Towards Human-Level Inductive Functional Programming

Susumu Katayama(✉)

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

Abstract. Inductive programming is the framework for automated programming, obtaining generalized recursive programs from ambiguous specifications such as input-output examples. Development of an inductive programming system at the level of human programmers is desired, but it involves the trade off between scale and versatility which are difficult to go together.

This paper presents our research idea to enable synthesis of long programs while not limiting the algorithm to any domain, by automatically collecting the usage and request frequency of each function, estimating its usefulness, and reconstructing the component library containing component functions with which to synthesize desired functions. Hopefully this research will result in a more human-like automatic programming, which can lead to the development of adaptive planning with artificial general intelligence.

Keywords: Inductive programming · Code reuse · Functional programming

1 Introduction

Inductive functional programming (IFP) is the framework for automated programming for synthesizing recursive functional programs from ambiguous specifications such as input-output examples. This paper discusses how we can realize a human-level IFP system, where *human-level* means that the system is general-purpose but at the same time can synthesize large-scale programs. *General-purpose* means that the system can cope with unexpected synthesis problems for a Turing-complete (or nearly Turing-complete) language rather than only synthesizing programs in domain-specific languages by following a tailored procedure.

Previously, we developed a general-purpose practical IFP system called MagicHaskeller[2][3][1]. MagicHaskeller can instantly synthesize short functional programs without any restriction of the search space based on any prior knowledge, by holding a large memoization table in the memory.

[1] http://nautilus.cs.miyazaki-u.ac.jp/~skata/MagicHaskeller.html

© Springer International Publishing Switzerland 2015
J. Bieger (Ed.): AGI 2015, LNAI 9205, pp. 111–120, 2015.
DOI: 10.1007/978-3-319-21365-1_12

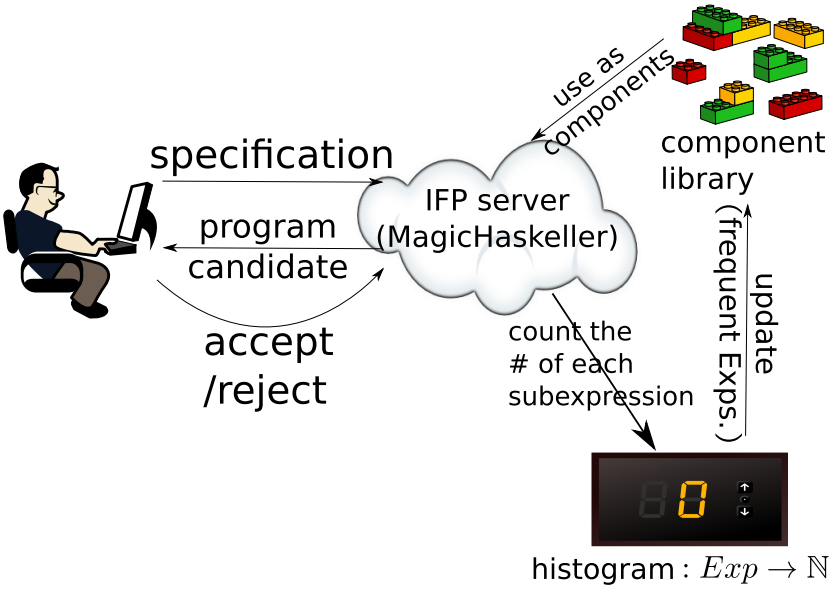

Fig. 1. Learning the component library from the data from the Internet

The other representative IFP systems are Igor II[4] and ADATE[5]. However, those are neither updated recently nor practical. Igor II enforces a tight restriction on the example set given as the specification, and ADATE requires high skill for synthesis of simple programs. Moreover, due to the absence of memoization, they have obvious disadvantage in the practical speed compared to MagicHaskeller which can start synthesis with its memoization table filled with expressions.

MagicHaskeller can synthesize only short expressions in a general-purpose framework by exhaustive search in the program space. In order to synthesize longer programs, the search has to be biased, because the program space is infinite. The most popular bias is *language bias* that restricts the search space around desired programs by carefully selecting the domain-specific language to be used. Language bias kills the generality, and thus is not our choice.

On the other hand, human programmers are ideal general-purpose inductive programming systems, and can synthesize programs in Turing-complete languages without any language bias. When we humans program, we name and reuse frequently-used functions and procedures, and synthesize larger libraries and programs using those library functions and procedures as the components. In other words, we adopt the bias based on the frequency of use.

