An Analytical Inductive Functional Programming System that Avoids Unintended Programs *

Susumu Katayama
University of Miyazaki
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. The analytical approach shows greater promise for application to greater problems because the search space is restricted by the given example set, but it requires much more examples written in order to yield results that reflect the user's intention, which is bothersome and causes the algorithm to slow down. On the other hand, the generate-and-test approach does not require long description of input-output examples, but does not restrict the search space using the example set. This paper proposes a new approach taking the best of the two, called "analytically-generate-and-test approach", which is based on analytical generation and testing of many program candidates. For generating many candidate programs, the proposed system uses a new variant of IGOR II, the exemplary analytical inductive functional programming algorithm. This new system preserves the efficiency features of analytical approaches, while minimizing the possibility of generating unintended programs even when using fewer input-output examples.

Categories and Subject Descriptors I.2.2 [*ARTIFICIAL INTELLIGENCE*]: Automatic Programming; D.1.1 [*PROGRAMMING TECHNIQUES*]: Applicative (Functional) Programming

General Terms Algorithms

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., the IGOR II system [15][8]), and the other is the generate-and-test approach that generates many programs and selects those that satisfy the given condition (e.g., the MAGICHASKELLER system [10] and the ADATE system [18]). Analytical methods can use the information of I/O examples for limiting the size of the search space for efficiency purposes, but they have limitations on how to provide I/O example pairs as the specification, i.e., beginning with the simplest input(s) and progressively increasing their complexity. They often require many lines of examples because an example for more or less complex inputs is indispensable for specifying their users' intention correctly. This requirement not only makes the algorithm less attractive especially for synthesis of small programs by lowering its usability, but also causes the algorithm to slow down. 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 there are some cases where analytical systems outperform generate-and-test systems, as we shall see in Section 4.2.

This paper proposes yet another approach, named analytically-generate-and-test approach, which can use the I/O examples for guiding search but still does not require many examples for specifying users' intentions correctly. The idea is to break the set of I/O examples into a small set for guiding search and another small set for specifying the users' intentions, and to use the former for analytically generating a prioritized infinite stream of programs, and then to use the latter for filtering those that reflects the users' intention (Figure 1). Because the latter set is separate from the former, complex examples can be included in the latter set without increasing the size of the former set.

Why filtration by a complex example is effective for reflecting the users' intention is 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

* This paper extends the short unreviewed versions [14][13]

PEPM'12, January 23–24, 2012, Philadelphia, PA, USA.

[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.

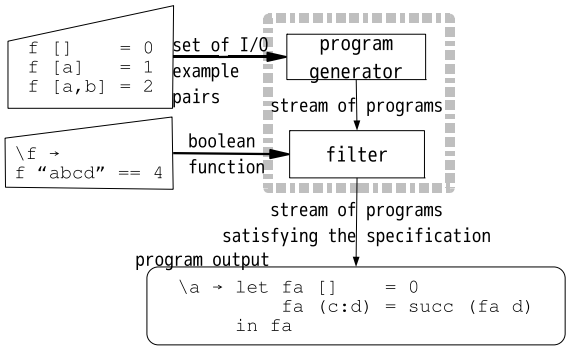

Figure 1. The structure of the proposed system.

$$
\begin{aligned}
&reverse\ [\,] &&= [\,]\\
&reverse\ [a] &&= [a]\\
&reverse\ [a,b] &&= [b,a]\\
&reverse\ [a,b,c] &&= [c,b,a]\\
&reverse\ [a,b,c,d] &&= [d,c,b,a]
\end{aligned}
$$

Table 1. Example of the input source text for synthesizing the $reverse$ function using IGOR II$_H$ (taken from Version 0.7.1.3 of IGOR II$_H$ release)

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 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, the same effect applies 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.

2. Background

The proposed analytically-generate-and-test IFP system generates a (usually infinite) stream of many programs analytically and selects those that satisfy the condition given separately by the user. Since existing analytical synthesis systems such as IGOR II generate only one or a few best programs, a new IGOR II-variant algorithm that generates a stream of programs is invented, which is inspired by the way MAGICHASKELLER generates an infinite stream of programs.

[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 synthesize a function which differs from $reverse$ 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.

This section explains the background of this research by firstly positioning IFP among related research fields and then introducing IGOR II and MAGICHASKELLER.

2.1 Inductive functional programming

IFP deals with generation of recursive programs from ambiguous specifications through generalization. This means the challenge is not generating "interesting" programs, but how the generalization reflects the intention of the specification writer. Being able to deal with many different specifications is another important factor, because program synthesis is not always possible for existing algorithms when complicated specifications are given. The current state-of-the-art IFP systems often succeed in synthesizing functions which cannot easily be written by unskilled programmers, though they cannot synthesize programs consisting of many lines.

IFP solves different problems from "deductive" automatic programming that transforms (or compiles) specifications without ambiguity into programs. Program transformation systems (e.g. [19]) and semi-automatic proof systems (e.g. [1]) are examples of deductive program synthesizers. Ambiguous specifications can be given to proof systems, but they do not automatically induce the general rules that cannot be deduced from the given specifications.

