An Analytical Inductive Functional Programming System that Avoids Unintended Programs

Susumu Katayama

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

Abstract. Inductive functional programming (IFP) is a research field extending from software science to artificial intelligence that deals with functional program synthesis based on generalization from ambiguous specifications, usually given as input-output example pairs. Currently, the approaches to IFP can be categorized into two general groups: the analytical approach that is based on analysis of the input-output example pairs, and the generate-and-test approach that is based on generation and testing of many candidate programs. This paper proposes a new analytical inductive functional programming system that generates, tests, and selects from many program candidates. For generating many candidate programs, the proposed system uses a new variant of IGOR II$_H$, the exemplary analytical inductive functional programming algorithm. This new system preserves the efficiency features of analytical approaches, while being robust to changes in the number of input-output examples while minimizing the possibility of generating unintended programs. In addition, this research can be considered a milestone in the fusion of both approaches in that it provides an analytical algorithm implemented in the same way as a generate-and-test algorithm and reveals the strengths and weaknesses of both approaches.

1 Introduction

Inductive functional programming (IFP) algorithms automatically generate functional programs from ambiguous specifications such as a set of input-output (I/O) example pairs or a loose condition to be satisfied by inputs and outputs. The term can include cases where no recursion is involved, as in genetic programming, but it usually involves generation of recursive functional programs.

Currently, two approaches to IFP are under active development. One is the analytical approach that performs pattern matching to the given I/O example pairs (e.g., [1] which was used for implementing the IGOR II system), and the other is the generate-and-test approach that generates many programs and selects those that satisfy the given condition (e.g., [2] which was used for implementing the MAGICHASKELLER system and [3] which was used for implementing the ADATE system). Analytical methods are efficient in general, but they have limitations on how to provide I/O relations as the specification, and, in general, the user has to provide many I/O examples, beginning with the simplest one(s) and progressively increasing their complexity. On the other hand, generate-and-test methods do not usually have limitations on the specification to be given (except, of course, that it must be

written in a machine-executable form[1]), but tend to require more time compared to analytical methods.

This paper considers the improvements that can be made to the algorithm behind IGOR II [1][4], which is the state-of-the-art analytical IFP system, to generate many program candidates by rewriting it using Spivey's monadic interface for combinatorial search [5]. The resulting algorithm has the following advantages:

- It can achieve the same properties as generate-and-test methods by generating all possible programs by using the fixed set of operators
- It could be combined with MAGICHASKELLER, which also uses Spivey's monadic interface for implementation.

The first point will be amplified by considering the example of synthesizing the *reverse* function, assuming it is not known how to implement it. IGOR II requires the trace (or the set of all I/O pairs that appear during computation) of the biggest example as the set of examples, as in Table 1.[2] In this case, the first four lines are the computational traces of the last line and, hence, cannot be omitted. In general, it is tedious to write down all the required elements in order to generate a desired program, or a program with the intended behavior. Moreover, it is sometimes necessary to consider which examples are necessary in order to reduce the number for efficiency reasons. As a result, the user sometimes has to tune the set of examples until a desired program is obtained within a realistic time span.

On the other hand, MAGICHASKELLER, which is a type of generate-and-test system that uses systematic exhaustive search for generating programs, can generate a desired program from only one example of *reverse* $[1,2,3,4,5] \equiv [5,4,3,2,1]$. Obviously, this example has enough information to specify the intended function — the intention of the writer of this longer example is clear, while short examples such as *reverse* $[\,] = [\,]$ or *reverse* $[a] = [a]$ can be interpreted in many different ways.[3] MAGICHASKELLER often succeeds in generalization from only one example by the minimal program length criterion, or by selecting the shortest program that satisfies the given specification. Likewise, it can be expected that the same effect can apply even when using an analytical approach by analytically generating many candidate programs based on the given few examples and selecting those that satisfy one larger example, rather than generating a single candidate.

[1] Some readers may think that termination within a realistic time span is another limitation on the specification. Termination of the specification does not mean termination of the test process within a realistic time span, because the latter involves execution of machine-generated programs which may request arbitrary computation time. For this reason, time-out is almost indispensable for systems like MAGICHASKELLER, and in this case there is no such limitation on the specification.

[2] Throughout this paper, HASKELL's notation is used for expressions.

[3] Of course, the well-known *reverse* function is not the only function that satisfies the longer example. For example, there can be a case where we want to change the return value only for $[\,]$. This is not a problem, however, because no one would consider such a function by only giving the example of *reverse* $[1,2,3,4,5] \equiv [5,4,3,2,1]$, but most people would add the example for $[\,]$ in this case.

2 Preparation

This section introduces related IFP systems, IGOR II and MAGICHASKELLER.

2.1 Igor II

The algorithm behind IGOR II [1] synthesizes a recursive program that generalizes the given set of I/O examples by regarding them as term-rewriting rules through pattern matching. Early versions by Kitzelmann were written in MAUDE and interpreted, but recent implementations are in HASKELL, named IGOR II$_H$ [4], which is a simple port, and IGOR II$^+$ [6], which is an extension with support of catamorphism/paramorphism introduction. Such support is known to result in efficient algorithms, though this paper does not deal with those morphisms and, thus, is a counterpart of IGOR II and IGOR II$_H$.

