Efficient Exhaustive Generation of Functional Programs Using Monte-Carlo Search with Iterative Deepening

Susumu Katayama

University of Miyazaki
1-1 W. Gakuenkibanadai, Miyazaki, Miyazaki 889-2155, Japan
skata@cs.miyazaki-u.ac.jp

Abstract. Genetic programming and inductive synthesis of functional programs are two major approaches to inductive functional programming. Recently, in addition to them, some researchers pursue efficient exhaustive program generation algorithms, partly for the purpose of providing a comparator and knowing how essential the ideas such as heuristics adopted by those major approaches are, partly expecting that approaches that exhaustively generate programs with the given type and pick up those which satisfy the given specification may do the task well. In exhaustive program generation, since the number of programs exponentially increases as the program size increases, the key to success is how to restrain the exponential bloat by suppressing semantically equivalent but syntactically different programs. In this paper we propose an algorithm applying random testing of program equivalences (or Monte-Carlo search for functional differences) to the search results of iterative deepening, by which we can totally remove redundancies caused by semantically equivalent programs. Our experimental results show that applying our algorithm to subexpressions during program generation remarkably reduces the computational costs when applied to rich primitive sets.

1 Introduction

Inductive functional programming is a machine learning field of generation of functional programs by generalization from ambiguous specifications such as input-output examples or constraints over programs. Due to the ambiguity in the way to generalize the specification, in inductive program synthesis it is often the case that the generated programs do not meet the user's intention.

Human programmers usually take the following steps:

$$\text{invent the algorithm} \rightarrow \text{check it by browsing} \rightarrow \text{test it}$$

among which inductive synthesis replaces the algorithm invention part and helps the testing part.

There exist two approaches to inductive functional programming: the generate-and-test approach such as genetic programming (GP) (e.g. [1],[2]) that first generates programs and then tests if they satisfy the specification, and the analytical

T.-B. Ho and Z.-H. Zhou (Eds.): PRICAI 2008, LNAI 5351, pp. 199–210, 2008.

approach that is to some extent based on analysis of the I/O examples, such as the two step methods that first generate a non-recursive program implementing the computational traces from I/O examples and then fold it into a recursive program. (e.g. [3]) Recently, in the analytical approach an algorithm that extends the classical Summers' method is proposed, that searches the hypothesis space narrowed by the template that is obtained by generating the least general generalizations of the input set and the output set [4], replacing the two step methods for its efficiency.

Inductive functional programming by GP can be applied to various problem frameworks. On the other hand, because GP algorithms usually search rather a big hypothesis space, without human labor they tend to consume more computation time. The recent Summers-like method synthesizes programs quickly, though they are limited to synthesis from I/O examples that satisfies some conditions.

We have been working on efficient implementation of exhaustive program generation for given types.[5][6][7] Our main interest is to tell the baseline performance of non-heuristic search, and hopefully provide a new, usable method within the generate-and-test framework. Although our algorithm described in those papers successfully generates small programs without any prior knowledge except the type information, it have been having the following problems:

- it lacks in formalization, although it should efficiently generate infinite number of proofs based on Herbelin's LJT with regard to Curry-Howard isomorphism, i.e., correspondence between proofs and programs;
- it generates lots of mathematically equivalent functions implemented in different ways, most of which are actually identity or constant functions, and which cause inefficiency and human unreadability of the results.

This research continues our policy, and proposes an algorithm that completely removes the redundancy caused by semantically equivalent programs. Instead of removing functions that are theoretically known to be equivalent as suggested in [7], from the generated lazy infinite stream our proposed algorithm completely removes the redundancy caused by semantically equivalent programs, by combining Monte-Carlo search with iterative deepening. By extending the literature on random testing[8], our algorithm can be applied polymorphically — it can be applied to any function, provided that its parameter values can be generated randomly[1] and that an equivalence relation can explicitly be defined between its return values.

Experimental results show that our algorithm effectively restrain exponential bloat when applied to a rich primitive set. This means that our algorithm is useful in practical cases where we want to generate expressions consisted of standard library functions rather than reinventing well-known toy functions from scratch.

[1] Note that [8] shows even higher-order functions can be generated randomly.

2 Exhaustive Program Generation

In this section we review our systematic search algorithm with regard to automatic theorem proving.

