MagicHaskeller: System demonstration

Susumu Katayama

University of Miyazaki
1-1 W. Gakuenkibanadai, Miyazaki, Miyazaki 889-2192, Japan
skata@cs.miyazaki-u.ac.jp

Abstract. This short paper introduces the usage and behavior of MagicHaskeller , which is one of the representative inductive functional programming systems. Although MagicHaskeller had been a generate-and-test method based on systematic exhaustive search, an analytical synthesis engine was added to its recent versions, which enables a new method that generates many programs analytically from the given insufficient set of input-output examples and tests those programs with a separately given predicate. This paper mentions both engines.

1 Overview

MagicHaskeller [Katayama(2005b)] is an inductive functional programming system based on systematic exhaustive search. *Inductive functional programming (IFP)* is a form of programming automation, where recursive functional programs are synthesized through generalization from the ambiguous specification usually given as a set of input-output pairs. Currently, there are two approaches to IFP: *analytical approach* that synthesize programs by looking into the input-output pairs and conducting inductive inference, and *generate-and-test approach* that generates many programs and picks up those that satisfy the specification. MagicHaskeller has been playing the representative role as the generate-and-test method based on systematic exhaustive search since the first binary release in 2005. Since its Version 0.8.6 release, analytical search algorithm has been added, and a new approach that could be called *analytically-generate-and-test approach* has been made possible, where the analytical synthesizer generates many programs from the given insufficient set of input-output examples and picks up those that satisfy the predicate separately given as a part of the specification. In this short demonstration paper, we present how to use both synthesis engines and the results from use of them.

2 Building and installation

MagicHaskeller has been developed in Haskell and its recent versions are released as a library. In order to build and install its copy, first you need to install Version 6.10.* or 6.12.* of Glasgow Haskell Compiler (GHC). Although Version 0.8.6.1 of MagicHaskeller can be build with Version 7 of GHC, its analytically-generate-and-test functionality does not work with this version of GHC.

Also, in order to ease the installation process you should have the Cabal [Jones(2005)] package that is the standard framework for distributing HASKELL programs. In addition, if cabal-install package is installed, simply typing

```
cabal update
cabal install MagicHaskeller
```

builds and installs the MAGICHASKELLER system into the user's home directory.

Implemented as a library, the typical usage of MAGICHASKELLER is to use it within Glasgow Haskell Compiler interactive (GHCi), like **QuickCheck** [Claessen and Hughes(2000)]. This is achieved by invoking GHCi with `-package MagicHaskeller` option. The language extension with Template Haskell [Sheard and Peyton Jones(2002)] is also necessary if you want to do various things by using its oxford bracket syntax. Thus, MAGICHASKELLER is usually invoked with

```
ghci -package MagicHaskeller -XTemplateHaskell
```

In this paper, when quoting use of the interactive system, we always supply the `-v0` option which means verbosity level 0, in order to avoid clutter.

3 The modules for exhaustive search

The systematic exhaustive search modules define functions that generate all the programs (up to semantical equivalence) which can be constructed using the given primitive set with function applications and lambda abstractions, as an infinite stream[Katayama(2005a)]. They also define functions for testing the generated programs to leave only the programs that satisfy the given specification. The algorithm used for generating the stream of programs effectively enumerates all the proofs for the proposition corresponding to the given type under Curry-Howard isomorphism, based on sequent calculus.[Katayama(2010)]

The greatest feature of these modules is that they can synthesize programs by only selecting the primitive set and writing the specification in the form of a predicate. The specification need not be a set of input-output pairs. The type of the function has to be notified to the algorithm, but the user need not give it explicitly. It can be inferred from the specification given as a predicate.

To the end of this section, we exemplify the usage of the systematic exhaustive search engine. More examples can be found in [Katayama(2006)], though it describes an older version of MAGICHASKELLER .