These algorithms run in the following way:

1. Obtain the least general generalization of the set of the I/O examples by antiunification. This step extracts the common constructors and allows the uncommon terms to be represented as variables. Here, the same variable name is assigned to terms with the same example set. Variables that do not appear in the argument list represent unfinished terms.
2. Try the following operators[4] in order to complete the unfinished terms. Then, expressions with the least cost are kept, and others are abandoned. The cost function will be explained later in this section.
   - **Case partitioning** operator introduces a case partitioning based on the constructor set of input examples, and tries this for each argument. Now, case bodies can include new unfinished terms. Each case can be finished by applying this algorithm recursively, supplying each field of the constructor application as additional inputs.
   - **Constructor introduction** [5] operator introduces a constructor, if all output examples share the same one at the outermost position. Also introduce new functions to all fields, and supply the same set of arguments for that of the left hand side. Again, this part can be finished by applying this algorithm recursively, because it is possible to infer the I/O relation of the new function by reusing the same input example list and using each field of the constructor applications as output examples.
   - **Defined function call** operator introduces either a function from the background knowledge (namely a predefined primitive function that works as a heuristic) if available, or a function already defined somewhere (causing a

[4] The term 'operator' is also used for 'operator' in 'binary operator'. In order to avoid confusion, in the latter case, either its arity or HASKELL's operator name will always be mentioned, for example, '(+) operator'.

[5] This is usually called 'introducing auxiliary functions' (e.g., [6]), but in this paper it is called 'constructor introduction', because 1) it is the common constructor that is introduced specifically by this operator, 2) auxiliary functions are introduced even by other operators, and 3) the term 'auxiliary function' can be confused with the third operator.

Table 1. Example of synthesizing the *reverse* function using IGOR II$_H$: the input source text (taken from Version 0.7.1.3 of IGOR II$_H$ release) (left) and the resulting program (right)

```
reverse []            = []
reverse [a]           = [a]
reverse [a, b]        = [b, a]
reverse [a, b, c]     = [c, b, a]
reverse [a, b, c, d] = [d, c, b, a]
```

```
reverse []                = []
reverse (a0 : a1)         = fun1 (a0 : a1) : fun2 (a0 : a1)
fun1    [a0]              = a0
fun1    (_ : (a1 : a2))   = fun1 (a1 : a2)
fun11   (_ : (a1 : a2))   = a1 : a2
fun2    (a0 : a1)         = reverse (fun5 (a0 : a1))
fun5    [_]               = []
fun5    (a0 : (a1 : a2)) = a0 : fun9 (a0 : (a1 : a2))
fun9    (a0 : (a1 : a2)) = fun5 (fun11 (a0 : (a1 : a2)))
```

recursive call). These functions are called *defined function*s in both cases, and they are also represented as a set of I/O example pairs. Now, for each defined function f, the IGOR II algorithm tries to match the set of output examples that the unfinished term should return to that of f. Then, successful f's are adopted here.
Each argument of f is unknown, and thus a new function is introduced here. Again, it can finish this part by applying this algorithm recursively, because the I/O relation of the new function can be inferred by reusing the same input example list and using the input examples of the defined function as output examples.

Catamorphism introduction (optionally with IGOR II$^+$) introduces catamorphism. This can make some synthesis tractable, while it can slow down others. This operator is not yet included in the current implementation of the proposed algorithm.

Table 1 shows an example of synthesizing the *reverse* function using IGOR II$_H$.

Limitations of Igor II The IGOR II algorithm has the following problems:

- IGOR II does not work correctly if we omit a line in the middle of the set of I/O examples; taking an example of *reverse*, if we omit the fourth line stating *reverse* $[a, b, c] = [c, b, a]$ from Table 1, it fails to synthesize correctly.
- There are many possible combinations while matching the target function to a defined function. Hence, an increase in the number of I/O examples easily slows down the synthesis.

Those problems incur a trade-off between the efficiency and the accuracy: in order to minimize the ambiguity a big example should be included in the example set; however, this means that all the smaller examples also have to be included, and as a result, the efficiency is sacrificed. This is problematic especially when examples increase in different dimensions. In fact, some functions such as multiplication cannot be synthesized by IGOR II$_H$ due to this trade-off. The proposed system solves this trade-off.

Cost and preference bias When searching in a broad space, in which priority order to try options is also an important factor in order to find answers in a realistic

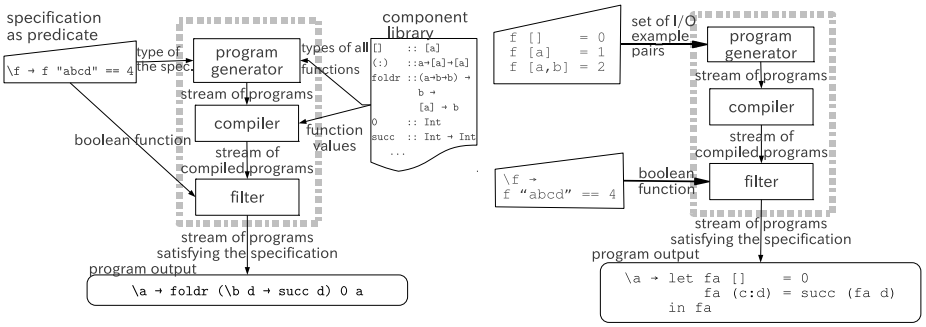

Fig. 1. The structure of MagicHaskeller (left) and the proposed system (right).

time span. Igor II defines a cost function that returns a tuple of the number of case distinctions, the number of open rules, etc., and the returned tuples are compared in lexicographical order. The search is implemented statefully by keeping track of the set of the best programs with the least cost.