Inductive logic programming (ILP) (e.g. [17]) that synthesizes logic programs by inductive inference is related to IFP. The main advantage of IFP is that it is better at synthesizing recursive functions than ILP which requires desperately many negative examples for synthesizing recursive functions (e.g. [5]).

Programming by demonstration (PBD) [2] and programming by example (e.g. [3]) are also important relatives of IFP. They synthesize generalized programs from computational traces and/or I/O examples exploiting domain-specific knowledge. The SMARTpython system [16] applies version space algebra to general imperative programming tasks. Its limitation in comparison with IFP is that the full information of program state transition has to be made available as the computational trace.

Programming by sketch (e.g. [20]) is another interesting synthesis framework, where the synthesizer finds a finite program that is semantically equivalent to a given function definition and at the same time matches to a given template. Although its problem framework is different from that of IFP, its process of repeatedly limiting the search space with auto-generated negative examples is that of inductive programming, and its algorithm might be applied to more general synthesis tasks.

Genetic programming (GP) is another approach to program synthesis. Synthesis of recursive programs has not been a main target of most GP systems, though there have been some attempts at it (e.g. [23]). The ADATE system [18] can be categorized as a GP system in a wide sense (though it does not conduct crossovers).

2.2 IGOR II

The algorithm behind IGOR II[15] 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$_H$[8], which is a simple port, and IGOR II$^+$ [6], which is an extension with support of catamorphism/paramorphism introduction. Such support is reportedly 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 appear in the output but do not appear in the input represent unfinished terms.

2. Try the following operators[4] in order to complete the unfinished terms.

   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 it introduces new functions to all fields, and supply the same set of arguments as 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 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.

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

   Then, expressions with the least cost are kept, and others are abandoned. The cost function will be explained in Section 2.2.2.

2.2.1 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 a recursive program.
- 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 by enabling to specify the big example as the test function.

2.2.2 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 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.

2.2.3 Synthesizing total functions

IGOR II offers two termination checkers that ensure termination of synthesized programs on the given example inputs, and its users can choose one of them. The idea behind both of them is rather simple: a recursive call to function f may not occur unless the example argument set supplied when f is making a recursive call is "smaller" than that supplied when f was created.

The two termination checkers differ in the definition of the size. One is just based on the total number of the constructors in all arguments; the other is the default, and is based on the lexicographical order of the argument vector, comparing each argument by the number of constructors in it.

2.3 MAGICHASKELLER

MAGICHASKELLER[9][11][12] is a generate-and-test method based on systematic search. One of its design policies states "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 do not always mean that an algorithm can synthesize big programs, unless the heuristics are designed adequately and work well. According to benchmarks from the literature [7] and inductive-programming.org[6], at least it can be claimed that MAGICHASKELLER performs well compared with other methods.

Figure 2 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 component library, as 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.

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

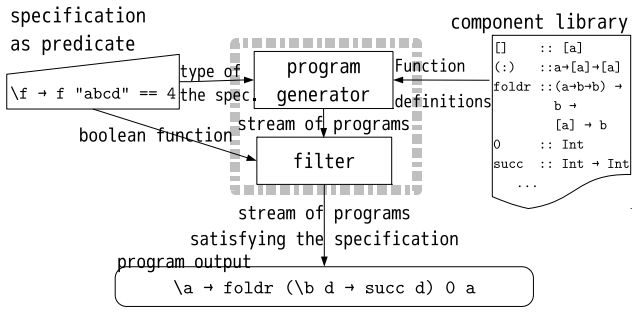

Figure 2. The structure of MAGICHASKELLER.

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 [10] is essentially an extension of an automatic prover algorithm that can generate infinite number of proofs exhaustively[12]. 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 [21]. 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[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 $head :: forall\ a.\ [a] \to a$.

Early versions of MAGICHASKELLER try to detect and prune as many semantically equivalent expressions as possible by applying known optimization rules[10]. 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

This section describes the analytical IFP algorithm that generates a stream of programs, which is necessary for implementing the proposed analytically-generate-and-test system. In this paper, this algorithm for just generating a stream of programs is called "the proposed algorithm", and the whole system including the testing is called "the proposed system".

The algorithm presented here is simplified to some extent. The algorithm used for evaluation infers types while generating programs in order to narrow the search space, though this part is omitted in this paper.

IGOR II generates functional programs as constructor term-rewriting systems that consist of linear rules. The proposed algorithm generates single λ-expressions, however, because it has turned out that direct generation of corresponding λ-expressions is simpler to implement and to explain. The proposed algorithm generates λ-expressions with constructors and a fixpoint combinator defined in Table 2. Taking a set of I/O examples $\mathbf{d}^n$ representing the target function and possibly a set of background knowledge functions Γ, the algorithm returns a prioritized stream of recursive functions, or es.

