Live Programming By Example

ANONYMOUS AUTHOR(S)

Programming by example is a powerful approach progam synthesis and has enjoyed particular success as an aid to novice programmers. While the theortical foundations of programming by example continue to expand the potential for synthesis, it remains a special use case tool. Aside from the FlashFill tool embedded within Excel spreadsheets, programming by example is not used by everyday programmers. This is in part due to the lack of a native interface to support this new paradigm of programming. We propose using live programming to help novice programmers better understand examples as a mode of programming.

Additional Key Words and Phrases: Programming By Example, Live Programming

ACM Reference format:
Anonymous Author(s). 2017. Live Programming By Example. *Proc. ACM Program. Lang.* 1, 1, Article 1 (January 2017), 7 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

```
f :: Int ⇒ Int
f x =
  let y = 7
  in
    x+y
```

Over the years, the lives of programmers have been made dramatically easier with the continuous development of excellent programming tools. In the past decade, domains like deployment and databases have been revolutionized by tools like container-based infrastructure and NoSQL databases. In spite of this, the way programmers actually develop software on a daily basis has been relatively stagnant. The process is still writing code, compiling it, and then finally testing it. Since actually testing code is steps removed from writing it, programmers are constantly simulating the execution of their code in their head and only occasionally testing it. If the program does not work as expected, then the programmer will have to divert his attention to debugging before continuing to write code. The more code the programmer writes before testing, the more code he or she will have to sift through when debugging.

Live programming seeks to improve this by making the process less sequential, thereby increasing the speed of development iteration. This is often done by executing the code on a number of examples as it is being modified, thus the programmer immediately knows how his or her changes are affecting the software being developed. With this information, the programmer can instantly identify when the computer's execution diverges from his or her mental simulation and respond accordingly. This prospect of improved development efficiency has drawn institutions such as Microsoft Research to investigating this new type of programming environment [6].

2017. 2475-1421/2017/1-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Professor Piskac's group at Yale is exploring an extension of live programming by developing a cooperative programming environment in which the code-example interaction is bidirectional. This means that not only will changes to the code be reflected in the examples, but changes to the examples will be reflected in the code. By changing the output of an example, the programmer can trigger a code synthesis engine to modify his or her program to output the correct result. This is a difficult feat to achieve and incorporates a variety of components including generating examples, error localization, error repair, code synthesis, tool development, and editor integration. This paper concerns the first piece of this complex system, example generation.

The group's implementation of a cooperative programming environment targets Haskell, a modern, functional programming language. While Haskell is not as widespread as established imperative languages like Java or Python, it is rapidly increasing in popularity and can serve as an excellent introduction to functional programming or programming in general. Due to the lack of side effects and immutable data structures, functional languages have recently made a resurgence in response to the growing need for concurrency when scaling applications. A cooperative programming environment for Haskell will lower the barrier to entry and promote the learning and usage of functional languages.

1.1 Problem

Examples are the bread and butter of cooperative programming. They are both the source of feedback for the programmer as well as the initiator of code synthesis. As a result, the quality of examples directly influences the quality of feedback the programmer receives as well as the potential avenues for code synthesis the programmer can exploit.

Given that the purpose of cooperative programming is to serve the programmer, it is imperative that the examples generated are human processable. This idea of being human processable is two-fold. First, the environment should not inundate the programmer with thousands of random, potentially redundant examples, but should present a small set of representative examples that is just large enough to capture the behavior of the program. If the programmer cannot get the feedback he or she needs at a glance, then the development process once again becomes segmented instead of continuous. Second, the examples should stay as consistent as possible even as the code evolves. Even if the set of generated examples is representative for each iteration of the code, if the values of the arguments in the examples changes between iterations, then the programmer will have to mentally reevaluate the examples every time to see which section of the code each one reflects. Therefore, example reuse is critical. Other challenges relevant to example generation are more specific to the target language, Haskell. Since Haskell is a functional language, Haskell programmers often write higher-order functions to simplify common operations. To test these functions, the examples will have to include first-order functions as arguments. In addition, Haskell has the notion of type variables and type classes that allow for polymorphic functions with contexts that restrict type variables to be of certain type classes. This poses a difficulty as examples are always composed of concrete types. Because these features are used ubiquitously in Haskell development, example generation in a cooperative programming environment for Haskell should support them.