Simply keeping track of the set of best programs is heap efficient, but that also means that second-best programs measured by the given cost function are abandoned, thus making it difficult to salvage a right program when the best programs are not actually those intended by the user.

2.2 MagicHaskeller

MagicHaskeller [7][8][9] is a generate-and-test method based on systematic search. One of its design policies goes "Programming using an automatic programming system must be easier than programming by one's own brain", and ease of use is its remarkable feature compared with other methods. Unlike other methods requiring users to write down many lines of programming task specification for each synthesis, users of MagicHaskeller only need to write down the specification of the desired function as a boolean function that takes the desired function as an argument. For example, the reverse function can be synthesized by only writing $printOne\ \$\ \lambda f \rightarrow f$ `"abcde"` $\equiv$ `"edcba"`. This is achieved by not using heuristics whose effectiveness is questionable and by enabling a general purpose primitive set (called a component library) that can be shared between different syntheses. On the other hand, because it searches exhaustively, its ability to synthesize big programs is hopeless. However, having heuristics and not doing an exhaustive search does not always mean that an algorithm can synthesize big programs, unless it is designed adequately and works well. According to benchmarks from the literature [10] and inductive-programming.org[6], at least it can be claimed that MagicHaskeller performs well compared with other methods.

Figure 1(left)depicts the structure of MagicHaskeller. Its heart is the program generator, which generates all the type-correct expressions that can be expressed by function application and λ abstraction using the functions in the given

[6] `http://www.inductive-programming.org/repository.html`

component library as a stream from the smallest and increasing the size. The generation is exhaustive, except that MagicHaskeller tries not to generate syntactically different but semantically equivalent expressions. The generation of expressions with the given type is equivalent to that of proofs for the given proposition under Curry-Howard isomorphism, and the MagicHaskeller algorithm [2] is essentially an extension of an automatic prover algorithm that can generate infinite number of proofs exhaustively.[9] MagicHaskeller adopts the breadth-first search for generating infinite number of proofs, and this is achieved by using a variant of Spivey's monad for breadth-first search [11]. All the generated expressions are compiled and tested by the given test function. By generating a stream of expressions progressively from the smallest and testing them, the most adequate generalization of the given specification that avoids overfitting comes first by the minimal program length criterion.

The component library corresponds to the set of axioms in a proof system under Curry-Howard isomorphism (e.g., [12]). It should consist of total functions including constructors and paramorphisms / catamorphisms, because permitting partial functions in the component library may make any type inhabited and causes search space bloat. As a result, MagicHaskeller with the default component library cannot generate partial functions without an inhabited type, such as $head :: [a] \to a$.

Early versions of MagicHaskeller try to detect and prune as many semantically equivalent expressions as possible by applying known optimization rules.[2] This involves guessing which in the component library are case functions, catamorphisms, or paramorphisms. Because such guessing does not work for user-defined types, this optimization was once removed, but now it is available by an option or by *init075* action.

3 Proposed Algorithm

As mentioned in Section 1, the proposed algorithm is an Igor II -variant that generates many program candidates, which is implemented using Spivey's monadic framework for combinatorial search [11][5]. For this reason, Spivey's framework is first reviewed and then the actual implementation is described.

When describing the algorithm, first the design policy and then the (somewhat simplified) algorithm are presented. The algorithm used for evaluation infers type while generating programs in order to narrow the search space, though this part is omitted in this paper.

3.1 Preparation: Monadic framework for combinatory search

Spivey [11][5] defined a very convenient interface to search strategies that simplifies the task of writing combinatory search algorithms. It can be defined as an extension of Haskell 's *MonadPlus* class (which means that the structure of the search strategies is monadic and monoidal) with a new method *wrap*:

class *MonadPlus* $m \Rightarrow$ *Search* m **where**
  wrap :: $m\ a \to m\ a$

MonadPlus class has two methods inherited from its parent *Monad* class, namely *return* :: *forall* $a.\ a \to m\ a$ and $(\triangleright)$ :: *forall* $a\ b.\ m\ a \to (a \to m\ b) \to m\ b$, and

two monoidal methods, the unit *mzero* :: *forall a. m a* and the product *mplus* :: *forall a. m a* → *m a* → *m a*. As usual, *return* is used to wrap a normal value to make a search result, ▷ to combine two successive search operations, *mzero* to represent search failure, and *mplus* to try two alternatives (and *msum* to try a list of alternatives). In addition, *wrap* is used for degrading the priority of the current action.

Other monad-related functions from the standard library are also available. For example, the following is a code that enumerates all integers that are results of repeated addition, subtraction, and multiplication over 1, 2, and 3, ordered from those with small number of operators progressively.

```
nums ::     Search m ⇒ m Int
nums = fromList [1 . . 3] `mplus`
           wrap (liftM3 ($) (fromList [(+), (−), (∗)]) nums nums)
fromList = msum. map return
```

This way, Spivey's monadic framework makes programming tasks of combinatorial search quite simple. One can enumerate expressions by enumerating function heads, enumerating their arguments by recursive calls, and applying them in the lifted way. Additionally, you may have to filter out those functions and subexpressions that do not satisfy the given constraints during enumeration.