The grammar defines functions in the uncurried form. Tuples are assumed to be catenable: e.g., for

$$\begin{aligned}\mathbf{u}^l &= (u_1, ..., u_l)\\ \mathbf{v}^m &= (v_1, ..., v_m)\end{aligned}$$

the following equations hold:

$$\begin{aligned}\left(\mathbf{u}^l, w, \mathbf{v}^m\right) &= (u_1, ..., u_l, w, v_1, ..., v_m)\\ \left(\mathbf{u}^l, \mathbf{v}^m\right) &= (u_1, ..., u_l, v_1, ..., v_m)\end{aligned}$$

The grammar provides a compact and convenient way of writing the relation between I/O example sets and the induced function. It identifies functions with sets of I/O pairs, thus

$$f\mathbf{p}_1 = q_1\ \wedge\ ...\ \wedge\ f\mathbf{p}_n = q_n \quad \Leftrightarrow \quad f \supset \left\{ \begin{array}{c} \mathbf{p}_1 \mapsto q_1 \\ \vdots \\ \mathbf{p}_n \mapsto q_n \end{array} \right\}$$

holds.

The set of rules for recursively generating generalized functions based on the grammar follows in this section. Each generation rule is formalized as a sequent rule. Each sequent is in the form of

$$\Gamma \vdash e \supset \mathbf{d} \qquad (\text{or} \quad \Gamma \vdash e' \supset \mathbf{d}\ )$$

which means "e (or e') is a function which uses the set of available functions Γ and generalizes (or is induced from) $\mathbf{d}$". Since "e generalizes $\mathbf{d}$" is a kind of restriction put into e, it can be put in the same light as the type of e. Likewise, each sequent rule can be regarded as a typing rule.

Because the proposed algorithm obtains a stream of generalized recursive programs es from a given set of I/O examples $\mathbf{d}$, what it does corresponds to finding expressions with a given type, and finding proofs for a given predicate under Curry-Howard isomorphism. Thus, the proposed algorithm generates proof trees by following sequent rules upward.

Once these sequent rules are defined, the above can be implemented monadically, using Spivey's interface for combinatorial search[21][22]: if there are more than one rules to which the current sequent matches, all of them are tried and the results are combined with the monoidal addition operator $\oplus$; if a rule has more than one premise, they are sequentially tried in the lifted manner. This is essentially the same way as MAGICHASKELLER is implemented[12]. The interface includes a function $wrap$ that lowers the priority of the current search, which is inserted to the case partitioning operator and the defined function introduction operator in order to avoid infinite loops.

3.1 Generation of es

According to the grammar defined in Table 2, generation of e generalizing the I/O example set $\mathbf{d}$ involves generation of e', where recursive calls to the function e itself are additionally available within e. Thus, the `Fix` rule in Figure 3 holds.

3.2 Generation of e's

Generation of e's is also done based on Table 2. The careful reader should have noticed that constructor expressions, function applications, and case expressions can be introduced by corresponding operators: constructor introduction, defined function introduction, and case partitioning, respectively. Projections do not have their direct counterpart operator; this is because induction of projection function is part of antiunification. More precisely, antiunification has

C^n	$\in$	$\mathcal{C}^n$	n-ary constructors
f^n	$\in$	$\mathcal{F}^n$	n-ary function variables
u, v, w	$\in$	$\mathcal{V}$	non-functional variables
$\mathbf{u}^n, \mathbf{v}^n, \mathbf{w}^n$	$::=$	$(u_1, ..., u_n)$	tuples of n variables (The order matters.)
p, q, r	$::=$	$u \mid C^n\ \mathbf{p}^n$	patterns
$\mathbf{p}^n, \mathbf{q}^n, \mathbf{r}^n$	$::=$	$(p_1, ..., p_n)$	tuples of n patterns (The order matters.)
d^n	$::=$	$\mathbf{p}^n \mapsto p$	I/O pairs
$\mathbf{d}^n$	$::=$	$\left\{ \begin{array}{l} d_1^n \\ \vdots \\ d_m^n \end{array} \right\}$	n-ary functions as sets of I/O pairs (The order does not matter.)
γ	$::=$	$f^n \supset \mathbf{d}^n$	named functions
Γ	$::=$	$\gamma_1, ..., \gamma_n$	sets of named functions (The order does not matter.)
e	$::=$	$\mathbf{fix}(\lambda f.e')$	generalized (possibly) recursive functions
e'	$::=$	$\lambda \mathbf{u}^n.\ u$	projections
	$\mid$	$\lambda \mathbf{u}^n.\ C^k\ (e_1 \mathbf{u}^n, ..., e_k \mathbf{u}^n)$	constructor expressions
	$\mid$	$\lambda \mathbf{u}^n.\ f^k\ (e_1 \mathbf{u}^n, ..., e_k \mathbf{u}^n)$	function applications
	$\mid$	$\lambda \mathbf{u}^n.\ \mathbf{case}\ u\ \mathbf{of}$	
		$C_1^{k_1} \mathbf{v}_1^{k_1} \rightarrow e_1(\mathbf{u}^n, \mathbf{v}_1^{k_1})$	
		$\vdots$	case expressions
		$C_m^{k_m} \mathbf{v}_m^{k_m} \rightarrow e_m(\mathbf{u}^n, \mathbf{v}_m^{k_m})$	

Table 2. The grammar. $\mathbf{fix}$ will be interpreted as a fixpoint combinator. n's in $\mathbf{p}^n$, d^n, $\mathbf{d}^n$, etc. can be omitted if not ambiguous. Subscripted variable names such as u_i have the same type as its unsubscripted version u, but this does not apply to primed names like e'.