Curry-Howard isomorphism is the observation that logic formulas and their proofs have the same structure as types and functions. For example, just in the same way as deriving a proof for B from proofs for $A \to B$ and A, we can obtain the value fx of type B from a function f of type $A \to B$ and a value x of type A, where $A \to B$ denotes the function type taking A as the argument and returning B. Also, just in the same way as deriving a proof for $A \to B$ from a proof of B under the assumption of A, we can obtain a function $\lambda x.E$ of type $A \to B$ by constructing a value E of type B using a variable x of type A as its argument. So far we explained the isomorphism between the propositional logic and simply typed lambda calculus, but there are also isomorphisms between richer logic and lambda calculus.

Under Curry-Howard isomorphism, an automated theorem prover corresponds to an algorithm generating a functional program of the given type. Taking advantage of this fact, some theorem provers such as Coq and Agda have the aspect of deductive programming systems that generate a function satisfying the specification given as a type, though it is hard to totally automate deductive programming under such an expressive type system and they depend on human guidance to some extent.

On the other hand, our systematic exhaustive search algorithm corresponds to generating infinite number of proofs as an infinite stream, under second-order intuitionistic propositional logic, and picking up those which satisfy the given specification. Extending automatic provers to generate infinite number of proofs instead of only one does not dramatically change the algorithm a lot, except that

- in order to consider combinations of infinite possibilities, we have to interleave them somehow or other, using e.g. Spivey's monad for breadth-first search[9] or monad for depth-bound search[10];
- in order to generate all the proofs, even if $A \leftrightarrow B$ we may not replace A with B, because there can be many proofs for $B \to A$, which means G4ip[11], a.k.a. Dyckhoff's LJT [12], cannot be applied, at least straightforwardly;
- the algorithm must be as efficient as possible, using e.g. memoization, though we do not exhaust here all of those that were used.

Our algorithm uses the inference rules of the cut-free LJT with the implication and universal quantification in the Curry style, which is behind the systematic exhaustive search algorithm. More exactly, we prohibit higher-rank polymorphism and use unification for efficiency reasons.

An inductive data type can be made available by providing its constructors and its induction function as assumptions at the left hand side of the turnstile. For example, generating programs of type $\forall a.[[a]] \to a$ corresponds to proving the following sequent:

$$[] :: \forall a.[a],\ (:) :: \forall a.a \to [a] \to [a],\ foldr :: \forall ab.b \to (a \to b \to b) \to [a] \to b\ ;\ \vdash\ \forall a.[[a]]$$

where $[X]$ denotes the type for lists of X's.

One drawback of this approach is that the induction function $foldr$ introduces an existential type when instantiating the type variable a using the ∀L rule, which makes the algorithm inefficient. For this reason or other, [7] prohibited functional types appearing within container types, as in $[a \to b]$ or $(a \to b, c)$, but still suffers from vast search space.

In this paper, we regard lists in the same light as their isomorphic types $\forall b.(a \to b \to b) \to b \to b$, and use the following rule:

$$\frac{\Gamma, xs :: [A]; \vdash op :: A \to B \to B \qquad \Gamma, xs :: [A]; \vdash x :: B}{\Gamma, xs :: [A]; \vdash foldr\ op\ x\ xs :: B}\ \text{foldr}$$

The point is that the type A can be obtained by pattern matching, and thus does not introduce a new existential type.

The same idea applies to other inductive types. We do not deal with coinductive types in this paper.

3 Thinning Up a Stream of Program Sets

3.1 Monte-Carlo Filtration of Program Sets

We use Monte-Carlo search to see if two functions are different, by searching for a point where their values are different. More specifically, we define an equivalence relation based on a random point set as in Definition 1.

Definition 1 (Equivalence by a random point set). *For a random point set* $r = \{r_1..r_n\}$*, we define*

$$f \sim_r g \Leftrightarrow f \sim_{\{r_1..r_n\}} g \overset{def}{\Leftrightarrow} \forall i \in \{1..n\}.f(r_i) = g(r_i)\ .$$

For any point set r, $\sim_r$ becomes an equivalence relation. Moreover, $\sim_{\{r_1..r_n\}}$ is a *refinement* of $\sim_{\{r_1..r_{n-1}\}}$, i.e. $f \sim_{\{r_1..r_n\}} g \Rightarrow f \sim_{\{r_1..r_{n-1}\}} g$. If the random sequence r is exhaustive, e.g., if r is uniform and its domain set is countable, $\sim_{\{r_1..r_\infty\}}$ should equal to the intentional equality. (Note that any set of valid programs or Turing machines is always countable.)