In spite of the simplicity, this class covers useful search strategies like breadth-first search, depth-first search, and depth-bound search with/without iterative deepening. In addition, the design choice of using *wrap* method makes the algorithm applicable to best-first search without the idea of the depth in a search tree as long as the cost takes a natural number.

3.2 The design policy

The design policy is:

- First, candidates for the head (namely, the outermost function) explaining the examples are searched for, and then the spine (namely, the set of arguments) is formed for each of them by recursively synthesizing all the possible subexpressions to which each of the head candidates can be applied. This is in the same way as the *nums* example shown in Section 3.1. Also, MagicHaskeller is implemented in the same way.
- Case partitioning can be introduced by regarding case functions as the head; constructor introduction by regarding constructors as the head; and call for defined functions by regarding the functions themselves as the head.
- Antiunification has two effects: detecting common constructors and detecting subexpressions changing together; the former can be considered as a part of case partitioning and constructor introduction, and the latter can be implemented as a new operator **projection function introduction** which finalizes the synthesis of the current subexpression by finding an argument whose examples unify with the return value examples.
- Argument subexpressions are finished to form a spine by lazy recursive calls at once rather than explicitly representing unfinished expressions for now and then finishing them later by recursive calls as in Igor II$_H$. Since lazy evaluation

works, unbound variables can be identified with thunks, and it can be claimed that IGOR II$_H$ explicitly implements the lazy evaluation part of the proposed algorithm.
- All the search operators (except catamorphism introduction for now) are tried sooner or later, and this is implemented by taking their *msum*.
- For now, only natural costs are used. Cost n can be introduced by applying the *wrap* function (which is the function that lowers the priority by one depth in the Spivey's interface) n times. Case partitionings are assigned more cost than other operators because they cost in IGOR II. Constructor introduction and projection function introduction are assigned the least cost, because they correspond to antiunification, which supersedes other operators.

3.3 Outline of the algorithm

The simplified algorithm takes the tuple of the set of I/O pairs specifying the desired function and the sets of I/O pairs specifying defined functions, and it returns the prioritized set of generalized functions in the form of Spivey's search monad. If we call the function *synthesize*, it adds the current target I/O pairs to the set of defined functions, tries the four operators (which can be achieved by just adding the four computations with the *mplus* method), and calls the *synthesize* function recursively in order to synthesize the arguments by solving the induction problems from the I/O pairs that are presented by each of the operators.

3.4 Abstract definitions of the four operators

The proposed algorithm does not antiunify, and it has an additional operator, projection function introduction, instead. This section is devoted to providing with abstract definitions of these operators.

Projection function introduction is the simplest operator. Under the condition of

$$\forall j \in \{1...m\}.\ f\ x_{j1}\ ...\ x_{ji}\ ...\ x_{jn} = x_{ji}$$

it induces

$$\forall v_1...v_n.\ f\ v_1\ ...\ v_i\ ...\ v_n = v_i$$

This operator is tried for each argument $i \in \{1...n\}$.

Constructor introduction extracts a common constructor among the output examples of the I/O pairs of the given target function f. Under the condition of

$$\forall j \in \{1...m\}.\ f\ x_{j1}\ ...\ x_{jn} = C\ y_{j1}\ ...\ y_{jq}$$

it induces

$$\forall v_1...v_n.\ f\ v_1\ ...\ v_n = C\ (h_1\ v_1\ ...\ v_n)\ ...\ (h_q\ v_1\ ...\ v_n)$$

where

$$\forall i \in \{1...q\}.\forall j \in \{1...m\}.h_i\ x_{j1}\ ...\ x_{jn} = y_{ji}$$

and q is the arity of C. Further induction of $h_1 ... h_q$ from the newly introduced I/O pairs is required unless C is nullary.

In practice, C need not be a constructor but can be a function from a library. When this is permitted, synthesis from, e.g.,

$$\begin{aligned} &sum\ [\,] &&= 0 \\ &sum\ [x] &&= x \\ &sum\ [x, y] &&= x + y \\ &sum\ [x, y, z] &&= x + (y + z) \end{aligned}$$

is also possible.

Case partitioning focuses on an argument of the target function f, and puts together I/O pairs with such actual arguments that share the same constructor. Under the condition of

$$\begin{aligned} &\forall j \in \{1...q\}.\forall k \in \{1...p_j\}. \\ &\quad f\ x_{jk1}\ ...\ x_{jk(i-1)}\ (C_j\ z_{k1}\ ...\ z_{ko_j})\ x_{jk(i+1)}\ ...\ x_{jkn} = y_{jk} \end{aligned}$$

it induces

$$\begin{aligned} &\forall j \in \{1...q\}.\forall v_1...v_n.\forall u_1...u_{o_j}. \\ &\quad f\ v_1\ ...\ v_{i-1}\ (C_j\ u_1\ ...\ u_{o_j})\ v_{i+1}\ ...\ v_n = h_j\ v_1\ ...\ v_{i-1}\ v_{i+1}\ ...\ v_n\ u_1\ ...\ u_{o_j} \end{aligned}$$

where

$$\begin{aligned} &\forall j \in \{1...q\}.\forall k \in \{1...p_j\}. \\ &\quad h_j\ x_{jk1}\ ...\ x_{jk(i-1)}\ x_{jk(i+1)}\ ...\ x_{jkn}\ z_{k1}\ ...\ z_{ko_j} = y_{jk} \end{aligned}$$

Further inference of $h_1 ... h_q$ from the newly introduced I/O pairs is required. This operator is tried for each argument $i \in \{1..n\}$.

