Proxy Key),
   monoTypeWithVars ["v"] (Proxy :: Proxy Val),
   monoTypeWithVars ["m"] (Proxy :: Proxy (Map Key Val)),
   background [
     con "Nothing" (Nothing :: Maybe A),
     con "Just" (Just :: A -> Maybe A)],
   series [sig1, sig2, sig3]
   where
     sig1 =
       [con "empty" (Map.empty :: Map Key Val),
        con "lookup" (Map.lookup :: Key -> Map Key Val -> Maybe Val)]
     sig2 =
       [con "insert" (Map.insert :: Key -> Val -> Map Key Val -> Map Key Val)]
     sig3 =
       [con "delete" (Map.delete :: Key -> Map Key Val -> Map Key Val)]
```

Figure 4: A signature for maps.

The reader might wonder why we go to the trouble of defining the types `Key` and `Val`, rather than simply using `Map Int Int`. We in fact used `Map Int Int` at first, but got several strange properties in which the same variable appeared both as a key and a value, such as the following:

```
lookup x (insert y y m) = lookup x (insert y x m)
insert x y (insert y x m) = insert y x (insert x y m)
insert x x (insert y y m) = insert y y (insert x x m)
```

Because these properties mix up keys and values, they are not useful. By keeping `Key` and `Val` distinct, we avoid generating them. In general, we have found that QuickSpec finds better properties when conceptually different types are kept distinct.

Running `quickSpec mapsSig` produces the following 11 conjectures:

```
 1. lookup k empty = Nothing
 2. lookup k (insert k v m) = Just v
 3. lookup k (insert k2 v empty) = lookup k2 (insert k v empty)
 4. insert k v (insert k v2 m) = insert k v m
 5. insert k v (insert k2 v m) = insert k2 v (insert k v m)
 6. delete k empty = empty
 7. lookup k (delete k m) = Nothing
 8. delete k (delete k2 m) = delete k2 (delete k m)
 9. delete k (delete k m) = delete k m
10. delete k (insert k v m) = delete k m
11. insert k v (delete k m) = insert k v m
```

These properties are quite revealing of how the map API works. For example, what does the `insert` function do when the key to be inserted is already present in the map? It might conceivably throw an exception, or leave the map unchanged. But property 4 reveals that the new value overwrites the old value. Similarly, property 9 shows that deleting a key which is not present does nothing. Properties 10 and 11 show that each of `delete` and `insert` undoes the effect of the other. Property 2 shows that `lookup` retrieves a value added by `insert`. Property 8 shows that `delete` commutes with itself: when deleting two keys from the map, the order in which we delete the pairs doesn't matter.

We might wonder why there is no similar commutativity property for `insert`—that is, a conjecture stating that, when we add two key-value pairs to the map, the order in which we add the pairs doesn't matter. Formally, this can be expressed as the property

```
insert k v (insert k2 v2 m) = insert k2 v2 (insert k v m)
```

The reason this property is not found is because it doesn't hold! If the two keys are the same, that is `k = k2`, then the order of insertion *does* matter, because the value which is inserted last takes precedence. The property holds only if `k /= k2`.

QuickSpec can discover such conditional properties, but we must explicitly declare what conditions we are interested in. The condition we need here is inequality between keys. We declare it by adding the following line to the signature, in the background function section:

```
predicate "/=" ((/=) :: Key -> Key -> Bool)
```

Now QuickSpec will discover properties having e.g. `k1 /= k2` as a precondition. It will test these properties by generating *random* keys and discarding any test cases where `k1 = k2`.[1]

Before re-running QuickSpec, we need to make two small adjustments. Firstly, QuickSpec by default tests its conjectures on 1000 randomly-generated test cases. However, conditional properties can be hard to falsify, and 1000 tests is not always enough. We increase the number of tests to 10000, by adding the declaration `withMaxTests 10000` to the signature, e.g. below the `monoTypeWithVars` declarations.

Secondly, QuickSpec only discovers properties up to a given *size limit*. By default, properties with at most 7 symbols (variables and functions) on each side are discovered. However, when discovering conditional properties, each precondition adds 1 to the computed size. We therefore need to tell QuickSpec

[1] Sometimes, this generation strategy produces bad quality test data. When that happens, the user can provide their own test data generator by using the operator `predicateGen`, a variant of `predicate` which accepts a generator as an argument.

to search for slightly bigger properties. To increase the size limit to 8 symbols, we add the declaration `withMaxTermSize 8` to the signature.

Having done that, we re-run QuickSpec, and it produces the following 4 conditional conjectures, as well as 5 unconditional equations of size 8 which we do not reproduce here:

```
12. k /= k2 => lookup k (insert k2 v m) = lookup k m
13. k2 /= k => insert k v (insert k2 v2 m) = insert k2 v2 (insert k v m)
14. k /= k2 => lookup k (delete k2 m) = lookup k m
15. k /= k2 => insert k v (delete k2 m) = delete k2 (insert k v m)
```

Properties 12 and 14 show that the result of `lookup` is not affected by inserting or deleting an unrelated key. Properties 13 and 15 show that `insert` commutes with both itself and `delete`, when the two keys involved are different. (The fact that `delete` commutes with itself was found earlier as the unconditional property 8.)

In fact, these properties form a *complete specification* for the map operations. This is because we can use them to transform any particular sequence of calls of `insert` and `delete`, starting from `empty`, into a *normal form* where only `insert` is used, each key is inserted once and the keys are inserted in ascending order. For example:

```
  insert 3 'a' (delete 4 (insert 3 'b' (insert 2 'c' empty)))
= { property 15, twice }
  insert 3 'a' (insert 3 'b' (insert 2 'c' (delete 4 empty)))
= { property 6 }
  insert 3 'a' (insert 3 'b' (insert 2 'c' empty))
= { property 4 }
  insert 3 'a' (insert 2 'c' empty)
= { property 13 }
  insert 2 'c' (insert 3 'a' empty)
```

Any call to `lookup` for a map of this form can be rewritten to either a `Just` or `Nothing`. For example, if we apply `lookup 3` to the map above, we get `Just 'a'` by properties 12 and 2.

Note that QuickSpec is not able to see that the specification is complete. Rather, we checked this by hand. The approach is similar to that used in [5].

2.3 The Soundness of Music

In Chapter 20 of his Haskell textbook [15], Hudak designs a library of musical structures. The core of this library is the `Music` datatype below. It consists of primitive entities (notes and rests), operations to combine musical structures (parallel and sequential composition) and an operation to change the tempo of a piece of music. The full library includes operations to change the pitch or instrument of a piece of music, which we omit here.

```
data Music =
    Note Pitch Rational   -- Note p d plays note p for duration d
  | Rest Rational         -- Rest d is a rest of duration d
  | Music :+: Music       -- m1 :+: m2 plays m1 and m2 in sequence
  | Music :=: Music       -- m1 :=: m2 plays m1 and m2 simultaneously
  | Tempo Rational Music  -- Tempo x m plays m sped up by a factor of x
  | ...
```

For example, the expression `Note (C,4) 1` represents a middle C with a duration of one whole note. `Note (C,4) 1 :=: Note (E,4) 1 :=: Note (G,4) 1` represents a C major chord (C, E and G). Finally, `(Note (C,4) 1 :=: Note (E,4) 1 :=: Note (G,4) 1) :+: Rest 1 :+: Note (D,4) 1` represents a C major chord, followed by one whole note of silence, followed by a D. Hudak shows how to construct increasingly intricate pieces of music using these combinators.

In Chapter 21 of his book, Hudak defines and proves the algebraic properties of his musical structures. Can QuickSpec discover these properties automatically? It can—but getting it to do so required a good deal of thought. In this section, we report on our experiences using QuickSpec on the `Music` datatype. We start by showing what QuickSpec discovered, and then describe the problems we encountered.

The final output of QuickSpec is shown in Figure 5. It includes all of Hudak's axioms (they are properties 1, 2, 3, 7, 9, 11, 19, 20, 21, 22, 27 and 28), as well as other interesting and not-so-interesting properties. By studying QuickSpec's output, we can discover that:

- `Rest 0` is an identity for both `:+:` and `:=:` (properties 1, 2 and 9), and is unaffected by `Tempo` (property 20). Thus a zero-length rest represents an empty piece of music.
- `:+:` and `:=:` are associative (properties 3 and 11), and `:=:` is also commutative and idempotent (properties 7 and 8). `:+:` distributes over `:=:` on the left (property 15), but not on the right. (This is because, in the expression `(m :=: n) :+: o`, the piece `o` starts playing only once *both* `m` and `n` are finished.)
- Consecutive rests can be combined (property 4), as can consecutive zero-length notes of the same pitch (property 6). Parallel rests can be combined by taking the one with the longest duration (property 13). A rest played in parallel with a note of the same length can be ignored (property 14).

  Consecutive notes of the same pitch can *not* be combined if they have non-zero length. (Musically, repeating a note sounds different than playing it continuously.)
- Multiplying a piece's tempo by 1 has no effect (property 19). Multiplying the tempo by xy is the same as multiplying it by x and then by y (property 22). Multiplying the tempo of a note or rest by x reduces its duration by a factor of x (properties 24 and 26). Finally, `Tempo` distributes over `:+:` and `:=:` (properties 27 and 28).

  These properties constitute a complete specification of `Tempo`. This is because we can use them to *eliminate* any use of `Tempo` from a piece of music: first use properties 27 and 28 to push `Tempo` inwards until it reaches a note or a rest, then use properties 24 and 26 to remove it.

These properties reveal a good deal about how the `Music` datatype works, and include all of Hudak's handmade axioms. However, setting QuickSpec up to find them took some care. We now describe the steps we went through to find these properties.

Random Generator To use QuickSpec on a custom datatype such as `Music`, we must define an `Arbitrary` instance, which describes how to generate random values of that type. Our generator was defined in a standard way, and we do not discuss it here; see [4] for more information.

Observational Equivalence Hudak defines two musical objects to be equal if they sound the same when performed. Formally, the music library provides an operator