2 APPROACH

This project creates a tool called Exampler that takes as input a Haskell source file and the name of a function within that source file. Exampler will generate a set of representative

examples for the specified function and watch the source file for changes to know when to update the set of examples.

We define a set of representative examples as a set of examples that provides significant or complete branch coverage of the specified function. The program attempts to generate such a set by repeatedly testing the function on random examples and then observing differences in branch coverage. If an example improves branch coverage, then it is kept as part of the representative set. Given enough random examples, it is possible to get full branch coverage of all reachable branches in a program. Since only examples that improve branch coverage are included in the set, we have also constructed an example set that is small enough for the programmer to understand.

As for example reuse, it appears that it would be as simple as keeping the generated examples in memory. The issue is that Haskell is a compiled language, so the program would need to be recompiled whenever the source file of interest is modified. The obvious solution is to write the examples to a file, but this does not work for examples for higher-order functions since first-order functions have no textual representation in the compiled program. The only way to deal with this is to dynamically reload the module as it changes so the program can keep all of the examples in memory.

Concerning polymorphic functions, the program inspects the type structure of the function of interest and builds a new type by replacing the type variables with concrete types that are valid given the function's context. The concrete type can then be used to generate examples for the polymorphic function.

All of these features result in a robust tool that can generate sets of representative examples for a function and maintain them as the function changes. Furthermore, the program is just a simple executable wrapper around a reusable module that can be leveraged in any cooperative programming or live programming environment.

3 INTRODUCTION

3.1 Generating an Example

An example is just a set of arguments and the result of executing the function with those arguments. Since Haskell is a statically-typed language, an example generator needs to be able to generate examples in a type-safe manner. The example generating function takes arbitrary functions as arguments, so it only knows that the function matches the polymorphic type for all functions, `a ⇒ b`. This means that the example generator cannot generate random values for all arguments at once as it only knows the type of the first argument, `a`. Exampler solves this by generating a random value for the first argument and then currying the function with that value to produce a new function `b ⇒ c`. This process is repeated until the function is fully evaluated and a full set of arguments is accumulated as well as a result.

The random values are generated via QuickCheck. QuickCheck defines a type class, Arbitrary, and instances of Arbitrary have a function, arbitrary, that generates a random value of that type. If a programmer wants to use Exampler (or QuickCheck) for custom data types as arguments, then he or she will have to define an Arbitrary instance for them so the program can generate random values for that type.

Another type-related difficulty is representing the resulting example since the arguments and result can be of any type, but the example generator can only have one return type. QuickCheck gets around this by requiring arguments and results to be instances of the Show type class so they can be represented as strings, but this is not compatible with supporting

higher-order functions since arbitrary functions cannot be represented as a string without some loss of information. If Exampler were to use this method, then it could not reuse any examples that have functions as arguments.

To handle the case of functions as examples, Exampler uses the Existential Quantification language extension for Haskell that allows types that share a common type class to be treated as a single type. Since all arguments are required to be instances of Arbitrary and all result types are automatically instances of Exampleable (a type class defined by Exampler), an example can be represented as a list of AnyArbitrary and the result can be represented as AnyExampleable (AnyArbitrary and AnyExampleable are data types that wrap any instance of the Arbitrary or Exampleable type classes, respectively). When the example is recycled, the program can unwrap the arguments and do a type-safe cast to their original types during evaluation.