Note that this definition is slightly different from case partitioning of Igor II. In the proposed algorithm, the number of cases is equivalent to that of constructors that appear, while Igor II is more liberal about the number of cases and may put together I/O pairs with different constructors. Also, case partitioning of the proposed algorithm removes constructors, while that is done by antiunification of Igor II.

Defined function introduction matches the output examples of the target function f to those of a defined function g. Under the condition of

$$\begin{aligned} &\forall j \in \{1...m\}.\ f\ x_{j1}\ ...\ x_{jn} = \theta_j(w_{r_j}) \\ &\forall i \in \{1...p\}.\ g\ z_{i1}\ ...\ z_{iq} = w_i \end{aligned}$$

for existing substitutions $\theta_1 ... \theta_m$ and $r_1 ... r_m \in \{1...p\}$, it induces

$$\forall v_1...v_n.\ f\ v_1\ ...\ v_n = g\ (h_1\ v_1\ ...\ v_n)\ ...\ (h_q\ v_1\ ...\ v_n)$$

where

$$\forall k \in \{1...q\}.\forall j \in \{1...m\}.\ h_k\ x_{j1}\ ...\ x_{jn} = \theta_j(z_{r_jk})$$

Further inference of $h_1 ... h_q$ from the newly introduced I/O pairs is required. This operator is tried for each selection of defined function g and for each selection of $r_j|_{j\in\{1...m\}}$. Then, the loop checker checks if g is called with a "smaller" argument list in a well-formed sense than when g was first called.

3.5 Implementation of the four operators

The implementation of the four operators comes from interpretation of their abstract definitions.

All the applicable operators are tried, and if there are multiple possibilities within the operators, all of them are tried. This can be implemented by adding all the possibilities with *mplus*.

The applicability of each operator is decided by pattern matching. When the operators is defined in the format of "Under the condition of A it infers B (where C)" in Section 3.4, the target I/O examples for f is matched to A, and all the corresponding B's represent the induced programs.

Operators other than projection function introduction have the "where C" part, which defines the I/O examples for synthesizing the subexpressions recursively.

3.6 Efficient matching using a generalized trie

For each defined function and for each output example of the target function, the defined function introduction operator collects all the output examples of the defined function that the target output example matches to. The naive implementation of this process executes matching mn times for each defined function, where m denotes the number of I/O examples of the target function, and n denotes that of the defined function, and thus forms a bottleneck here. Our idea is to use the generalized trie [13] indexed by the output example expressions and to put all the I/O examples into the trie. Then, the n examples can be processed at once while descending the trie, by collecting values whose keys match the given expression. This is possible because the indexing of such generalized tries reflects the data structure of the index type, unlike hash tables.

Although it is difficult to be specific about the time complexity of the resulting algorithm, the algorithm reduces the computation time a great deal, and matching is not the bottleneck any longer.

4 Experimental Evaluation

This section presents the results of the evaluation of the proposed system empirically on its time efficiency and robustness to changes in the set of I/O examples. Here the term "the proposed system" means the system that executes the algorithm introduced in Section 3 and filters the resulting stream of programs with the user-supplied test function. (Figure 1(right)) The reader should note that users of the proposed system have to write the test function, as well as the I/O example pairs, while users of MagicHaskeller have only to write the test function.

4.1 Experiment conditions

Compared systems The proposed system was compared with the nolog release of Igor II$_H$ Ver. 0.7.1.2, which is the latest nolog release (or release without logging overhead) at the time of writing, and with MagicHaskeller Ver. 0.8.5-1. Comparisons with other conventional inductive programming systems are omitted

since comparisons between conventional systems including IGOR II$_H$ and MAGICHASKELLER on the same programming tasks are already in the literature ([10] and inductive-programming.org[6]).

As for the search monad for the proposed system, based on preliminary experiments, Spivey's monad for breadth-first search [11] was selected over other alternatives that fit into Spivey [5]'s interface, such as depth-bound search and their recomputing variants, such as the *Recomp* monad [2].

MAGICHASKELLER was initialized with its *init075* action, which means that aggressive optimization without proof of exhaustiveness was enabled, like in its old stand-alone versions. By default, MAGICHASKELLER does not look into the contents of each component library function (or background knowledge function in the terminology of analytical synthesis) but only looks at their types. With *init075* action, however, it prunes the redundant search by guessing which are consumer functions such as case functions, catamorphisms, and paramorphisms, though some expressions with user-defined types may become impossible to synthesize due to language bias. This condition is fairer when compared with analytical approaches that know what case functions do.

Set of programming tasks Table 2 shows the test functions of the target functions used for filtering the generated programs. These test functions are higher-order predicates that the target functions should satisfy, and they were supplied to MAGICHASKELLER and the testing phase of the proposed system without modifications.

Their expected behaviors are not shown in this paper explicitly, mainly due to the page limit. They are explained in the benchmark site[6], though we believe that their test functions explain what the target functions are supposed to do and that usual human mind can generalize from the examples correctly with the hint of their names.