```
perform :: Context -> Music -> Performance
```

```
 Note :: (PitchClass, Int) -> NonNeg -> Music
 Rest :: NonNeg -> Music
(:+:) :: Music -> Music -> Music
  1. m :+: Rest 0 = m
  2. Rest 0 :+: m = m
  3. (m :+: n) :+: o = m :+: (n :+: o)
  4. Rest x :+: Rest y = Rest (x + y)
  5. Note p 0 :+: Note q 0 = Note q 0 :+: Note p 0
  6. Note p 0 :+: Note p 0 = Note p 0

(:=:) :: Music -> Music -> Music
  7. m :=: n = n :=: m
  8. m :=: m = m
  9. m :=: Rest 0 = m
 10. m :=: (m :+: n) = m :+: n
 11. (m :=: n) :=: o = m :=: (n :=: o)
 12. m :=: Note p 0 = Note p 0 :+: m
 13. Rest x :=: Rest y = Rest (max x y)
 14. Rest x :=: Note p x = Note p x
 15. (m :+: n) :=: (m :+: o) = m :+: (n :=: o)
 16. m :=: (n :+: (m :+: o)) = (m :=: (n :+: m)) :+: o
 17. m :=: ((m :=: n) :+: o) = (m :=: n) :+: o
 18. Rest x :=: (m :+: Rest x) = m :+: Rest x

Tempo :: Pos -> Music -> Music
 19. Tempo 1 m = m
 20. Tempo x (Rest 0) = Rest 0
 21. Tempo x (Tempo y m) = Tempo y (Tempo x m)
 22. Tempo (x * y) m = Tempo x (Tempo y m)
 23. Tempo x (Note p 0) = Note p 0
 24. Tempo x (Rest y) = Rest (y / x)
 25. Tempo x (Note p x) = Note p 1
 26. Tempo x (Note p y) = Note p (y / x)
 27. Tempo x m :+: Tempo x n = Tempo x (m :+: n)
 28. Tempo x m :=: Tempo x n = Tempo x (m :=: n)
 29. Tempo (x / y) (Note p 1) = Tempo x (Note p y)
 30. Rest x :+: Tempo y (Rest z) = Tempo y (Rest z) :+: Rest x
 31. Tempo x (Rest (y + x)) = Rest 1 :+: Tempo x (Rest y)
 32. Tempo x (Rest (max y x)) = Rest 1 :=: Tempo x (Rest y)
 33. Tempo x (m :+: Rest x) = Tempo x m :+: Rest 1
 34. Tempo x (Rest x :+: m) = Rest 1 :+: Tempo x m
 35. Tempo x (m :=: Rest x) = Rest 1 :=: Tempo x m
```

Figure 5: Complete QuickSpec output on the music example.

which transforms a piece of music into the specific sequence of notes that should be played, together with their timings. (Here, `Context` encodes various settings such as the speed of the performance.) Hudak then defines two pieces m_1 and m_2 to be equivalent if $\forall c.\, \texttt{perform}\ c\ m_1 = \texttt{perform}\ c\ m_2$.

This is an *observational equivalence*, where two objects are considered equal if they produce the same result in all contexts. By default, QuickSpec uses the built-in Haskell operator `==` for equality testing, which for `Music` means structural equality. However, we can also configure QuickSpec to use observational equality. To do so, we must first declare the type in the signature using the following syntax:

```
monoTypeObserveWithVars ["m"] (Proxy :: Proxy Music).
```

We then define our observational equality function by making an instance of the `Observe` type class:

```
instance Observe Context Performance Music where
  observe context m = perform context m
```

To test $m_1 = m_2$, QuickSpec will pick random `Contexts` c, and test that $\texttt{perform}\ c\ m_1 = \texttt{perform}\ c\ m_2$ holds for all c. In general, the first parameter to `Observe` is a *context* in which we can evaluate the object, and the second parameter is the *result* we get upon evaluating it.

Preconditions At this point, we ran QuickSpec, and were greeted with a curious *lack* of properties. There were *no properties at all* about `:+:`, not even associativity. Testing with QuickCheck revealed why not:

```
> quickCheck (\m1 m2 m3 -> m1 :+: (m2 :+: m3) =~= (m1 :+: m2) :+: m3)
*** Failed! Exception: 'Ratio has zero denominator' (after 4 tests)
```

It appears that we encountered a division by zero. The reason is that in `Tempo`, the speed argument must be a *positive* number—it is not allowed to multiply the speed by zero (or indeed by a negative number). Thus, although `Tempo` is declared as accepting any `Rational` argument, it has a hidden precondition. What's more, `Note` and `Rest` also have a precondition—their `Rational` argument must be non-negative.

We would like to tell QuickSpec to only apply `Tempo` to positive arguments, and `Note` and `Rest` to non-negative arguments. Unfortunately, this is not currently possible. Instead, we modified the music API to expose these preconditions in the types. We first defined types of positive and non-negative rationals:

```
newtype Pos = Pos Rational deriving (Eq, Ord, Show, Num, Fractional)
newtype NonNeg = NonNeg Rational deriving (Eq, Ord, Show, Num, Fractional)
```

Then we defined versions of `Tempo`, `Note` and `Rest` which take a `Pos` or `NonNeg` argument:

```
tempo :: Pos -> Music -> Music
note :: Pitch -> NonNeg -> Music
rest :: NonNeg -> Music

tempo (Pos x) m = Tempo x m
note p (NonNeg x) = Note p x
rest (NonNeg x) = Rest x
```

We used these in our signature instead of the original constructors.

Observational Equivalence Again When we re-ran QuickSpec, it indeed found that `:+:` is associative, along with many other properties. But something was still strange. Among the properties discovered were these two:

```
Rest x = Rest y
m :+: Rest x = m
```

The first states that all rests are equivalent, which must surely be wrong. The second states that adding a rest to the end of a piece of music has no effect. This sounds reasonable, as the extra rest cannot be heard. But it is wrong, because `(m :=: Rest x) :=: n` is not equivalent to `m :=: n`: in the first, there is a gap of `x` whole notes between `m` and `n`, while in the second, there is no gap.

The problem was that our *observational equivalence* was defined incorrectly. It is *not* true that two pieces are equivalent if they sound the same, because the pieces may end with different lengths of silence. To detect silence at the end of a piece, we modified our observational equivalence test so that it *adds an extra note at the end of the piece* before performing it. The resulting `Observe` instance looks as follows:

```
instance Observe Context Performance Music where
  observe context m = perform context (m :+: Note (C, 4) 1)
```

Success Finally, we re-ran QuickSpec and it worked correctly, producing the properties shown in Figure 5. Figure 6 shows the final signature we used. We note that:

- We declared the `Music` type as using observational equality.
- We included arithmetic operations for `Pos` and `NonNeg`. Rather than declaring these operations by hand, we used `arith`, a pre-made signature provided by QuickSpec which declares `0`, `1` and `+`. Since 0 is not positive, we excluded it for `Pos` by using the `without` combinator. (Another useful pre-made signature is `lists`, which declares some common list functions.)
- We included multiplication and division for tempos, and maximum for durations, as we guessed that they might satisfy interesting properties.
- To encode the fact that every positive number is non-negative, we added an injection function from `Pos` to `NonNeg`. We used `""` for the name of the function, which makes it invisible in properties.

Conclusion Getting good results from QuickSpec for the music library required some care. We had to define a notion of observational equality, and found that the obvious definition was wrong. We also had to tighten the API to avoid creating pieces with negative duration or infinite tempo. Once we did that, QuickSpec was able to produce high-quality conjectures for the music library, finding all of Hudak's axioms.

3 Combining Theorem Provers and Theory Exploration

QuickSpec was originally designed as a stand-alone tool for automatically discovering equational conjectures about functional programs written in Haskell, as shown above. In itself it only relies on testing, but it has been connected to several automated theorem provers and successfully used as a lemma discovery tool. The HipSpec system was designed for automating proofs by induction [6] and used QuickSpec to generate potential lemmas for the main conjecture given by the user. HipSpec then tried to prove these lemmas as well, and any lemma proved could be used in subsequent proof attempts of other lemmas and the main