The same thing can be achieved by collecting the data about how frequently each expression is requested and/or used, and organize the library consisting of frequently-used expressions and their subexpressions. This paper presents our research idea for realization of general-purpose large-scale IFP that is

specialized only to people's requirements, by collecting those frequency information from the Internet and reflecting them in the *component library*, or the set of functions and non-functional values with which to synthesize compound functions, of MagicHaskeller.(Fig. 1)

The rest of this paper is organized in the following way. Section 2 introduces MagicHaskeller. Section 3 argues that learning the library is a promising approach to synthesis of longer expressions. Section 4 discusses how to learn the library, including the details such as what to synthesize and how to collect data. Section 5 discusses how this research will be evaluated. We can expect that realization of large-scale IFP by this research will result in understanding the mechanism of adaptive problem solving conducted by humans and applications to learning behavior policies of intelligent agents such as robots; this is discussed in Section 6.

2 MagicHaskeller: A General-Purpose IFP System

The proposed research idea is automatic learning of the component library used by MagicHaskeller based on the data collected from the Internet. This section introduces MagicHaskeller.

MagicHaskeller is the representative IFP system adopting the generate-and-test approach.[2] When an ambiguous specification such as a set of input-output examples is given, MagicHaskeller firstly infers the type of desired expressions. Then, it generates expressions having that type (or more general type) that can be expressed using the component library functions, combined with function applications and λ-abstractions. They are generated exhaustively from the shortest one increasing the length, in the form of the infinite stream. They are then tested against the specification in order, and those passed the test are the synthesized functions.

The function taking the given type and returning the exhaustive infinite set of expressions having the type can be implemented efficiently by memoization, for this function recursively calls itself many times, because type-correct expressions consist of type-correct subexpressions. Memoization makes execution of this function very fast in most cases after *training*, or filling up the memoization table at the invocation of the synthesizer.

If implemented naively, the memoization table can be too large even when generating only short expressions if use of higher-order functions is permitted in order to implement recursive functions. This problem can practically be avoided when synthesizing short expressions, however, by pruning semantically equivalent expressions[3] and by sharing one memoization table served by one memory-rich computer among all the clients in the world. The Web version of MagicHaskeller uses more than a hundred component library functions, but

[2] The released MagicHaskeller library includes the analytically-generate-and-test module, but in this paper MagicHaskeller refers to the other part that implements the generate-and-test approach and serves the MagicHaskeller on the Web cloud.

thanks to this pruning the memoization table fits to the 64GB memory, and the server has been in use without any critical trouble since its birth three years ago.

Other notable features of MagicHaskeller include:

- it has a Web interface that enables program synthesis as offhanded as using a Web search engine;
- it supports various types, including numbers, characters, lists, tuples, higher-order functions, and their combinations.

3 Synthesizing Longer Expressions: How and Why

The points of MagicHaskeller are as follows:

- ability to synthesize usable expressions not limited to toy programs, by using a component library with more than a hundred functions;
- promptness thanks to memoization, despite of using such a large component library;
- avoiding redundancy in the memoization table caused by using a large component library, by eliminating expressions which are semantically equivalent to existing ones.

Although MagicHaskeller eliminates redundant expressions based on semantical equivalence, it cannot check infinite number of all the possible expressions within finite time. Hence, the search strategy of MagicHaskeller is biased to shorter expressions, based on the idea of Occam's Razor. It adopts no other bias than the length of expressions in order to cope with unexpected problem domains rather than specific use cases.

However, enumeration of expressions consisting of fixed library entities from the shortest expressions increasing the length never generates expressions longer than some length. This fact is the severest barrier when trying to make MagicHaskeller as powerful as human programmers.

Ideas for solving this problem include:

1. adopting the search strategy that searches promising branches deeper based on learning which branch is promising, and
2. learning the component library to make it consist of useful compound functions, and synthesizing expressions from more and more complicated components.

This paper argues that the solution 2 is promising.[3] The reasons are itemized below:

[3] We are *not* claiming that the solution 2 is *more* promising than the solution 1. Rather, we think that the solution 2 may be regarded as some form of the solution 1, by regarding use of learned functions as deep search without branching. Even then, the solution described in the form of the solution 2 is more straightforward than the solution 1.