Our algorithm uses a lazy infinite stream to correctly yield a complete set of representatives in the limit. The rough idea is:

- by random testing we can often prove the differences between functions, but can never prove the equivalences;
- therefore, we just abandon functions that may be equivalent to other functions, except one representative, i.e., we obtain (for each depth bound) a complete set of equivalence class representatives by the equivalence based on some random number/function set;
- because we use iterative deepening for generating programs, by using different set of random numbers at each depth limit, "innocent" functions that are unfortunately abandoned but are in fact different from others will eventually be recovered at some depth, provided that we use an exhaustive random number generator.

Table 1. Example of how filtration after program generation works

This is an example of obtaining the representatives for each depth bound $[\{f_1\}, \{f_1\}, \{f_1, f_2\}, ...]$ from the set of functions for each depth bound $[\{f_1\}, \{f_1, f_1'\}, \{f_1, f_1', f_1'', f_2\}, ...]$ and random number sequences for each depth bound $[[1,4], [1,4,2], [1,4,2,3], ...]$, where $f_1 = f_1' = f_1''$ and f_1 and f_2 are defined as follows:

x	1	2	3	4	5	6	...
$f_1(x)$	1	2	3	4	5	6	...
$f_2(x)$	1	2	1	4	5	6	...

but we do not know these facts in advance.

depth bound	1	2
(multi)sets of functions	$\{f_1\}$,(*1)	$\{f_1, f_1'\}$,
random numbers	$[1,4]$,	$[1,4,2]$,
map functions	$\{[f_1(1)=1, f_1(4)=4]\}$,	$\{\ [f_1(1)=1,\ f_1(4)=4,\ f_1(2)=2],$ $[f_1'(1)=1,\ f_1'(4)=4,\ f_1'(2)=2]\ \}$,
equivalence classes	$\{\{f_1\}\}$	$\{\{f_1, f_1'\}\}$
representatives	$\{f_1\}$,	$\{f_1\}$,
differentiate if desired (*2)	$\{f_1\}$,	$\{\}$,

cont'ed	3	...
	$\{f_1, f_1', f_1'', f_2\}$,	...
	$[1,4,2,3]$,	...
	$\{\ [f_1(1)=1,\ f_1(4)=4,\ f_1(2)=2,\ f_1(3)=3],$ $[f_1'(1)=1,\ f_1'(4)=4,\ f_1'(2)=2,\ f_1'(3)=3],$ $[f_1''(1)=1,\ f_1''(4)=4,\ f_1''(2)=2,\ f_1''(3)=3],$ $[f_2(1)=1,\ f_2(4)=4,\ f_2(2)=2,\ f_2(3)=1]\ \}$,	...
	$\{\{f_1, f_1', f_1''\}, \{f_2\}\}$	...
	$\{f_1, f_2\}$,	...
	$\{f_2\}$,	...

(*1) Although we use the set brackets, i.e. { and }, we assume these can be multisets, because we do not have a universal method to prove two functions are equivalent.
(*2) $differentiate\,[S_1,\ S_2,\ S_3,\ ...\,] = [S_1,\ S_2 \setminus S_1,\ S_3 \setminus S_2,\ ...]$. Also see Theorem 1.

Actually, just following the above policy breaks the implicit assumption of iterative deepening that the search space of a deeper iteration is a superset of that of a shallower one. However, if we include all the random sample points used in earlier iterations, i.e., if we append new random sample points instead of totally replacing the point set in each iteration, the equivalence relation refines as the point set increases, and thus we can assure that representatives that once appear will always appear at each of the deeper levels, as stated in the following Theorem 1. Thanks to this theorem, we can differentiate the filtration results by using the syntactical difference at the end if necessary. (Table 1)

Theorem 1. *Let $S(s)$ denote the set we obtain by removing the structure of list s. (For example, $S([3,6,2,2,5]) = \{2,3,5,6\}$) There exists an $O(mn \log n)$-time algorithm A that takes a list of length n and a random sequence of length m and returns a list, such that $S(A(xs, \{r_1...r_m\}))$ is a complete set of representatives of the quotient set $S(xs)/\sim_{\{r_1...r_m\}}$, and $S(A(xs, \{r_1...r_{m-1}\})) \subset S(A(xs +\!\!+ ys, \{r_1...r_m\}))$ where $+\!\!+$ denotes list concatenation.*[2]

Proof. (sketch) $A(xs, rs)$ is the algorithm that sorts xs by the preorder $\leq_{rs}$ and the equivalence $\sim_{rs}$, and selects the first element from each of the resulting

[2] We follow the conventions in the functional programming literature and use plural forms for list variables, e.g. xs, ys, etc. rather than x, y, etc. respectively.

equivalence classes, where $\leq_{rs}$ is the lexicographical order of the function values at the random points, defined recursively as

$$f \leq_{\phi} g \ ,$$
$$f \leq_{\{r_1..r_m\}} g \Leftrightarrow f \leq_{\{r_1..r_{m-1}\}} g \wedge (f \sim_{\{r_1..r_{m-1}\}} g \rightarrow f(r_m) \leq g(r_m)) \ .$$

The sorting algorithm used here must be stable, i.e. it must not change the order between equivalent elements. (Many well-known sorting algorithms such as mergesort and quicksort satisfy this requirement.)