3.1 A simple example

This is an example of having MAGICHASKELLER synthesize functions that takes `"abc"` and returns `"aabbcc"`:

```
$ ghci -package MagicHaskeller -XTemplateHaskell -v0
Prelude> :m +MagicHaskeller.LibTH
Prelude MagicHaskeller.LibTH> init075
Prelude MagicHaskeller.LibTH> printAll $ \f -> f "abc" == "aabbcc"
\a -> list_para a [] (\b _ d -> b : (b : d))
\a -> list_para a (\_ -> []) (\b _ d _ -> b : (b : d 0)) 0
\a -> list_para a (\_ -> []) (\b _ d _ -> b : (b : d 0)) 0
\a -> list_para a (\_ -> []) (\b _ d _ -> b : (b : d True)) True
\a -> list_para a (\_ -> []) (\b _ d _ -> b : (b : d True)) False
\a -> list_para a (\_ -> []) (\b _ d _ -> b : (b : d False)) True
\a -> list_para a (\_ -> []) (\b _ d _ -> b : (b : d False)) False
\a -> list_para a (\_ -> []) (\b _ d _ -> b : (b : d [])) []
\a -> list_para a (\_ -> []) (\b _ d _ -> b : (b : d [])) a
\a -> list_para a (\_ -> []) (\b c d _ -> b : (b : d c)) a
\a -> list_para a (\_ -> []) (\b c d _ -> b : (b : d c)) []
\a -> list_para a (\_ -> []) (\b _ d _ -> b : (b : d Nothing)) Nothing
\a -> list_para a (\b -> b) (\b _ d e -> b : (b : d e)) []
\a -> list_para a (\b -> b) (\b c d _ -> b : (b : d c)) a
\a -> list_para a (\b -> b) (\b c d _ -> b : (b : d c)) []
\a -> list_para a (\b -> b) (\b _ d _ -> b : (b : d [])) a
\a -> list_para a (\b -> b) (\b _ d _ -> b : (b : d [])) []
^CInterrupted.
Prelude MagicHaskeller.LibTH> printAllF $ \f -> f "abc" == "aabbcc"
\a -> list_para a [] (\b _ d -> b : (b : d))
^CInterrupted.
```

The text regions between `Prelude` and `>` are prompts of GHCi. The first line after the GHCi invocation is for bringing module `MagicHaskeller.LibTH` into scope. The second line initializes the environment and set the component library, i.e. the set of combinators with which to construct programs, with a recommended set of combinators. The third line requests to print all the expressions which can be synthesized using the combinators in the component library that satisfy the predicate `\f -> f "abc" == "aabbcc"`.

Then, synthesized programs are printed line by line, from the smallest one with the least number of function applications, increasing the program size progressively. The `list_para` function is (a function isomorphic to) the list paramorphism (e.g. [Augusteijn(1999)]) and defined in the module `MagicHaskeller.LibTH`. Other notations are the same as HASKELL 's — `\v -> e` means $\lambda v.e$, `[]` denotes the empty list, and `(:)` is the binary constructor that adds one element to the first position of a list. Because the algorithm generates an infinite stream of programs, it has to be interrupted on the way.

The letter `F` in `printAllF` means filtering out expressions that are semantically equivalent to any of the already printed expressions by using a randomized algorithm.[Katayama(2008)] This is useful, though it affects the efficiency. In the above example, seemingly all the synthesized and printed programs are equivalent to the firstly generated one. However, there is always the possibility where some of them are proved to be different and printed later while using `printAllF` if the computation is not interrupted.

3.2 An example of using a rich library

A variant of the filter for removing semantically equivalent expressions can be applied during program generation rather than after program generation, in order to reduce the number of programs and quicken the synthesis. Because the filtration itself takes time, program generation with this technique is not always quicker than that without it. However, it is known to be quicker when using a rich set of combinators as the component library. In the Version 0.8.6.1 of MagicHaskeller , it can be tried by using `MagicHaskeller.LibTH.exploit`.

```
Prelude MagicHaskeller.LibTH> exploit $ \f -> f "abc"=="abcba"
\a -> list_para (reverse a) a (\_ c _ -> a ++ c)
^CInterrupted.
```

Although use of a rich library may be considered as cheating when benchmarking, it can be more useful than using a poor library, making the results more readable.