The left column of Table 2 shows the set of function names that were to be synthesized. They were selected by the following conditions:

- their I/O example pairs that are usable for synthesis are bundled in the IGOR II$_H$ release, and
- they have already been compared with MAGICHASKELLER somewhere.

The second condition is about the adequacy of the task, and it was decided not to exclude those whose evaluation is temporarily postponed at the benchmark site[6]. Those programs that are too easy and require less than 0.5 second on all the systems were also excluded from the table. All of the other functions that were correctly answered by IGOR II$_H$ within five minutes are included, provided that they satisfy the above conditions.

The first condition is included in order to fix the I/O example set by using those bundled as is. In analytical synthesis, the efficiency largely depends on the number of examples (except for the cases where the computation finishes instantly). For example, the set of I/O example pairs bundled in IGOR II$_H$ for generating ($\equiv$) compares two natural numbers between 0 and 2 in 9 ways — recursive programs could not be obtained if there were only 4 examples, while the computation would not be completed in a realistic time if there were 16 examples. Due to this problem,

Table 2. Test functions for target functions used to filter results from MagicHaskeller and the proposed system.

name	test function
addN	$addN\ 3\ [5,7,2] \equiv [8,10,5]$
allodd	$allodd\ [3,3] \wedge \neg\ (allodd\ [2,3]) \wedge allodd\ [1,3,5] \wedge \neg\ (allodd\ [3,7,5,1,2])$
andL	$\neg\ (andL\ [True, False]) \wedge andL\ [True, True]$ $\wedge\ andL\ [True, True, True] \wedge \neg\ (andL\ [False, True, True])$
concat	$concat$ `["abc","","de","fghi"]` $\equiv$ `"abcdefghi"`
drop	$drop\ 3$ `"abcde"` $\equiv$ `"de"`
$(\equiv)$	$3 \equiv 3 \wedge \neg\ (4 \equiv 6) \wedge 0 \equiv 0 \wedge \neg\ (2 \equiv 0) \wedge \neg\ (0 \equiv 2) \wedge \neg\ (3 \equiv 5)$
evenpos	$evenpos$ `"abcdefg"` $\equiv$ `"bdf"`
evens	$evens\ [4,6,9,2,3,8,8] \equiv [4,6,2,8,8]$
fib	$fib\ 0 \equiv 1 \wedge fib\ 1 \equiv 1 \wedge fib\ 3 \equiv 3 \wedge fib\ 5 \equiv 8 \wedge fib\ 7 \equiv 21$
head	$head$ `"abcde"` $\equiv$ `'a'`
init	$init$ `"foobar"` $\equiv$ `"fooba"`
$(+\!\!+)$	`"foo"` $+\!\!+$ `"bar"` $\equiv$ `"foobar"`
last	$last$ `"abcde"` $\equiv$ `'e'`
lasts	$lasts$ `["abcdef","abc","abcde"]` $\equiv$ `"fce"`
lengths	$lengths$ `["abcdef","abc","abcde"]` $\equiv [6,3,5]$
multfst	$multfst$ `"abcdef"` $\equiv$ `"aaaaaa"`
multlst	$multlst$ `"abcdef"` $\equiv$ `"ffffff"`
negateAll	$negateAll\ [True, False, False, True] \equiv [False, True, True, False]$ $\wedge\ negateAll\ [False, True, False] \equiv [True, False, True]$
oddpos	$oddpos$ `"abcdef"` $\equiv$ `"ace"` $\wedge$ $oddpos$ `"abc"` $\equiv$ `"ac"`
reverse	$reverse$ `"abcde"` $\equiv$ `"edcba"`
shiftl	$shiftl$ `"abcde"` $\equiv$ `"bcdea"`
shiftr	$shiftr$ `"abcde"` $\equiv$ `"eabcd"`
sum	$sum\ [7,3,8,5] \equiv 23$
swap	$swap$ `"abcde"` $\equiv$ `"badce"`
switch	$switch$ `"abcde"` $\equiv$ `"ebcda"`
take	$take\ 3$ `"abcde"` $\equiv$ `"abc"`
weave	$weave$ `"abc" "def"` $\equiv$ `"adbecf"`

pragmatically it makes little sense to insist that an algorithm is quicker by some seconds if the example set is fine-tuned.

For this reason, the same set of I/O pairs as that included in Igor II$_H$ -0.7.1.2 was used for analytical synthesis, namely, Igor II$_H$ and the proposed system. That said, some I/O example sets bundled in Igor II$_H$ -0.7.1.2 are obviously inadequate in that they seem not to supply enough computational traces. In Section 4.3, it will be shown what number of examples is enough and not too big for the corrected sets of examples.

No background knowledge functions were used by Igor II$_H$ and the proposed system except the use of addition for the *fib* task.

Environment The experiments were conducted on one CPU core of the Intel® Xeon®CPU X3460 2.80 GHz. The source code was built with Glasgow Haskell Compiler Ver. 6.12.1 under the single processor setting.

Table 3. Benchmark results (left) and results for different number of I/O examples (right). "#exs." means the number of examples. Each number (except those below "#exs.") shows the execution time in seconds, rounded to the nearest integer. This is the time until the first program is obtained for MagicHaskeller and the proposed system. >300 represents that there was no answer in 5 minutes. Slashed-out numbers like $\not{0}$ mean that the result was wrong, that is, the behavior of the generated function to unspecified I/O pairs did not reflect the user's intension. ∞ means "impossible in theory" — this is only used for MagicHaskeller, when the requested function is a partial function without inhabited type and thus cannot be synthesized with the default primitive component set of MagicHaskeller.