It is trivial that the above algorithm A requires $O(mn \log n)$-time and the result of A is a complete set of representatives. We can prove the inclusion relation between the complete set of representatives by using the next Lemma 1 with the total order of "appearing earlier in xs". □

Lemma 1. *Let $\sim$ and $\approx$ denote two equivalence relations on U, where $\approx$ refines $\sim$. Let $\leq$ denotes a total order on U. For all finite $S, T \subset U$ such that $\forall s \in S. \forall t \in T. s \leq t$, define complete sets of representatives of $\sim$ and $\approx$ as*

$$Q = \left\{ \min_{\leq} c \;\middle|\; c \in S/\sim \right\}$$
$$R = \left\{ \min_{\leq} c \;\middle|\; c \in (S \cup T)/\approx \right\}$$

then,

$$Q \subset R$$

where $\min_{\leq} P$ is the minimal element of P by $\leq$, i.e., $\min_{\leq} P = x$ s.t. $x \in P \wedge \forall y \in P.\ x \leq y$.

This lemma means that if we always pick up the elements that have some property most as the representatives of two equivalence relations where one refines the other and from them, the resulting complete set of representatives of the former includes that of the latter.

3.2 Filtration during Program Generation

By applying the above method to an infinite set of functional programs, we can provably obtain an infinite stream of complete set of equivalence class representatives. Our further interest now is to apply the filter to subprograms *during* program generation rather than *after* program generation, in the hope of some leverage in saving heap consumption.

This is achieved by applying this filter to each item on the memoization table that binds types to sets of expressions. However, there is an implementation issue with regard to existential types: in order to provide polymorphism, our old method memoizes the function that binds types which may include existential types to the set of all expressions whose type unifies with the given type, and generates subexpressions recursively; on the other hand, in order to apply our Monte-Carlo filter, the element functions in the infinite set to be filtered must have the same type. For example, the memo table binds query $[a]$ to the set of

expressions that may include expressions with type [*Char*] and those with type [*Int*] — this obviously causes problems when thinning up the set.

In order to cope with this situation, we use two different memoization tables: one is dedicated to enumerate possible substitutions for each of the existential types, and the other holds the (Monte-Carlo filtered) expressions that have the same type as the query type. In order to obtain a stream of expressions of a given type, our algorithm firstly looks up the first table to obtain possible substitutions, replaces the existential types of the query type, and then looks up the second table.[3]

Another problem is that the number of random samples used per one expression increases as we go deeper iteration, fueling the fire rather than restraining the exponential bloat. This is problematic especially when applying our filter to subexpressions generated during the whole program generation in order to leverage the efficiency. For this reason, we define two exhaustive filters:

Filter 1 , which is efficient but permits some duplicates, that uses a different set of few random numbers for each data type at each iteration, and amends the fewness by accumulating the resulting stream (Table 2), and

Filter 2 , which is inefficient but prohibits any duplicate, that add a random number as the search goes deeper (Table 1).[4]

By applying Filter 1 during program generation and then applying Filter 2 to the final result, we obtain exhaustive but not redundant results. Moreover, this process is more efficient than only applying Filter 2 during program generation, because the amount of data is already thinned up by the Filter 1 when Filter 2 is applied.

The idea behind Filter 1 instead of Filter 2 is using the union of the set of representatives under criterion (random point) r_1, that of representatives under criterion r_2, ... instead of using the set of representatives under their direct product $(r_1, r_2, ...)$. After applying whichever filter, programs returning different values at random point r_n will survive for all n.

Remark 1. In order to apply Monte-Carlo search, there must be an algorithm for generating random number sequence for the domain type. Fortunately, this can be achieved easily in most types including functional ones, using the polymorphic random testing library QuickCheck[8].[5]

4 Experimental Results

We applied our algorithm to MagicHaskeller[13], our systematic exhaustive search library for Haskell.

[3] These steps can be optimized by holding pointers to the entries in the second table, along with substitutions at each entry of the first table.