```
musicSig = [
  monoType (Proxy :: Proxy NonNeg),
  monoType (Proxy :: Proxy Pos),
  monoTypeWithVars ["p", "q", "r"] (Proxy :: Proxy Pitch),
  monoTypeObserveWithVars ["m", "n", "o"] (Proxy :: Proxy Music),
  background [
    arith (Proxy :: Proxy NonNeg),
    arith (Proxy :: Proxy Pos) `without` ["0"],
    con "*" ((*) :: Pos -> Pos -> Pos),
    con "/" ((/) :: Pos -> Pos -> Pos),
    con "max" (max :: NonNeg -> NonNeg -> NonNeg),
    con "" (\(Pos x) -> NonNeg x)]
  series [sig1, sig2, sig3]]
  where
    sig1 =
      [con "Note" note,
       con "Rest" rest,
       con ":+:" (:+:)]
    sig2 = [con ":=:" (:=:)]
    sig3 = [con "Tempo" tempo]
```

Figure 6: A signature for the music library.

conjecture. This allowed improvements in automation beyond the previous state of the art. HipSpec was a light-weight prover: it simply applied (structural) induction to conjectures and sent off the resulting subgoals to an external automated theorem prover. In essence, HipSpec uses QuickSpec to construct a richer background theory for the inductive proofs, making them feasible to prove automatically. This is also useful in interactive proof assistants when beginning a new formalisation of some stucture about which few theorems have been proved: QuickSpec has been combined with Isabelle/HOL in the Hipster system [19, 17], and similarly also with HOL4 [24]. Here, conjectures proposed by QuickSpec can be proved automatically by custom tactics, or if unsuccessful, left open for the user to prove interactively.

3.1 Facilitating Interoperability: TIP and TIP-Tools

We have developed a suite of tools to make it easier to combine QuickSpec with various theorem provers and other systems (`http://tip-org.github.io`). These are based around a language format called TIP [7], which is a very slight extension of SMTLIB [1], and associated tools for converting problems and conjectures to and from TIP and other common formats for theorem provers [25], using the command line tool `tip`. We also provide a TIP front-end for calling QuickSpec, using the command line tool `tip-spec`. This makes it is possible to call QuickSpec on files given in TIP format, which is used in for example the Hipster system mentioned above. Hipster first translates from Isabelle to TIP, then passes the TIP file to QuickSpec (using the `tip-spec` tool) and finally translates the resulting conjectures back to Isabelle/HOL format. Note that when calling QuickSpec via the `tip-spec` front end, the generator functions, used to create random values for testing, are automatically derived, based on the datatypes. This means that the user has a little less flexibility than when using QuickSpec directly through Haskell. In the next section,

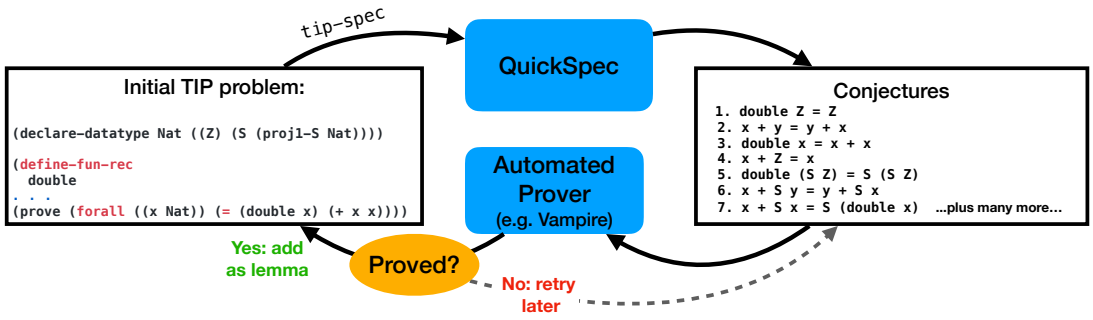

Figure 7: QuickSpec explores the functions and types in the original problem and produce a set of conjectures. Any lemmas proved may be used when finally trying the original conjecture.

we demonstrate how a new theorem prover can easily be combined with QuickSpec using the TIP-tools.

3.2 An Experiment: Combining QuickSpec and Vampire

How easy is it to combine theory exploration with an off-the-shelf theorem prover? The automated first-order prover Vampire has recently been extended to support proofs by induction [13], although with a focus on conjectures over the integers and with many repeated variables. Vampire does not support lemma discovery as such, but has techniques for generalising conjectures which can be useful if conjectures contain repeated variables or subterms. We wanted to test if we could combine QuickSpec and Vampire, in order to carry out a small experiment to demonstrate the importance of auxiliary lemmas for many inductive proofs, on a set of benchmarks somewhat different from those that Vampire's induction has previously been evaluated on. We use a very similar architecture for combining QuickSpec and a prover as used previously in e.g. HipSpec, shown in Figure 7.

For the experiment, we used Vampire 4.5.1 on a set of benchmarks from the TIP repository [7], specifically compiled for testing inductive provers on proofs about recursive datatypes, and often requiring auxiliary lemmas. These benchmarks were originally presented in [16]. The benchmark set contains 50 theorems and is available online in TIP format[2]. As the TIP benchmarks are written in a format allowing pattern matching in recursive function definitions, we first translate them to the SMT-LIB format accepted by Vampire, with axiomatised function definitions[3].