– analogy to the human approach to programming and planning
  As already mentioned in the introduction, the process of naming frequently-used expressions, constructing the library consisting of those named expressions, and writing more complicated programs using those names as components, is similar to the way human programmers program.
  That process is also similar to the process of human planning through learning skills. Let us take the example of executing and learning the task of "going to school by train". In order to construct the solution to this task, we need to have already learned the executable solutions of its subtasks named as "walk from home to the station", "ride a train", and "walk to school". If we know how to execute those subtasks, we can find the solution of the task named as "go to school by train" by using only the subtask names and the constraints between subtasks without minding the implementation of each subtask. By repeating the solution, the name of "going to school by train" and the task are related, and the task becomes available for executing larger supertasks. This process is similar to the process of finding the desired program by combining library functions in the way consistent with their type constraints.
– analogy to successful AI approaches
  Our idea of making the component library consist of useful compound functions and synthesizing expressions from more and more complicated components has similarities to the following successful AI approaches:
  - **Genetic Algorithms**
    Genetic algorithms (GA) search for the fittest solutions by repeatedly applying crossovers and mutations to the population under natural selection. Adequately designed genetic algorithms sometimes find the best solution among other algorithms.
    The idea behind GA is to search among combinations of good characters via crossovers, assuming that good individuals consist of good characters. On the other hand, our presented idea is to search among combinations of component library functions which are from good (useful) expressions, assuming that good expressions consist of good subexpressions. At this point, our idea is similar to that behind GA.
    The reader may think genetic programming (GP) can be another option because it more directly inherits the idea behind GA. GP does not satisfy our purpose, however, because it requires designing of the fitness function and other parameters just for synthesizing one expression, and because it is not good at synthesizing recursive functions.
  - **Deep Learning**
    Deep learning[6][1] typically performs unsupervised pre-training for units near the inputs (or units far from the outputs) to extract features in artificial neural networks (ANN) with multiple hidden layers or recurrent neural networks (RNN). By deep learning the performance of ANNs has improved by leaps and bounds.
    ANNs with multiple layers (including those obtained by unfolding RNNs) can be regarded as function models which approximate the desired func-

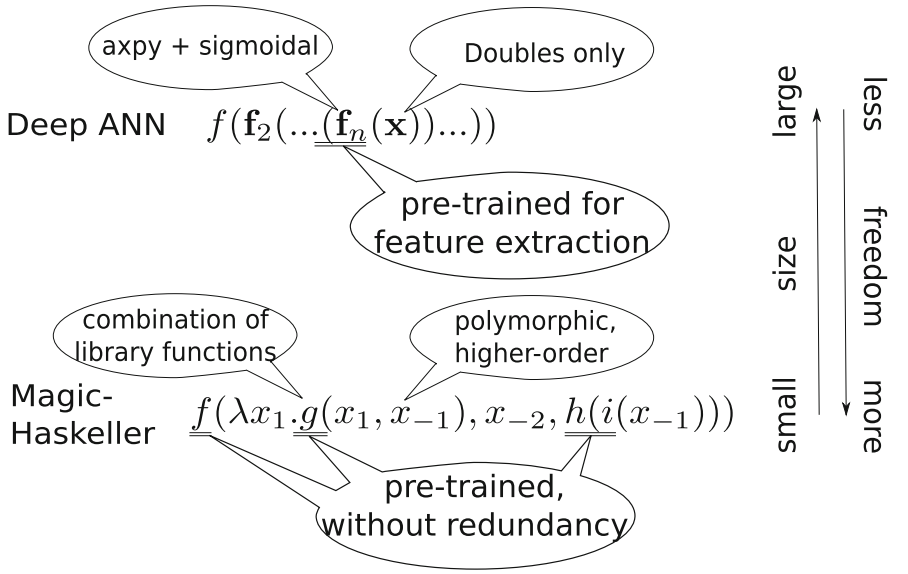

Fig. 2. Comparison with deep learning. The presented research idea adopts more flexible primitive function set than deep learning, but they are similar in that they both are pre-trained and eliminate redundancy.

tion by composing functions, where the neurons work as (families of) primitive functions. Units near inputs correspond to the innermost functions, and they play the role of extracting features.
Our presented research idea is to synthesize functions using functions from the component library, where the redundancy is eliminated by excluding semantically equivalent functions. This is similar to feature extraction by pre-training of deep learning.(Fig. 2)

4 How to Learn the Library

The previous Section 3 argued that learning the library is a promising approach to synthesis of longer expressions. This section discusses how to learn the library, including the details such as what to synthesize and how to collect data.

4.1 What to Synthesize

Currently, the algorithm of MagicHaskeller is mainly used for synthesis of pure Haskell functions, where *pure* means freedom from side effects. However, it can be applied to synthesis of pure functions in other higher-order languages with λ-abstraction such as JavaScript, by organizing the component library consisting of Haskell translations of library functions of the language. Moreover, technically speaking, pure functions in any language can be synthesized, provided that any

Haskell expression which MagicHaskeller generates can be compiled into the language.