4 The modules for analytical synthesis

Since Version 0.8.6, an analytical synthesis engine is added to MagicHaskeller . It uses an algorithm that extends Igor II [Kitzelmann(2007)] [Hofmann et al.(2010)Hofmann, Kitzelmann, and Schmid] to enable generation of many programs as a stream of lists, where programs with few case splittings are highly prioritized and appear early. This is made possible by not stopping search when the best programs are found but continuing the search in the breadth-first manner.

Igor II sometimes suffers from the trade-off between the correctness of the generated programs and the computational complexity: (only) unwanted programs are generated unless there are enough number of examples to correctly specify the desired function, and no programs are generated if the number of examples is too large due to the time complexity for conducting unification. This trade-off often forces trial-and-error to users in order to find the adequate number of I/O example pairs, and sometimes causes search failure due to lack in such a number. Those cases often happen especially when arguments increase in different dimensions and there are some corner cases.

For example, Igor II cannot correctly synthesize the `concat` function in the Standard Prelude of Haskell and the `allodd` function that takes a list of integers and returns if all of its elements are odd. In both cases wrong functions are generated when given several examples, and no functions are obtained, at least within five minutes, when given more than ten examples.

Generating many programs analytically from insufficient input-output pairs and then filtering them with a separately-supplied predicate is an answer to this trade-off. In fact, those functions can be synthesized with our new analytical synthesis modules without any trial-and-error on the users' side. The trade-off is resolved by dividing the set of input-output examples into those for guiding the search and those for avoiding generation of unintended solutions. For the latter purpose, the user usually need only one general (in that it is not at an edge or corner case) example, and rarely need to enumerate many examples. In addition, even if some examples are required here, that hardly influences the efficiency. On the other hand, the set of input-output examples for guiding the search can be minimized. Moreover, even if there are not enough number of examples for this purpose, the solution can be found, though the search is less efficient than when the optimal number of examples are given.

The advantage of analytical synthesis over systematic exhaustive search is that there are functions which the exhaustive algorithm cannot synthesize within a realistic time span but analytical algorithms can. On the other hand, some other functions such as the Fibonacci function can be synthesized by exhaustive search but cannot be synthesized by analytical algorithms (without using a specialized addition function, which can be regarded as cheating). Currently the analytical synthesis engine works separately from the systematic exhaustive search engine, except that they share some common modules such as those defining the language and those implementing combinatorial search. Their cooperation to make a new synthesis engine will be tried in future. Also note that a paramorphism introduction operator like in
[Hofmann and Kitzelmann(2010)] is not implemented yet.

4.1 A simple example

In the next example the length function is synthesized. The analytical synthesis engine generates many programs that generalize `{f [] = 0; f [a] = 1}` without using any background knowledge functions. Then, it compiles the generated programs, and filter them with the predicate `\f -> f "12345" == 5`.

```
$ ghci -package MagicHaskeller -XTemplateHaskell -v0
Prelude> :m MagicHaskeller.RunAnalytical
Prelude MagicHaskeller.RunAnalytical> :set prompt >
> quickStart [d| f [] = 0; f [a] = 1 |] noBKQ (\f -> f "12345" == 5)
\a -> let fa (b@([])) = 0
          fa (b@(c : d)) = succ (fa d)
       in fa a :: forall t2 . [t2] -> Int
^CInterrupted.
```

The first two lines are not essential. The first line is for bringing module `MagicHaskeller.RunAnalytical` into scope. The second line literally replaces the command prompt with `>>`. The latter is not indispensable, but it is recommended when using a narrow screen, because a lot of information has to be input at each analytical synthesis.