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 for the latter a new operator **projection function introduction** is invented which finalizes the synthesis of the current subexpression by finding an argument whose examples equal the return value examples.

What follow are the formal definitions and detailed descriptions of those operators.

Projection function introduction is the simplest operator that induces a projection function by identifying an argument which is always equal to the return value. From the I/O relation

$$\forall m \in \{1...M\}.\ t(p_{m1}, ..., p_{mn}, ..., p_{mN}) = p_{mn}$$

the operator induces the target function t as

$$\forall v_1 ... v_N.\ t(v_1, ..., v_n, ..., v_N) = v_n$$

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

The `Proj` rule in Figure 3 corresponds to this operator.

Constructor introduction extracts a common constructor among the output examples of the I/O pairs of the given target function t. From the I/O relation

$$\forall m \in \{1...M\}.\ t(p_{m1}, ..., p_{mN}) = C^K(q_{m1}, ..., q_{mK})$$

it induces

$$\forall v_1 ... v_N.\ t(v_1, ..., v_N) = C^K(e_1(v_1, ..., v_N), ..., e_K(v_1, ..., v_N))$$

where

$$\forall k \in \{1...K\}. \forall m \in \{1...M\}. e_k(p_{m1}, ..., p_{mN}) = q_{mk} \quad (1)$$

Further induction of $e_1 ... e_K$ from the newly introduced I/O pairs shown in Equation 1 is required unless $K = 0$.

The `Constr` rule in Figure 3 corresponds to this operator.

In practice, C^K 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, where $+$ is not a constructor but a library function.

Defined function introduction matches the output examples of the target function t to those of a defined function f. From the I/O relations

$$\forall m \in \{1...M\}. \quad t(p_{m1}, ..., p_{mN}) = \theta_m(q_{a_m}) \quad (2)$$

$$\forall l \in \{1...L\}. \quad f^K(r_{l1}, ..., r_{lK}) = q_l \quad (3)$$

for existing substitutions $\theta_1 ... \theta_M$ and assignments $a_1 ... a_M \in \{1...L\}$, the operator induces

$$\forall v_1 ... v_N.\ t(v_1, ..., v_N) = f^K(e_1(v_1, ..., v_N), ..., e_K(v_1, ..., v_N)) \quad (4)$$

where

$$\forall k \in \{1...K\}. \forall m \in \{1...M\}.\ e_k(p_{m1}, ..., p_{mN}) = \theta_m(r_{a_m k}) \quad (5)$$

A substitution is a set of replacements that replace variables with patterns. The idea behind this operator is to try to match the output of the target function to that of each rule of f, i.e., q_l. If that succeeds, the Equation 3 is rewritten using the required substitution θ_m as

$$f(\theta_m(r_{a_m 1}), ..., \theta_m(r_{a_m K})) = \theta_m(q_{a_m})$$

Thus, with Equation 2,

$$t(p_{m1}, ..., p_{mN}) = f(\theta_m(r_{a_m 1}), ..., \theta_m(r_{a_m K}))$$

is obtained. Equation 5 follows comparing this equation with Equation 4.

Further inference of $e_1 ... e_K$ from the newly introduced I/O pairs is required. This operator is tried for each selection of defined function f and for each selection of $a_m|_{m \in \{1...M\}}$. Then, when making a recursive call rather than calling a background knowledge function, the termination checker checks if f is called with a "smaller" argument list in a well-formed sense than when f was first called, i.e., if

$$\forall m \in \{1...M\}.\ (\theta_m(r_{a_m 1}), ..., \theta_m(r_{a_m K})) < (p_{m1}, ..., p_{mK})$$

$$\frac{\Gamma, f^k \supset \mathbf{d}^k \vdash e' \supset \mathbf{d}^k}{\Gamma \vdash \mathbf{fix}(\lambda f^k.e') \supset \mathbf{d}^k}\ \texttt{Fix} \qquad \frac{}{\Gamma \vdash \lambda\left(\mathbf{u}^i, u, \mathbf{v}^j\right).u \supset \left\{\begin{array}{c}\left(\mathbf{p}_1^i, p_1, \mathbf{q}_1^j\right) \mapsto p_1 \\ \vdots \\ \left(\mathbf{p}_M^i, p_M, \mathbf{q}_M^j\right) \mapsto p_M\end{array}\right\}}\ \texttt{Proj}$$

$$\frac{\Gamma \vdash e_1 \supset \left\{\begin{array}{c}\mathbf{p}_1^N \mapsto p_{11} \\ \vdots \\ \mathbf{p}_M^N \mapsto p_{M1}\end{array}\right\} \quad ... \quad \Gamma \vdash e_K \supset \left\{\begin{array}{c}\mathbf{p}_1^N \mapsto p_{1K} \\ \vdots \\ \mathbf{p}_M^N \mapsto p_{MK}\end{array}\right\}}{\Gamma \vdash \lambda\mathbf{u}^N.C^K\left(e_1\mathbf{u}^N, ..., e_K\mathbf{u}^N\right) \supset \left\{\begin{array}{c}\mathbf{p}_1^N \mapsto C^K(p_{11}, ..., p_{1K}) \\ \vdots \\ \mathbf{p}_M^N \mapsto C^K(p_{M1}, ..., p_{MK})\end{array}\right\}}\ \texttt{Constr}$$