Without theory exploration, Vampire[4] proves 7 out of 50 conjectures, as shown in Table 1. Next, we ran QuickSpec as a pre-processing step. For each TIP problem file, QuickSpec explored the functions occurring in the problem to produce extra conjectures. Vampire was then given the conjectures to prove, and any it managed to prove were allowed to be used as lemmas in the proof of the main theorem (see Figure 7). This was all implemented in just a 90-line shell script. Vampire now proved an additional 18 benchmarks, shown in Table 2. Note that for three problems (properties 33–35) QuickSpec timed out during testing, as the functions (e.g. exponentiation recursively defined on natural numbers) produced very large test terms unless custom settings were used. In total Vampire proved 25 benchmarks. As a comparison, the same benchmarks were used to evaluate several inductive theorem provers in [6]; here HipSpec proved 44 problems (excluding the three needing custom settings), CLAM [16] was reported to

[2] `https://github.com/tip-org/benchmarks/tree/master/benchmarks/prod`

[3] Using the command: `tip ⟨filename⟩ --axiomatize-funcdefs2 --smtlib`

[4] Using the best-performing version from [13] with structural induction and generalisation turned on: `vampire_rel_4.5.1 -ind struct -indgen on`

have proved 41 fully automatically (and the rest interactively), and Zeno [27] proved 21. While Vampire's induction is not tailored specifically to these kinds of problems, its performance is still reasonable compared to dedicated inductive provers, when combined with QuickSpec for lemma discovery.

Prop.		Prop.	
2	*length (xs ++ ys) = length (ys ++ xs)*	43	*elem x ys ⇒ elem x (union zs ys)*
3	*length (xs ++ ys) = (length ys) + (length xs)*	45	*elem x (insert x ys)*
15	*x + (Suc x) = Suc (x + x)*	46	*x = y ⇒ elem x (insert y zs)*
37	*(elem x zs) ⇒ elem x (ys ++ zs)*		

Table 1: Properties proved by Vampire (without extra lemmas from QuickSpec).

Prop.		Prop.	
1	*double x = x + x*	18	*rev ((rev xs)++ ys) = (rev ys) ++ xs*
4	*length (xs ++ xs) = double (length xs)*	19	*(rev (rev xs)) ++ ys = rev (rev (xs ++ ys))*
5	*length (rev xs) = length xs*	20	*even (length (xs ++ xs))*
6	*length (rev (xs ++ ys)) = (length xs) + (length ys)*	22	*even (length (xs ++ ys)) = even (length (ys ++ xs))*
10	*rev (rev xs) = xs*	23	*half (length (xs ++ ys)) = half (length (ys ++ xs))*
11	*rev (rev xs ++ rev ys) = ys ++ xs*	24	*even (x + y) = even (y + x)*
13	*half (x + x) = x*	25	*even (length (xs ++ ys)) = even ((length ys) + (length xs))*
16	*even (x + x)*	26	*half (x + y) = half (y + x)*
17	*rev (rev (xs ++ ys)) = (rev (rev xs)) ++ (rev (rev ys))*	30	*rev ((rev xs) ++ []) = xs*

Table 2: Additional properties proved by Vampire with help of lemmas from QuickSpec.

4 Related Work in Theory Exploration and Automated Conjecturing

Conjecture generation systems tend to roughly fall into three categories: heuristic rule-based systems, term generation-and-testing (to which QuickSpec belongs) and neural network-based systems.

Rule-based and Heuristic The AM system is probably the first system designed to discover mathematical concepts [20]. It relied on several hundred heuristic rules to generate mathematical concepts and conjectures, with each rule contributing a pre-defined "interestingness score" to the result. The HR system had a similar goal of automating concept formation and conjecturing [8]. It took a set of axioms as inputs, and combined model finding with heuristic production rules to invent new conjectures of interest. The MATHsAiD system was specifically designed as a theory exploration system for mathematicians [21]. The discovery process was guided by a forward reasoning process, which instantiated templates called *theorem shells*, following heuristic plans constructed to reflect human mathematical reasoning.

In comparison, QuickSpec is a more light-weight system concerned only with conjecturing, not any other concept formation tasks. It does not employ explicit heuristic rules: it simply generates terms about the given concepts, and evaluates them on random values to learn which ones appear to be equal. It also

uses a very simple notion of interestingness: a term is interesting to explore if it is non-reducible with respect to what is known so far. This makes it more flexible, as it can be used to explore conjectures about any datatype for which it can generate examples.