Our current plan targets the following kinds of expressions:

- recursive functions in Haskell;
- recursive functions in JavaScript, especially, custom functions of Google Sheets;
- non-recursive functions using worksheet functions (of Microsoft Excel, &c.).

The reason for targeting spreadsheet functions as well as functions of usual programming languages is because the former can be better in the quantity and quality of collectable data. This is discussed further in Section 4.2.

As for spreadsheet functions, both recursive custom functions of Google Sheets and non-recursive worksheet functions will be dealt with. The synthesis of recursive custom functions will be dealt with because the synthesis of non-recursive functions using worksheet functions is less interesting than that of recursive functions. The synthesis of non-recursive worksheet functions will also be dealt with, because learning from abundant data collectable from the largest user base of Microsoft Excel is interesting, but it is not easy to synthesize custom functions of Excel in Visual Basic for Applications, which is first-order.

4.2 How to Collect Data

Collecting a large amount of usage data from the Internet will be a must for this research to be successful. We have two ways in mind:

1. *Providing an IFP service (or other services related to programming)*
   Since MagicHaskeller provides IFP service via a Web interface, we can analyze the server log to tell which queries were made and which answers were selected. This information should be useful for guessing desired functions.
   We need to take care of the quality and quantity of such queries. Most of them should be made by human users for the purpose of programming in practice in order to avoid contamination by unnaturally biased data. Currently, most of the queries to the MagicHaskeller server are unnatural ones based on academic curiosity about its ability, such as synthesizing the function taking x and returning $x/2$ if x is even and $x + 1$ otherwise. For this reason, it is questionable whether we should collect data for synthesis of Haskell expressions only in this way for now. This problem can be solved by increasing the percentage of practical users.
   On the other hand, spreadsheets such as Excel and Sheets have a lot of amateur users without such academic curiosity. Successful attraction for them will result in enough amount of data, though there must still be a defense against attacks for misleading the learning by a biased set of queries such as repeated identical ones.
2. *Obtaining packages from software repositories*
   If many of programs and libraries for the target language are made open-source, we can obtain a large amount of source codes by just downloading them.

The downloaded source codes can be processed in the compatible way as the queries to the server by following these steps for each function definition:
(a) generate an input-output pair for a random input, and
(b) send it as a query to the IFP server collecting data in the way described in 1. *Providing an IFP service (or other services related to programming)* to re-invent the function;
(c) if more than one program are synthesized, increase the number of random input-output pairs until one or zero program is obtained;
(d) if no program is synthesized, divide the function definition into subfunctions.

Those two ways can be combined. For example, the latter can be used to organize the initial component library, and then the former can be used to scale it up.

4.3 How to Organize and Update the Library

Frequently used expressions are candidates for component library functions because they are likely to be useful functions. However, all of such candidates cannot be adopted as component library functions with the same priority, because the space complexity increases as the number of them increases. For this reason, the library should be updated by selecting the function set rather than just adding some functions.

It is difficult to tell which is the best way of doing it now, because there are many options and parameters, and thus many policies. This section just shows the way which will be tried first.

When a Repository is Available. When dealing with a target language for which a software repository is available, we can exploit it for learning the initial configuration of the component library. In this case, the whole learning will be done in two steps:

1. Obtain the normalized set of expressions by processing the collected source codes in the way shown in Section 4.2, obtain the usage frequency of each subexpression, and obtain the set of the most frequent subexpressions for each subexpression length. They are sorted by each length, because shorter expressions tend to appear more frequently (especially any expression's frequency is always lower than or the same as those of its subexpressions). Then, organize the component library by hand using the obtained frequency information, and try the resulting IFP server. In this way, the function p which takes the length of the expression and the number it appears c and returns the priority $p(l, c)$ which the expression should have in the library can be guessed by trial-and-error.
2. Provide the IFP service, and sometimes update the library using $p(l, c + c')$ as the priority, where c' is the cumulative number the expression appears as a subexpression of each expression which is marked as correct by users. Because the cumulative values are used, it is unlikely that the library will become turbulent even when the library is updated frequently, but at the beginning each update should be checked by hand beforehand.

When a Repository is Not Available. When a repository is not available, the initial component library is set by hand in the same way as the currently-running MagicHaskeller server. The library can be updated in the same way as when a repository is available, using $p(l, c')$ where p is borrowed from another language.