$$\frac{\Gamma \vdash e_1 \supset \left\{\begin{array}{c}\mathbf{p}_1^N \mapsto \theta_1(p_{a_1 1}) \\ \vdots \\ \mathbf{p}_M^N \mapsto \theta_M(p_{a_M 1})\end{array}\right\} \quad ... \quad \Gamma \vdash e_K \supset \left\{\begin{array}{c}\mathbf{p}_1^N \mapsto \theta_1(p_{a_1 K}) \\ \vdots \\ \mathbf{p}_M^N \mapsto \theta_M(p_{a_M K})\end{array}\right\}}{\Gamma \vdash \lambda\mathbf{u}^N.f^K\left(e_1\mathbf{u}^N, ..., e_K\mathbf{u}^N\right) \supset \left\{\begin{array}{c}\mathbf{p}_1^N \mapsto \theta_1(q_{a_1}) \\ \vdots \\ \mathbf{p}_M^N \mapsto \theta_M(q_{a_M})\end{array}\right\}}\ \texttt{Def'd}$$

$$\text{where } \left(f^K \supset \left\{\begin{array}{c}(p_{11}, ..., p_{1K}) \mapsto q_1 \\ \vdots \\ (p_{L1}, ..., p_{LK}) \mapsto q_L\end{array}\right\}\right) \in \Gamma \text{ and } \begin{array}{llll}(\theta_1(p_{a_1 1}), & ..., & \theta_1(p_{a_1 K})) & < \mathbf{p}_1^K \\ & & \vdots & \\ (\theta_M(p_{a_M 1}), & ..., & \theta_M(p_{a_M K})) & < \mathbf{p}_M^K\end{array}$$

$$\frac{\Gamma \vdash e_1 \supset \left\{\begin{array}{ll}\left(\mathbf{p}_{11}^i,\ C_1^{K_1}\mathbf{r}_1^{K_1}, \mathbf{q}_{11}^j,\ \mathbf{r}_1^{K_1}\right) & \mapsto p_{11} \\ \vdots & \vdots \\ \left(\mathbf{p}_{1L_1}^i, C_1^{K_1}\mathbf{r}_{L_1}^{K_1}, \mathbf{q}_{1L_1}^j, \mathbf{r}_{L_1}^{K_1}\right) & \mapsto p_{1L_1}\end{array}\right\} \ ... \ \Gamma \vdash e_M \supset \left\{\begin{array}{ll}\left(\mathbf{p}_{M1}^i,\ C_M^{K_M}\mathbf{r}_1^{K_M}, \mathbf{q}_{M1}^j,\ \mathbf{r}_1^{K_M}\right) & \mapsto p_{M1} \\ \vdots & \vdots \\ \left(\mathbf{p}_{ML_M}^i, C_M^{K_M}\mathbf{r}_{L_M}^{K_M}, \mathbf{q}_{ML_M}^j, \mathbf{r}_{L_M}^{K_M}\right) & \mapsto p_{ML_M}\end{array}\right\}}{\Gamma \vdash \begin{array}{ll}\lambda\left(\mathbf{u}^i, u, \mathbf{v}^j\right).\ \mathbf{case}\ u\ \mathbf{of} & \\ C_1^{K_1}\mathbf{w}^{K_1} \rightarrow & e_1\left(\mathbf{u}^i, u, \mathbf{v}^j, \mathbf{w}^{K_1}\right) \\ \vdots & \vdots \\ C_M^{K_M}\mathbf{w}^{K_M} \rightarrow & e_M\left(\mathbf{u}^i, u, \mathbf{v}^j, \mathbf{w}^{K_M}\right)\end{array} \supset \left\{\begin{array}{lll}\left(\mathbf{p}_{11}^i,\ C_1^{K_1}\mathbf{r}_1^{K_1},\ \mathbf{q}_{11}^j\right) & \mapsto & p_{11} \\ \vdots & \vdots & \vdots \\ \left(\mathbf{p}_{1L_1}^i, C_1^{K_1}\mathbf{r}_{L_1}^{K_1}, \mathbf{q}_{1L_1}^j\right) & \mapsto & p_{1L_1} \\ & \vdots & \\ \left(\mathbf{p}_{M1}^i,\ C_M^{K_M}\mathbf{r}_1^{K_M},\ \mathbf{q}_{M1}^j\right) & \mapsto & p_{M1} \\ \vdots & \vdots & \vdots \\ \left(\mathbf{p}_{ML_M}^i, C_M^{K_M}\mathbf{r}_{L_M}^{K_M}, \mathbf{q}_{ML_M}^j\right) & \mapsto & p_{ML_M}\end{array}\right\}}\ \texttt{Case}$$

Figure 3. Generation rules as sequent rules.

holds for some well formed order $\leq$. The proposed algorithm uses the same order as IGOR II$_H$'s default.