3.2 Generating an Example for a Polymorphic Function

Since instances of type classes in Haskell are usually defined for concrete types, the example generator described in the previous section cannot support polymorphic functions out of the box. Instead of requiring the programmer to add the Arbitrary and Exampleable context restrictions to his or her polymorphic functions, Exampler automatically creates a concrete version of the function with appropriate concrete types.

Exampler accomplishes this by using Template Haskell, a language extension that allows for compile-time metaprogramming through abstract syntax tree manipulation. Given the name of a function, Exampler uses Template Haskell to look up its type represented as an abstract syntax tree. The program then traverses that abstract syntax tree and replaces type and type constructor variables with concrete type and type constructors, respectively. This process is complicated by polymorphic functions that have contexts that restrict type and type constructor variables to specific type classes. Before traversing the abstract syntax tree, the program processes these constraints to build a data structure that maps the names of the variables to a set of concrete types or type constructors that adhere to the constraints. This map is then used in the variable replacement step. The set of possible types is returned in an unmeaningful order, so there is also a list of of default preferred types and type constructors that will be chosen if possible (e.g., Int for types, List for type constructors), otherwise the first valid type is chosen.

This method allows the program to generate examples for any polymorphic function. Moreover, since Template Haskell generates code at compile-time, type safety is guaranteed.

3.3 Generating a Representative Set of Examples

The next step after generating examples is to identify a representative set of examples. As mentioned earlier, we consider a representative set to be one that has adequate branch coverage. Haskell has a convenient tool called Haskell Program Coverage (HPC) [4] that tracks program coverage by instrumenting programs at compile-time. This tool also provides a Haskell library that allows one to get program coverage while the program is executing instead of only after the program exits. After each example is generated and executed, the program writes the coverage data to a temporary file and runs HPC to generate a report. This report is compared to the one for the previous iteration to see if the example covered any previously uncovered branches. If it does, then it is shown to the programmer and is kept for future reuse. When HPC reports full branch coverage, then Exampler will have generated a set of representative examples.

3.4 Handling Code Updates

The value of Exampler is that it will continually update the example set as the code changes. As described in the approach section, for example reuse to work on higher-order functions, the program cannot exit and recompile each time the source file of interest is updated. The solution is to dynamically load the file via the GHC (Glasgow Haskell Compiler) API.

One ramification of dynamically loading modules is that accessing functions in the dynamically loaded module is inherently unsafe because there is no way to know the types of these functions when the main program is compiled. To combat this, Exampler will first generate a program that will import the modified module and print the updated type of the function of interest. Exampler can then read the output of that program and manually check the types. Besides adding type safety, this also enables Exampler to detect when the type signature of the function changes so it can respond by recompiling and restarting. Additionally, this step requires the modified module to compile successfully so Exampler can catch compilation errors early and notify the programmer.

Exampler also does not import the actual source file but makes a temporary copy instead. This is necessary because HPC reports coverage by module name, so Exampler assigns a unique module name to each copy. It also allows Exampler to automatically export the function in the duplicate module so the programmer does not have to worry about visibility.

3.5 Generating the Program

The final issue to overcome also deals with type safety. In order for Exampler to safely build a concrete type for the function of interest, it needs to know the type of the function at compile-time. Unfortunately, Exampler will not know the source file or function until run-time. To remedy this, Exampler actually generates a program that does the example generation for the provided file and function when the programmer runs it. This generated program communicates with the main program through pipes. It will notify the main program when the type signature of the function changes so that the main program can recompile the generated program. The full system architecture is diagrammed below.

4 RELATED WORK

The concept of live programming has been used to build tools for continuous feedback in a number of languages, but these have mostly been for visual languages such as Morphic [5] and PureData [7], or languages designed specifically for live programming such as YinYang [6]. While some of these tools like YinYang are able to provide more information about the program being executed through built-in language features, the language itself is not commonly used. The overarching cooperative programming project that Exampler is a part of aims to support Haskell, a language prevalent throughout academia and industry.