	Igor II$_H$	MagH	proposed
addN	25	0	2
allodd	>300	4	>300
andL	0	0	1
concat	>300	3	>300
drop	>300	0	0
(≡)	3	22	0
evenpos	0	8	0
evens	$\not{0}$	93	>300
fib	>300	16	>300
head	0	∞	0
init	0	3	0
(++)	3	0	0
last	0	∞	0
lasts	0	35	0
lengths	$\not{1}$	1	0
multfst	0	4	0
multlst	0	1	0
oddpos	0	8	0
reverse	0	0	0
shiftl	0	4	0
shiftr	0	42	0
sum	>300	0	>300
swap	0	>300	0
switch	0	>300	0
take	0	7	0
weave	>300	142	0

name	#exs.	Igor II$_H$	proposed
addN	3	$\not{0}$	> 300
	6	$\not{0}$	7
	9	$\not{0}$	6
	12	$\not{0}$	2
	15	$\not{0}$	0
	18	35	3
	21	> 300	> 300
allodd	6	$\not{0}$	> 300
	10	$\not{0}$	0
	15	> 300	26
	21	> 300	> 300
andL	1	$\not{0}$	> 300
	3	$\not{0}$	0
	7	0	0
	15	0	1
	31	> 300	> 300
concat	3	$\not{0}$	0
	6	$\not{0}$	0
	9	$\not{0}$	0
	13	> 300	> 300
drop	4	$\not{0}$	0
	6	$\not{0}$	0
	9	> 300	0
	12	> 300	0

4.2 Efficiency evaluation

The first experiment compares the efficiency of the proposed system with that of other systems using the same I/O examples as those bundled in the Igor II$_H$ release in order to make sure that the proposed system does not sacrifice the efficiency.

Table 3 (left)shows the benchmark results under the condition described in the previous section.

Comparisons between analytical systems The proposed system successfully avoids generating wrong functions by generating many programs and filtering them with a test condition. For all the cases where Igor II$_H$ generated the wrong result, it

either returned a correct result or did not terminate. Since yielding a wrong result is just as misleading and no better than not yielding anything, at this point the proposed system is at least as good as IGOR II$_H$.

In addition, the proposed system is as fast as or faster than IGOR II$_H$ except when synthesizing *andL*, if the time required for human users to enter the test condition is ignored. The reason IGOR II$_H$ is quicker than the proposed system on *andL* is because it specializes defined function introduction to *direct call*s, or calls with target function arguments.

On the other hand, the main reason the proposed system was faster than IGOR II$_H$ is because a novel efficient algorithm for trying to match many expressions at once, which was presented in Section 3.6, was developed. This algorithm does not have a direct connection with Spivey's monad and could be applied to IGOR II$_H$.

Comparison with MagicHaskeller When there are some case partitionings, MAGICHASKELLER tends to require more computation than analytical systems, which is why it cannot generate *swap* or *switch* in five minutes. Although both analytical systems and MAGICHASKELLER prioritize the search based on some cost functions, current versions of MAGICHASKELLER define the cost of a function as the number of function and constructor applications in the curried form, and thus having some functions with a bigger arity (like case functions) results in less priority. The cost function of MAGICHASKELLER may have room for tuning.

Also, MAGICHASKELLER with the default component library cannot generate partial functions without inhabited types such as $head :: forall\ a.\ [a] \to a$ and $last :: forall\ a.\ [a] \to a$.

On the other hand, since analytical systems cannot generate tail-recursive functions, they generate such functions in their linear recursive form. This sometimes results in unnecessarily complicated function definitions.

These facts could be suggested from benchmark results on conventional systems. However, by comparing IGOR II$_H$ and MAGICHASKELLER with an analytical system implemented in the same way as MAGICHASKELLER , it has become even clearer whether each difference is due to that in the implementation or that in the paradigm. Now that it has been shown that there is an obvious difference in the strengths and weakness of both approaches, a fusion of both approaches will hopefully improve the overall performance.

4.3 Robustness to changes in I/O examples

The main purpose for adding a generate-and-test aspect to the analytical IFP is to obtain a system that works as expected for a variety of I/O example sets. In this section, the robustness of the proposed system to variation in the number of I/O example pairs is empirically evaluated in comparison with that of IGOR II$_H$.

In this experiment, the raw sets of I/O examples from the IGOR II$_H$ release were not used; rather, an edited version with enough computational traces was used, since several sets are tested for each target function. When n I/O example pairs are required, the first n examples of the longest set of I/O examples are used. For example, Table 4 shows the set of I/O examples used for synthesis of *addN*;

Table 4. Set of I/O examples of *addN* used for evaluating the robustness of the analytical systems.