The `Def'd` rule in Figure 3 corresponds to this operator. $\mathbf{p}_m^K$ in its "where" clause means the first K of $\mathbf{p}_m^N$

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

$$\forall m \in \{1...M\}.\forall l \in \{1...L_m\}.$$
$$t(p_{ml1}, ..., p_{ml(n-1)}, p_{mln}, p_{ml(n+1)}, ..., p_{mlN}) = q_{ml}$$

where

$$p_{mln} = C_m^{K_m}(r_{l1}, ..., r_{lK_m})$$

the operator infers

$$\forall m \in \{1...M\}.\forall v_1...v_{n-1}.\forall v_{n+1}...v_N.\forall u_1...u_{K_m}.$$
$$t(v_1, ..., v_{n-1}, C_m^{K_m}(u_1, ..., u_{K_m}), v_{n+1}, ..., v_N)$$
$$= e_m(v_1, ..., v_{n-1}, C_m^{K_m}(u_1, ..., u_{K_m}), v_{n+1}, ..., v_N,$$
$$u_1, ..., u_{K_m})$$

where

$$\forall m \in \{1...M\}.\forall l \in \{1...L_m\}.$$
$$e_m(p_{ml1}, ..., p_{mlN}, r_{l1}, ..., r_{lK_m}) = q_{ml}$$

Further inference of $e_1...e_M$ from the newly introduced I/O pairs is required. This operator is tried for each argument $n \in \{1..N\}$. In the above case, there are $\sum_{m=1}^{M} L_m$ examples, and they are categorized into M cases based on the constructor at the nth input. The mth case is characterized by the K_m-ary constructor $C_m^{K_m}$, and has L_m examples.

The `Case` rule in Figure 3 corresponds to this operator.

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 IIis 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.

In addition, this definition is slightly different even from the implemented one in that the n-th argument of e_m, namely p_{mln}, is actually hidden for efficiency reasons when further inferring e_m, though the rule shown in this paper does not hide it in order to simplify termination checking.

3.3 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 [4] 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. 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.

An implementation of the proposed system is now released as a part of MAGICHASKELLER, which is available from Hackage, a collection of Haskell package releases[7]. However, in this paper "MAGICHASKELLER" means its conventional part conducting exhaustive search.

4.1 Experiment conditions

4.1.1 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 ([7] and inductive-programming.org [8]).

As for the search monad for the proposed system, based on preliminary experiments, Spivey's monad for breadth-first search [21] was selected over other alternatives that fit into Spivey [22]'s interface, such as depth-bound search and their recomputing variants, such as the *Recomp* monad [10].

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.

4.1.2 Set of programming tasks

Table 3 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.

The left column of Table 3 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, 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 but 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.

[7] `http://hackage.haskell.org/package/MagicHaskeller`

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

name	test function	expected behavior
addN	$addN\ 3\ [5,7,2] \equiv [8,10,5]$	$addN\ n = map\ (n+)$
allodd	$allodd\ [3,3] \wedge not\ (allodd\ [2,3]) \wedge allodd\ [1,3,5] \wedge not\ (allodd\ [3,7,5,1,2])$	$allodd = all\ odd$
andL	$not\ (andL\ [True, False]) \wedge andL\ [True, True] \wedge andL\ [True, True, True] \wedge not\ (andL\ [False, True, True])$	$andL = foldr\ (\wedge)\ []$
concat	$concat$ `["abc","","de","fghi"]` $\equiv$ `"abcdefghi"`	same as $Prelude.concat$
drop	$drop\ 3$ `"abcde"` $\equiv$ `"de"`	same as $Prelude.drop$
$(\equiv)$	$3 \equiv 3 \wedge not\ (4 \equiv 6) \wedge 0 \equiv 0 \wedge not\ (2 \equiv 0) \wedge not\ (0 \equiv 2) \wedge not\ (3 \equiv 5)$	same as $Prelude.\equiv$
evenpos	$evenpos$ `"abcdefg"` $\equiv$ `"bdf"`	collect the $2n$th elements
evens	$evens\ [4,6,9,2,3,8,8] \equiv [4,6,2,8,8]$	$evens = filter\ even$
fib	$fib\ 0 \equiv 1 \wedge fib\ 1 \equiv 1 \wedge fib\ 3 \equiv 3 \wedge fib\ 5 \equiv 8 \wedge fib\ 7 \equiv 21$	$fib\ n$ is the nth Fibonacci number.
head	$head$ `"abcde"` $\equiv$ `'a'`	same as $Prelude.head$
init	$init$ `"foobar"` $\equiv$ `"fooba"`	same as $Prelude.init$
$(+\!\!+)$	`"foo"` $+\!\!+$ `"bar"` $\equiv$ `"foobar"`	same as $Prelude.+\!\!+$
last	$last$ `"abcde"` $\equiv$ `'e'`	same as $Prelude.last$
lasts	$lasts$ `["abcdef","abc","abcde"]` $\equiv$ `"fce"`	$lasts = map\ last$
lengths	$lengths$ `["abcdef","abc","abcde"]` $\equiv [6,3,5]$	$lengths = map\ length$
multfst	$multfst$ `"abcdef"` $\equiv$ `"aaaaaa"`	$multfst\ xs = map\ (\_ \rightarrow head\ xs)\ xs$
multlst	$multlst$ `"abcdef"` $\equiv$ `"ffffff"`	$multlst\ xs = map\ (\_ \rightarrow last\ xs)\ xs$
negateAll	$negateAll\ [False, True, False] \equiv [True, False, True] \wedge negateAll\ [True, False, False, True] \equiv [False, True, True, False]$	$negateAll = map\ not$
oddpos	$oddpos$ `"abcdef"` $\equiv$ `"ace"` $\wedge$ $oddpos$ `"abc"` $\equiv$ `"ac"`	collect the $2n+1$st elements
reverse	$reverse$ `"abcde"` $\equiv$ `"edcba"`	same as $Prelude.reverse$
shiftl	$shiftl$ `"abcde"` $\equiv$ `"bcdea"`	$shiftl\ xs = tail\ xs +\!\!+ [head\ xs]$
shiftr	$shiftr$ `"abcde"` $\equiv$ `"eabcd"`	$shiftr\ xs = last\ xs : init\ xs$
sum	$sum\ [7,3,8,5] \equiv 23$	same as $Prelude.sum$
swap	$swap$ `"abcde"` $\equiv$ `"badce"`	swap consecutive pairs of elements
switch	$switch$ `"abcde"` $\equiv$ `"ebcda"`	swap the first and the last elements
take	$take\ 3$ `"abcde"` $\equiv$ `"abc"`	same as $Prelude.take$
weave	$weave$ `"abc" "def"` $\equiv$ `"adbecf"`	merge two lists alternately