5 Evaluation

It is difficult to fairly evaluate a general-purpose inductive programming system using a set of benchmark problems, for all the benchmark problems can easily be solved by implementing the functions to be synthesized beforehand and including them in the component library. Even if doing that is prohibited by the regulations, including their subexpressions in the component library is enough to make the problems much easier.

This issue is critical especially when evaluating results of this research, which is based on the idea: "The key to successful inductive programming systems is the choice of library functions", because we may not fixate the library for comparison, but rather we have to evaluate and compare the libraries themselves.

It would be fairer to evaluate systems from the perspective of whether the infinite set of functions that can be synthesized covers the set of many functions which the users want. In the case of this research, since IFP service will be provided as a web application, we can evaluate how the obtained IFP system can satisfy programming requests based on the results of Web questionnaire and the Web server statistics.

6 Expected Contribution to AGI

Making large-scale IFP possible by this research may uncover the mechanism of adaptive problem solving conducted by humans, and may be applied to behavior policy learning of intelligent agents.

To repeat what is stated in Section 3, this research imitates the human adaptive intelligent behaviors of programming and planning. Especially, learning to plan is important in that it explains the process of skill learning of humans.

For example, imagine children learning addition of two numbers, say, 2+3. At first, they might use two piles of apples consisting of two and three of them, and compute the result by moving apples from one pile to the other one by one. After drills, however, their brain will come to associate $2 + 3$ to 5 instantly.(Fig. 3) Once they have obtained the library function "one-digit addition", they can learn multiple-digit addition by using it, and can go further to learn multiplication. This process is quite similar to the process of learning more and more complicated library functions by the presented research idea.

Adaptive planning for intelligent agents sometimes requires learning recursive procedures. This kind of program-like procedures are difficult to be represented by function approximation such as existing artificial neural network models, while IFP systems such as MagicHaskeller are good at representing them.

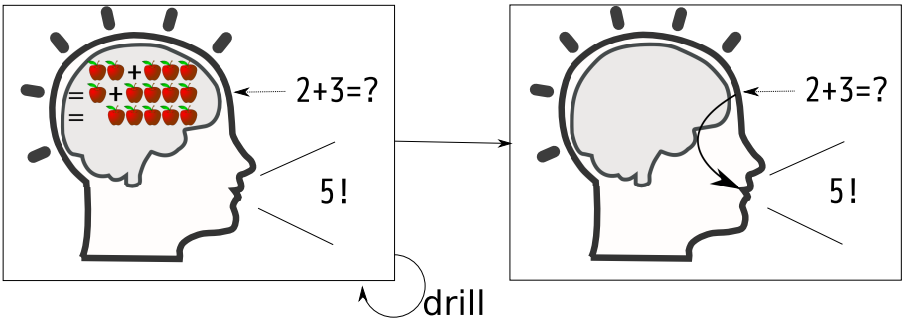

Fig. 3. Learning to calculate

Moreover, because MagicHaskeller can synthesize recursive programs from only a few positive examples, it can learn only from rewards, not requesting many negative examples which need to be generated by failing critically. This is why the proposed approach seems to be the best for learning complicated procedures with recursions only from the reward signal.

7 Conclusions

This paper presented our research idea for realizing a human-level IFP system by adding the library learning functionality to the Web-based general-purpose IFP system MagicHaskeller. It can be applied to uncovering the AGI mechanism for human-like learning of behavior and to developing intelligent agents.

References

1. Hinton, G.E., Salakhutdinov, R.R.: Reducing the Dimensionality of Data with Neural Networks. Science **313**(5786), 504–507 (2006)
2. Katayama, S.: Systematic search for lambda expressions. In: Sixth Symposium on Trends in Functional Programming, pp. 195–205 (2005)
3. Katayama, S.: Efficient exhaustive generation of functional programs using monte-carlo search with iterative deepening. In: Ho, T.-B., Zhou, Z.-H. (eds.) PRICAI 2008. LNCS (LNAI), vol. 5351, pp. 199–210. Springer, Heidelberg (2008)
4. Kitzelmann, E.: A Combined Analytical and Search-Based Approach to the Inductive Synthesis of Functional Programs. Ph.D. thesis, University of Bamberg (2010)
5. Olsson, R.: Inductive functional programming using incremental program transformation. Artificial Intelligence **74**(1), 55–81 (1995)
6. Schmidhuber, J.: Learning complex, extended sequences using the principle of history compression. Neural Comput. **4**(2), 234–242 (1992)