Term Generation and Testing Graffiti was a conjecture generation system specifically for graph theory [12]. It maintained a library of example graphs, and attempted to generate conjectures of certain shapes, then checking them for consistency with its library examples. Unlike QuickSpec, where new examples can be generated on demand, Graffiti's set of examples was fixed, leading to production of relatively many non-theorems. The systems IsaCoSy [18] and IsaScheme [22] are conceptually similar to QuickSpec, relying on term generation with different restrictions to avoid trivial and redundant conjectures, combined with random testing. A key difference is however that QuickSpec evaluates terms (and records the results), not whole equational statements, which greatly improves efficiency. Another closely related system from the functional programming community is Speculate [3], which like QuickSpec, was designed to discover properties about Haskell programs.

Neural Networks Recently, techniques using large neural network architectures from natural language processing have been applied to the problem of discovering new conjectures. Urban et al. train the transformer model GPT-2 on the Mizar Mathematical Library [28]. With the right parameters, they get GPT-2 to generate novel and well-typed conjectures in Mizar format. However, the output may include duplicates from the library as well as non-theorems. Rabe et al. present a system based on self-supervised language models for mathematical reasoning tasks, including generating a missing precondition for a given conditional statement, generating one side of an equation given the other side, as well as "free-form" conjecturing [23]. Between 13–30% of generated statements were both provable and new, with the remainder being for instance exact copies or alpha-renamings of statements from the training set, or simply false.

The neural network approach is of course radically different to QuickSpec. QuickSpec was designed to produce relatively small sets of interesting equations at a time, intended to be read by a human as a specification for a functional program. It rarely generates any false or duplicate conjectures, thanks to the integrated testing in the equation formation. Furthermore, QuickSpec is not dependent on training data about the theory at hand being available, but does on the other hand require generators to produce test data, with the caveat that the test data needs to be computable, or that a computable proxy can be used.

5 Conclusion and Further Work

The idea behind QuickSpec is simple: generate interesting terms and evaluate them on random test data to see which ones appear to be equal. Despite this, it is fast and efficient when given relatively small theories to explore. In this sense, QuickSpec is data-driven: it uses generator functions from the automated testing framework QuickCheck to generate and evaluate as many ground examples as it needs. This is also one of its limitations: trying to evaluate functions of high complexity (e.g. exponentiation in Peano arithmetic) can drastically slow down testing if a large test case is generated. Similarly, co-recursive functions can lead to non-terminating values [10]. One solution is to extend QuickSpec with observational equivalence checking, similarly as shown in the Music example in section 2.3. The other main weakness of QuickSpec is scalability. As it generates all non-redundant terms, the search space eventually becomes intractable when faced with large theories (say, a library with 30+ functions) or when asked to explore very large terms (sizes over about 9). Here, a combination with machine learning could be fruitful. A human user

can immediately judge which of QuickSpec's equations are "nice and sensible" or, on occasion, "weird and ugly" and thus not very interesting. Certain shapes of conjectures are often both more useful and aesthetically pleasing, and QuickSpec could be steered towards that part of the search space first. A recent extension to QuickSpec experiments with doing this, using user-provided templates when exploring larger theories and terms [11]. A natural next step is to instead attempt to learn templates from a library of lemmas, as suggested in [14], thus creating a hybrid system where machine learning steers the term generation towards interesting parts of the search space. Learned templates may also facilitate search of non-equational conjectures, while still managing search space size.

References

[1] Clark Barrett, Pascal Fontaine & Cesare Tinelli (2017): *The SMT-LIB Standard: Version 2.6*. Technical Report, Department of Computer Science, The University of Iowa. Available at `www.SMT-LIB.org`.

[2] R. S. Boyer & J S. Moore (1979): *A Computational Logic*. Academic Press.

[3] Rudy Braquehais & Colin Runciman (2017): *Speculate: discovering conditional equations and inequalities about black-box functions by reasoning from test results*. In: *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*, pp. 40–51, doi:10.1145/3156695.3122961.

[4] Koen Claessen & John Hughes (2000): *QuickCheck: a lightweight tool for random testing of Haskell programs*. In: *Proceedings of ICFP*, pp. 268–279, doi:10.1145/351240.351266.