Table 3. Test functions for target functions used to filter results from MAGICHASKELLER and the proposed system. The λ-abstraction part of each function is omitted. Thus, for example, the test functions for $addN$ and $(+\!\!+)$ are $\lambda addN \rightarrow addN\ 3\ [5,7,2] \equiv [8,10,5]$ and $\lambda(+\!\!+) \rightarrow$ `"foo"` $+\!\!+$ `"bar"` $\equiv$ `"foobar"` respectively. The expected behaviors are also stated.

4.1.3 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.

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 4 shows the benchmark results under the condition described in the previous section.

4.2.1 Comparison with IGOR II$_H$

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 wrong programs, the proposed system either returned correct programs 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 mentioned in Section 3.3, was developed. This algorithm does not have a direct connection with Spivey's monad and could be applied to IGOR II$_H$.

It should be noted that the proposed system requires more heap space than IGOR II$_H$, because it does not abandon suboptimal programs. The heap consumption may be ignored when using the *Recomp* monad [10] which recomputes instead of keeping temporary search results, but then the computation time increases. One possible solution to this problem might be using the *Recomp* monad and memoizing search results, which is the solution chosen by MAGICHASKELLER.

4.2.2 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 $head :: forall\ a.\ [a] \rightarrow a$ and $last :: forall\ a.\ [a] \rightarrow a$.

	IGOR II$_H$	MAGICHASKELLER	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	∅́	93	>300
fib	>300	16	>300
head	0	∞	0
init	0	3	0
(++)	3	0	0
last	0	∞	0
lasts	0	35	0
lengths	1́	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

Table 4. Benchmark results. Each number 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 ∅́ 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 intention. ∞ 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.

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.

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 5 shows the set of I/O examples used for synthesis of $addN$; 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 6 shows the results of the experiments. The results clearly show the merit of the proposed system over IGOR II$_H$. In all the experiments, the proposed system synthesizes desired programs in more cases than IGOR II$_H$. Especially for $addN$, $andL$, $concat$, and $drop$, the proposed system correctly synthesizes desired programs even from as few as 3 or 6 examples and the test function,

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

Table 5. Set of I/O examples of $addN$ used for evaluating the robustness of the analytical systems.

name	#exs.	IGOR II$_H$	proposed
addN	3	∅́	> 300
	6	∅́	7
	9	∅́	6
	12	∅́	2
	15	∅́	0
	18	35	3
	21	> 300	> 300
allodd	6	∅́	> 300
	10	∅́	0
	15	> 300	26
	21	> 300	> 300
andL	1	∅́	> 300
	3	∅́	0
	7	0	0
	15	0	1
	31	> 300	> 300
concat	3	∅́	0
	6	∅́	0
	9	∅́	0
	13	> 300	> 300
drop	4	∅́	0
	6	∅́	0
	9	> 300	0
	12	> 300	0

Table 6. Results for different number of I/O examples. "#exs." means the number of examples. The meanings of the symbols in the cells are the same as those in Table 4 .

while IGOR II$_H$ does not synthesize expected functions from such a small set of examples. This is because IGOR II$_H$ is satisfied with the most simple program explaining the analyzed I/O example pairs and does not continue the synthesis further. Taking the example of $addN$, as can be seen from Table 5, the first 6 examples of $addN$ simply return the second argument, and therefore IGOR II$_H$ cannot achieve the same result as that of the proposed system. This limitation would not be solved even if a test phase were added to IGOR II$_H$, because IGOR II$_H$ generates only a few programs. On the other hand, the proposed system generates a desired program even in such a hard situation, because it does not abandon programs just because there are other programs with less cost.