The third line is important, though is not very difficult. The set of declarations surrounded by the Oxford bracket `[d| . |]` is a declaration quote of Template Haskell having type `Q [Dec]`. The `quickStart` function takes the set of input-output pairs of the target function as the first argument, the sets of input-output pairs of the background knowledge functions, and the predicate with which to filter the generated programs. The careful reader may notice that different syntaxes are used between the first argument and the third argument. For example, a variable pattern is used in the first argument while concrete values are used in the third argument, and the implicit equality is used in the first argument while the explicit one is used in the third argument. These are not the results of the author's fancy. Variable patterns can be used within the first argument because it has the form of a function definition. They are often required by analytical synthesis in order to make recursive calls, because a recursive call involves pattern matching. On the other hand, the third argument that is used for filtering the generated programs has to use concrete values, because they will not be abstractly interpreted but compiled and executed.

Although in the above example the type of the target function is not supplied, it may be supplied as the type signature declaration in the first argument, like `[d| f :: [a]->Int; f [] = 0; f [a] = 1 |]`. When the type signature is omitted, the type is inferred from the types of constructors appearing in the input-output pairs. Integral literals are assumed to have type `Int`. They are treated specially and converted into combinations of `0`, `succ`, and `negate`.

Patterns with `@` are called *as-pattern*s, and the argument is matched to both patterns at the both sides of `@`. In the above case, because the two `b`s are unused, use of as-patterns are actually unnecessary. Such redundant use of as-patterns will be removed from the future releases.

4.2 An example with a background knowledge function

When background knowledge function(s) should be used, they are specified in the second argument. In this case, the analytical synthesizer generates higher-order functions that take background knowledge functions as arguments. This should be noted when specifying the test function.

The next example shows synthesis of multiplication using addition as the background knowledge function. Note that multiplication is one of the boring functions that cannot be synthesized by IGOR II. Since `(+) :: Int -> Int -> Int` is used as the background knowledge function, the resulting programs require `(+)` as the first argument.

```
> :{
| quickStartF
|     [d| mult 0 x = 0;    mult 1 0 = 0;
|         mult 1 1 = 1;    mult 1 2 = 2;
|         mult 2 0 = 0;    mult 2 1 = 2;
|         mult 2 2 = 4 |]
|     [d| add 0 x = x;     add 1 0 = 1;
|         add 1 1 = 2;     add 1 2 = 3;
|         add 2 0 = 2;     add 2 1 = 3;
|         add 2 2 = 4 |]
|     (\f -> f (+) 5 6 == 30)
| :}
\fa a b -> let fb (c@0) d = 0
               fb (c@succe) d | succe > 0 = fa d (fb e d)
                      where e = succe - 1
```

```
            in fb a b :: (Int -> Int -> Int) -> Int -> Int -> Int
\fa a b -> let fb (c@0) d = 0
               fb (c@succe) d | succe > 0 = fa (fb e d) d
                      where e = succe - 1
            in fb a b :: (Int -> Int -> Int) -> Int -> Int -> Int
^CInterrupted.
```

`:{` and `:}` are used in order to input a multi-line expression. This syntax of GHCi is introduced to Version 7, and module `MagicHaskeller.RunAnalytical` does not work with the version of GHCi, but the actual one-line input is hand-edited into this syntax in order to fit the input line into the page width of this paper. Also, the effect of the command for changing the prompts is actually cancelled within the `:{ . :}` block, but we assume that it works there.

In this example, we used an action with letter `F`, namely `quickStartF`, in order to avoid printing equivalent functions. Of course, we could use `quickStart` here instead, though doing so in this case results in printing tons of expressions. The two results printed are still equivalent if we limit the background knowledge function to `(+)`, but they are different if we use a non-commutative operator. For this reason, these two functions are recognized as different.