[5] Koen Claessen & John Hughes (2002): *Testing Monadic Code with QuickCheck*. *SIGPLAN Notices* 37(12), p. 47–59, doi:10.1145/636517.636527.

[6] Koen Claessen, Moa Johansson, Dan Rosén & Nicholas Smallbone (2013): *Automating Inductive Proofs using Theory Exploration*. In: *Proceedings of the Conference on Automated Deduction (CADE)*, *LNCS* 7898, Springer, pp. 392–406, doi:10.1007/978-3-642-38574-2_27.

[7] Koen Claessen, Moa Johansson, Dan Rosén & Nicholas Smallbone (2015): *TIP: Tons of Inductive Problems*. In: *Conference on Intelligent Computer Mathematics*, *LNCS* 9150, Springer, pp. 333–337, doi:10.1007/978-3-319-20615-8_23.

[8] Simon Colton (2002): *The HR Program for Theorem Generation*. In Andrei Voronkov, editor: *Automated Deduction—CADE-18*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 285–289, doi:10.1007/3-540-45620-1_24.

[9] Lucas Dixon & Moa Johansson (2007): *IsaPlanner 2: A Proof Planner for Isabelle*. DReaM Technical Report (System description).

[10] Sólrún Halla Einarsdóttir, Moa Johansson & Johannes Åman Pohjola (2018): *Into the Infinite - Theory Exploration for Coinduction*. In: *Proceedings of AISC 2018*, pp. 70–86, doi:10.1007/978-3-319-99957-9_5.

[11] Sólrún Halla Einarsdóttir, Nicholas Smallbone & Moa Johansson (2021): *Template-based Theory Exploration: Discovering Properties of Functional Programs by Testing*. Post-proceedings of IFL'20.

[12] Siemion Fajtlowicz (1988): *On conjectures of Graffiti*. *Annals of Discrete Mathematics* 38, pp. 113–118, doi:10.1016/S0167-5060(08)70776-3.

[13] M. Hajdu, P. Hozzova, L. Kovacs, J. Schoisswohl & A. Voronkov (2020): *Induction with Generalization in Superposition Reasoning*. In: *Intelligent Computer Mathematics. CICM 2020*, *LNCS* 12236, Springer, doi:10.1007/978-3-030-53518-6_8.

[14] Jonathan Heras, Ekaterina Komendantskaya, Moa Johansson & Ewen Maclean (2013): *Proof-Pattern Recognition and Lemma Discovery in ACL2*. In: *Proceedings of LPAR*, doi:10.1007/978-3-642-45221-5_27.

[15] Paul Hudak (200): *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, doi:10.1017/CBO9780511818073.

[16] Andrew Ireland & Alan Bundy (1996): *Productive Use of Failure in Inductive Proof*. Journal of Automated Reasoning 16, pp. 79–111, doi:10.1007/BF00244460.

[17] Moa Johansson (2017): *Automated theory exploration for interactive theorem proving: An introduction to the Hipster system*. In: Proceedings of ITP, LNCS 10499, Springer, pp. 1–11, doi:10.1007/978-3-319-66107-0_1.

[18] Moa Johansson, Lucas Dixon & Alan Bundy (2011): *Conjecture Synthesis for Inductive Theories*. Journal of Automated Reasoning 47(3), pp. 251–289, doi:10.1007/s10817-010-9193-y.

[19] Moa Johansson, Dan Rosén, Nicholas Smallbone & Koen Claessen (2014): *Hipster: Integrating Theory Exploration in a Proof Assistant*. In: Proceedings of CICM, Springer, pp. 108–122, doi:10.1007/978-3-319-08434-3_9.

[20] Douglas B. Lenat (1976): *AM, an artificial intelligence approach to discovery in mathematics as heuristic search*. Ph.D. thesis, Stanford University.

[21] R. L. McCasland, A. Bundy & P. F. Smith (2017): *MATHsAiD: Automated mathematical theory exploration*. Applied Intelligence, doi:10.1007/s10489-017-0954-8.

[22] Omar Montano-Rivas, Roy McCasland, Lucas Dixon & Alan Bundy (2012): *Scheme-based theorem discovery and concept invention*. Expert systems with applications 39(2), pp. 1637–1646, doi:10.1016/j.eswa.2011.06.055. Available at `https://hdl.handle.net/1842/6269`.

[23] Markus N. Rabe, Dennis Lee, Kshitij Bansal & Christian Szegedy (2021): *Mathematical Reasoning via Self-supervised Skip-tree Training*. Accepted for ICLR 2021, to appear.

[24] Juan Ricart (2019): *Hopster: Automated discovery of mathematical properties in HOL*. Master's thesis, Chalmers University of Technology. Available at `https://hdl.handle.net/20.500.12380/300420`.

[25] Dan Rosén & Nicholas Smallbone (2015): *TIP: Tools for Inductive Provers*. In: Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR), LNCS 9450, Springer, pp. 219–232, doi:10.1007/978-3-662-48899-7_16.

[26] Nicholas Smallbone, Moa Johansson, Koen Claessen & Maximilian Algehed (2017): *Quick specifications for the busy programmer*. Journal of Functional Programming 27, doi:10.1017/S0956796817000090.

[27] Willam Sonnex, Sophia Drossopoulou & Susan Eisenbach (2012): *Zeno: An automated prover for properties of recursive datatypes*. In: Proceedings of TACAS, Springer, pp. 407–421, doi:10.1007/978-3-642-28756-5_28.

[28] Josef Urban & Jan Jakubův (2020): *First Neural Conjecturing Datasets and Experiments*. In: Proceedings of CICM, doi:10.1007/978-3-030-53518-6_24.