5. Conclusions

An analytical IFP algorithm that can generate a stream of programs instead of just generating one or some program(s) was created. This algorithm enabled yet another approach to IFP, the analytically-generate-and-test approach, which generates a stream of programs analytically and filters it with a separately supplied test predicate. By adding the generate-and-test feature to analytical synthesis, the new approach solved the trade-off between the efficiency and the sensitivity to the users' intentions, which IGOR II, the exemplary analytical IFP system, 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.

As for the efficiency, the proposed system is quicker than IGOR II$_H$ in most cases. Some readers may suspect analytical (and analytically-generate-and-test) approaches do not show definite advantages while they have restrictions on the specification to be given. However, the proposed system still has room for improvements in efficiency, and is more promising for synthesis of greater programs than MAGICHASKELLER. For example, memoization of the function that generates a stream of generalized recursive functions which is used for synthesis of each subexpression is not yet implemented in the current proposed system, while its counterpart exists in MAGICHASKELLER. Even without such full memoization, just memoizing the result of applying each defined function to each input pattern should be greatly advantageous in efficiency, because execution of defined function introduction is quite time-consuming. In fact, the cata/paramorphism introduction operator, which can be regarded as a very limited form of such memoization that keeps some simple function application at hand, reportedly improves the synthesis efficiency[5].

Since the cata/paramorphism introduction operator changes the preference bias mainly for efficiency, it should be introduced with care in order not to tune the preference bias with respect to the tendency of the benchmark problem set. We are working on a more general way for making recursive calls more efficient.

Acknowledgments

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] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development*. Springer Verlag, 2004.

[2] A. Cypher, editor. *Watch What I Do: Programming by Demonstration*. MIT Press, Cambridge, 1993.

[3] W. Harris and S. Gulwani. Spreadsheet table transformations from examples. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming Language Design and Implementation*, 2011.

[4] R. Hinze. Generalizing generalized tries. *Journal of Functional Programming*, 10(4):327–351, 2000.

[5] M. Hofmann. *Schema-Guided Inductive Functional Programming through Automatic Detection of Type Morphisms*. PhD thesis, University of Bamberg, 2010.

[6] M. Hofmann and E. 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.

[7] M. Hofmann, E. Kitzelmann, and U. Schmid. A unifying framework for analysis and evaluation of inductive programming systems. In *Proceedings of the Second Conference on Artificial General Intelligence*, 2009.

[8] M. Hofmann, E. Kitzelmann, and U. Schmid. Porting IgorII from Maude to Haskell. In U. Schmid, E. Kitzelmann, and R. Plasmeijer, editors, *Approaches and Applications of Inductive Programming, Third International Workshop, AAIP 2009*, volume 5812 of *LNCS*, pages 140–158, 2010.

[9] S. Katayama. Power of brute-force search in strongly-typed inductive functional programming automation. In *PRICAI 2004: Trends in Artificial Intelligence*, volume 3157 of *LNAI*, pages 75–84. Springer-Verlag, August 2004.

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

[11] S. Katayama. Systematic search for lambda expressions. In *Trends in Functional Programming*, volume 6, pages 111–126. Intellect, 2007.

[12] S. Katayama. Recent improvements of MagicHaskeller. In *Approaches and Applications of Inductive Programming, Third International Workshop, AAIP 2009*, volume 5812 of *LNCS*, pages 174–193, 2010.

[13] S. Katayama. An analytical inductive functional programming system that avoids unintended programs. In E. Kitzelmann and U. Schmid, editors, *Approaches and Applications of Inductive Programming, Fourth International Workshop, AAIP 2011*, pages 33–48, 2011.

[14] S. Katayama. Generating many candidates in analytical inductive functional programming. In *Proceedings of the 38th SICE Symposium on Intelligent Systems*, 2011. in Japanese.

[15] E. 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.

[16] T. Lau. *Programming by Demonstration: a Machine Learning Approach*. PhD thesis, University of Washington, 2001.

[17] S. Muggleton. Inductive logic programming. *New Generation Comput.*, 8(4):295–318, 1991.

[18] R. Olsson. Inductive functional programming using incremental program transformation. *Artificial Intelligence*, 74(1):55–81, 1995.

[19] Y. Onoue, Z. Hu, H. Iwasaki, and M. Takeichi. A calculational fusion system HYLO. In *Algorithmic Languages and Calculi*, pages 76–106, 1997.

[20] A. Solar-Lezama, L. Tancau, R. Bodik, V. Saraswat, and S. Seshia. Combinatorial sketching for finite programs. In *Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[21] J. M. Spivey. Combinators for breadth-first search. *Journal of Functional Programming*, 10(4):397–408, 2000.

[22] J. M. Spivey. Algebras for combinatorial search. *Journal of Functional Programming*, 19:469–487, July 2009. ISSN 0956-7968.

[23] T. Yu. Polymorphism and genetic programming. In *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038 of *LNCS*, pages 218–233, 2001.