4.3 Using types unknown to MagicHaskeller

The actions introduced so far only works for types known beforehand, whose constructors appear in `MagicHaskeller.CoreLang.defaultPrimitives`. Types that do not appear there can be dealt with in the following way:

```
> :{
| quickStartCF $(c [d| f [] = 0; f [a] = 1; f [a,b] = 2 |]) noBK
|               (\f -> f "foobar" == 6)
| :}
\a -> let fa (b@([])) = 0
          fa (b@(c : d)) = succ (fa d)
      in fa a :: forall t6 . [t6] -> Int
^CInterrupted.
```

where the function `c` extracts the values of constructors appearing in the Oxford bracket and hand them over to the `quickStartCF` action. The code block between `$(` and `)` describes what should be spliced, and there must not be spaces between `$` and `(`.

5 Conclusions

This paper exemplified the usage and behavior of the newest version of MagicHaskeller . In addition to the usage of its older systematic exhaustive search engine, that of the newly added analytical synthesis engine was also explained. The new analytical synthesis engine is based on Igor II , but can generate many hypothesis programs. By generating many programs analytically and testing them, we can have a new synthesis system that is more powerful than Igor II .

References

[Augusteijn(1999)] L. Augusteijn. Sorting morphisms. In *Advanced Functional Programming, LNCS 1608*, pages 1–27. Springer Verlag, 1999.

[Claessen and Hughes(2000)] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP'00: Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming*, pages 268–279. ACM, 2000.

[Hofmann and Kitzelmann(2010)] Martin Hofmann and Emanuel Kitzelmann. I/O guided detection of list catamorphisms: towards problem specific use of program templates in ip. In *Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, PEPM '10, pages 93–100, 2010.

[Hofmann et al.(2010)Hofmann, Kitzelmann, and Schmid] Martin Hofmann, Emanuel Kitzelmann, and Ute Schmid. Porting IgorII from maude to haskell. In Ute Schmid, Emanuel Kitzelmann, and Rinus Plasmeijer, editors, *Approaches and Applications of Inductive Programming, Third International Workshop, AAIP 2009*, volume 5812 of *LNCS*, pages 140–158, 2010.

[Jones(2005)] Isaac Jones. The haskell cabal: A common architecture for building applications and libraries. In *Sixth Symposium on Trends in Functional Programming*, pages 340–354, 2005.

[Katayama(2005a)] Susumu Katayama. Systematic search for lambda expressions. In *Sixth Symposium on Trends in Functional Programming*, pages 195–205, 2005a.

[Katayama(2005b)] Susumu Katayama. MagicHaskeller: A search-based inductive functional programming system. http://nautilus.cs.miyazaki-u.ac.jp/~skata/MagicHaskeller.html, 2005b.

[Katayama(2006)] Susumu Katayama. Library for systematic search for expressions and its efficiency evaluation. *WSEAS Transactions on Computers*, 12(5):3146–3153, 2006.

[Katayama(2008)] Susumu Katayama. Efficient exhaustive generation of functional programs using monte-carlo search with iterative deepening. In Tu Bao Ho and Zhi-Hua Zhou, editors, *PRICAI*, volume 5351 of *Lecture Notes in Computer Science*, pages 199–210. Springer, 2008. ISBN 978-3-540-89196-3.

[Katayama(2010)] Susumu Katayama. Recent improvements of MagicHaskeller. In Ute Schmid, Emanuel Kitzelmann, and Rinus Plasmeijer, editors, *Approaches and Applications of Inductive Programming, Third International Workshop, AAIP 2009*, volume 5812 of *LNCS*, pages 174–193, 2010.

[Kitzelmann(2007)] Emanuel Kitzelmann. Data-driven induction of recursive functions from input/output-examples. In *AAIP'07: Proceedings of the Workshop on Approaches and Applications of Inductive Programming*, pages 15–26, 2007.

[Sheard and Peyton Jones(2002)] Tim Sheard and Simon L. Peyton Jones. Template metaprogramming for Haskell. In *Haskell Workshop 2002*, October 2002. URL http://research.microsoft.com/Users/simonpj/papers/meta-haskell/meta-haskell.ps.