Specific to Haskell, there is a tool called QuickCheck [2] that tries to check specific properties of a function by testing if the property holds for a large volume of random examples. While this is useful for testing once the code is written, it generates a massive number of examples and is not easily processed and understood by the programmer. Even so, QuickCheck is still quite useful and some of its modules are leveraged by Exampler to generate random examples.

Finally, there are many tools and IDEs that will automatically run tests for a program as the programmer updates it. These include Guard for Ruby and NCrunch for the Visual Studio IDE. While these are certainly handy, they require the programmer to write tests beforehand instead of giving automatic feedback. Also, since the tests are manually entered,

this approach does not work when there are major structural changes (i.e., type signature changes) to the code base.

5 FUTURE WORK

Besides implementing the other components of the cooperative programming environment, there is much work to be done to improve Exampler. Because the project is a critical component of a larger system, the goal was to support as much of Haskell as quickly as possible so that the other components could have access to an example generator. Consequently, the individual aspects of Exampler are somewhat naive.

One such aspect is generating a representative set of examples. The current approach is entirely random. While using a slew of random examples should always be able to find a representative set of examples eventually, the usefulness of these examples decreases as the time it takes to find them increases. If the feedback does not feel immediate anymore, then the programmer will not be able to rely on the tool. A more structured way to find a set of representative examples is through concolic execution [8]. Concolic is a portmanteau of concrete and symbolic execution which involves trying a random concrete input while following the execution symbolically to obtain the constraints along that execution path. The last constraint is then negated and sent, along with the other constraints, to a solver to find a new input that will follow a different execution path. This process then continues until input examples are generated for all paths or all paths up to a certain depth. Concolic execution would likely perform faster and more consistently in most cases, but the implementation requires a symbolic execution engine and none currently exist for Haskell. A possible alternative is to use a symbolic execution engine that targets an intermediate language used by the Haskell compiler such as C or LLVM (e.g., Cute [8] or KLEE [1], respectively). Concolic execution has been implemented for another functional language, Erlang, as a tool called CutEr [3], which could serve as a model for a Haskell implementation.

A second area for improvement concerns the interpretation of a representative set of examples. Just providing one example for each branch of a function may not provide adequate information to the programmer. For example, if the programmer is unaware of some edge cases and does not distinguish them from another branch, then having one example from that branch might not identify those edge cases, and the programmer will assume that his or her code is correct. While providing more examples can mitigate this issue, it also risks providing too much information for the programmer to quickly process and understand. Another solution could be to ask the programmer to specify certain properties that the function should satisfy in order to identify when that property does not hold, as is done in QuickCheck, but this requires more input from the programmer and disrupts the development process. These advantages and drawbacks suggest that there is a tradeoff between ease-of-use and usefulness, and there is much research to be done to determine where the optimum lies.

6 CONCLUSION

Exampler is a robust tool that integrates random examples with test coverage to provide a representative set of examples for a specified function. By dynamically reloading code as it is being modified and using metaprogramming to manipulate types, it supports all kinds of functions including higher-order and polymorphic functions. As it is meant to be a programming tool to speed up development, Exampler is extremely easy to use. To use the tool, the programmer just needs to run Exampler <path to source file> <name of function within source file> , and the program will maintain a set of representative

examples, notify the programmer of compilation errors, and recompile if the type signature of the function changes, all without any additional input from the programmer.

Having a constantly updating set of representative examples while coding benefits novice and veteran programmers alike. This immediate feedback allows veteran programmers to improve their development iteration speed and helps novice programmers draw connections between their actions and the program's output. Moreover, representative examples capture the gist of a program and can be used to help programmers comprehend obfuscated or poorly maintained code.

While Exampler is a convenient tool on it's own, it's main purpose is to be integrated into a cooperative programming environment for Haskell. Once implemented, this programming environment will allow programmers to manipulate the examples to synthesize updated code, essentially automating debugging. That being said, there is much work to be done and other members of Professor Piskac's group are hard at work on the code synthesis engine.