$addN :: Int \to [Int] \to [Int]$		$addN\ 1\ [1] = [2]$	
$addN\ 0\ [] = []$		$addN\ 1\ [2] = [3]$	-- 12 examples
$addN\ 1\ [] = []$		$addN\ 1\ [0,0] = [1,1]$	
$addN\ 2\ [] = []$	-- 3 examples	$addN\ 1\ [0,1] = [1,2]$	
$addN\ 0\ [0] = [0]$		$addN\ 1\ [1,0] = [2,1]$	-- 15 examples
$addN\ 0\ [1] = [1]$		$addN\ 2\ [0] = [2]$	
$addN\ 0\ [2] = [2]$	-- 6 examples	$addN\ 2\ [1] = [3]$	
$addN\ 0\ [0,0] = [0,0]$		$addN\ 2\ [2] = [4]$	-- 18 examples
$addN\ 0\ [0,1] = [0,1]$		$addN\ 2\ [0,0] = [2,2]$	
$addN\ 0\ [1,0] = [1,0]$	-- 9 examples	$addN\ 2\ [0,1] = [2,3]$	
$addN\ 1\ [0] = [1]$		$addN\ 2\ [1,0] = [3,2]$	-- 21 examples

when synthesizing from six I/O example pairs the lines from $addN\ 0\ [] = []$ to $addN\ 0\ [2] = [2]$ (and the line for the type signature) are used.

This experiment was only performed for the first five functions. Other conditions are the same as those in the previous section.

Table 3 (right)shows the results of the experiments. The results clearly show the merit of the proposed system, where, especially for *addN*, *andL*, *concat*, and *drop* the proposed system correctly generated a desired program from 3 or 6 examples and the test function, while IGOR II$_H$ is satisfied with the most simple program explaining the analyzed I/O example pairs and does not synthesize expected functions from a small set of examples. As can be seen from Table 4, the first 6 examples of *addN* simply return the second argument, and therefore IGOR II cannot do that even if a test process is added. On the other hand, the proposed system, residing between IGOR II$_H$ and MAGICHASKELLER, generates a desired program even in such a hard situation.

5 Conclusions and Future Work

An analytical IFP algorithm that can generate a stream of programs that generalize the given specification from the simplest to the least simple, instead of just generating the simplest program(s), was created.

By adding the generate-and-test feature to analytical synthesis, this algorithm solved the trade-off between the efficiency and the accuracy IGOR II had been suffering from. As a result, a desired program can be obtained without giving many I/O example pairs, and some functions that could not be synthesized analytically have become able to be synthesized.

In addition, by making the implementation of the new analytical IFP algorithm closer to that of MAGICHASKELLER, it has now become clear that both analytical and generate-and-test approaches have different strong points. This suggests that a complementary fusion of both approaches should be promising. As well, the threshold to fusing both approaches has been lowered. One option for fusing the software could be by adding a new operator that generates subexpressions by using MAGICHASKELLER and filters them by their I/O examples.

As for the efficiency, the new implementation is quicker than IGOR II$_H$ in most cases. However, as mentioned in Section 2.1, it should be noted that this result is not compared with the newest IGOR II$^+$. Further efficiency improvements using catamorphism or paramorphism introduction remain for future work.

Acknowledgements

Dr. Martin Hofmann kindly introduced the author to the implementation details of IGOR II$_H$ and answered the author's many questions. This work was supported by JSPS KAKENHI 21650032.

References

1. Kitzelmann, E.: Data-driven induction of recursive functions from input/output-examples. In: AAIP'07: Proceedings of the Workshop on Approaches and Applications of Inductive Programming. (2007) 15–26
2. Katayama, S.: Systematic search for lambda expressions. In: Sixth Symposium on Trends in Functional Programming. (2005) 195–205
3. Olsson, R.: Inductive functional programming using incremental program transformation. Artificial Intelligence **74**(1) (1995) 55–81
4. Hofmann, M., Kitzelmann, E., Schmid, U.: Porting IgorII from maude to haskell. In Schmid, U., Kitzelmann, E., Plasmeijer, R., eds.: Approaches and Applications of Inductive Programming, Third International Workshop, AAIP 2009. Volume 5812 of LNCS. (2010) 140–158
5. Spivey, J.M.: Algebras for combinatorial search. Journal of Functional Programming **19** (July 2009) 469–487
6. Hofmann, M., Kitzelmann, E.: 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 (2010) 93–100
7. Katayama, S.: Power of brute-force search in strongly-typed inductive functional programming automation. In: PRICAI 2004: Trends in Artificial Intelligence. Volume 3157 of LNAI., Springer-Verlag (August 2004) 75–84
8. Katayama, S.: Systematic search for lambda expressions. In: Trends in Functional Programming. Volume 6., Intellect (2007) 111–126
9. Katayama, S.: Recent improvements of MagicHaskeller. In Schmid, U., Kitzelmann, E., Plasmeijer, R., eds.: Approaches and Applications of Inductive Programming, Third International Workshop, AAIP 2009. Volume 5812 of LNCS. (2010) 174–193
10. Hofmann, M., Kitzelmann, E., Schmid, U.: A unifying framework for analysis and evaluation of inductive programming systems. In: Proceedings of the Second Conference on Artificial General Intelligence. (2009)
11. Spivey, J.M.: Combinators for breadth-first search. Journal of Functional Programming **10**(4) (2000) 397–408
12. Barendregt, H.: Lambda calculi with types. In Abramsky, S., Gabbay, D.M., Maibaum, T.S.E., eds.: Handbook of Logic in Computer Science. Volume 2., Oxford University Press (1992) 117–309
13. Hinze, R.: Generalizing generalized tries. Journal of Functional Programming **10**(4) (2000) 327–351