[4] Note that the two filters only differ in the sets of random numbers.

[5] In software engineering, Monte-Carlo search for programming errors is called random testing.

Table 2. Example of how filtration during generation works

The sets of functions to be filtered are $[\{f_1, f_2\}, \{f_1, f_2, g_1\}, \{f_1, f_2, g_1, h_1, h_2\}, ...]$, where f_1 and f_2 are obtained from the first search, g_1 is obtained from the first deepening, and h_1 and h_2 are obtained from the second deepening.
Assume that $f_1 = g_1$ and $f_2 = h_1$, and that the values of these functions are defined as follows:

x	1	2	3	4	5	6	...
$f_1(x)$	1	2	3	4	5	6	...
$f_2(x)$	1	2	1	4	5	6	...
$g_1(x)$	1	2	3	4	5	6	...
$h_1(x)$	1	2	1	4	5	6	...
$h_2(x)$	1	1	1	1	1	1	...

but we do not know these values in advance.

depth bound	1	2	3	...
sets of functions	$\{f_1, f_2\}$,	$\{f_1, f_2, g_1\}$,	$\{f_1, f_2, g_1, h_1, h_2\}$,	...
random numbers	$[1, 2]$,	$[3]$,	$[2, 6]$,	...
map functions	$\{\ [f_1(1) = 1,\ f_1(2) = 2],$ $[f_2(1) = 1,\ f_2(2) = 2]\ \}$,	$\{\ [f_1(3) = 3],$ $[f_2(3) = 1],$ $[g_1(3) = 3]\ \}$,	$\{\ [f_1(2) = 2,\ f_1(6) = 6],$ $[f_2(2) = 2,\ f_2(6) = 6],$ $[g_1(2) = 2,\ g_1(6) = 6],$ $[h_1(2) = 2,\ h_1(6) = 6],$ $[h_2(2) = 1,\ h_2(6) = 1]\ \}$,	...
equivalence classes	$\{\{f_1, f_2\}\}$,	$\{\{f_1, g_1\}, \{f_2\}\}$	$\{\{f_1, g_1, f_2, h_1\}, \{h_2\}\}$,	...
representatives	$\{f_1\}$,	$\{f_1, f_2\}$,	$\{f_1, h_2\}$,	...
accumulate(*1)	$\{f_1\}$,	$\{f_1, f_2\}$,	$\{f_1, f_2, h_2\}$,	...

(*1) *accumulate* $[S_1,\ S_2,\ S_3, ...] = [S_1,\ S_1 \cup S_2,\ S_1 \cup S_2 \cup S_3,\ ...]$. The whole algorithm should work even when the union is that of multiset (because finally duplicates will be removed by Filter 2), but we can remove some duplicates at this step by syntactical equivalence.

4.1 Experiment Conditions

The current MagicHaskeller is released with several program generator algorithms and some options. The algorithms differ in the hypothesis space in the way described in Sect. 2, and options permit us to selectively enable each rule for theoretical equivalence check between expressions, which are based on some known optimization rules[7].

In this paper we stick to the newly introduced generator described in Sect. 2 and do not compare it with the old program generator with vast hypothesis space, due to the page limitation. Also, we disable the theoretical equivalence check.

We used Version 6.8.2 of Glasgow Haskell Compiler (GHC) on Linux 2.6.22-14, running Intel Pentium D 2.8GHz as a single processor. We modified MagicHaskeller to use depth-bound search with iterative deepening instead of breadth-first search, where the expressions are prioritized by the program size, that is measured by the number of function applications.

We experimentally discuss how many random sample points for each depth bound should be used for Filter 1 in Sect. 4.2, because there is a tight trade-off. The numbers used for each depth bound of Filter 2 are $[6, 7, 8, ...]$. This decision is based on our intuition that two functions are often equivalent if their return values correspond at six points, our confidence in the algorithm's property that the functions will be recovered later if they are actually different, and our observation that the program generation is the bottleneck and the computation

cost of Filter 2 does not affect the total cost very much, if Filter 1 is applied during program generation.

Each execution timeouts 20 milliseconds after invocation. Programs that caused either timeout or an error during execution are removed for the iteration. (Stack space overflow is the most common one.) Since we are using a lazy language, such an error or timeout may occur during comparison between the return values of two programs; in such cases, both programs are removed, because it is difficult to tell which program is to be blamed.

We use different primitive sets named `mnat`, `mlist`, `mlistnat`, and `mrich`. `mnat` is a function set related to natural numbers, i.e. zero, successor, curried paramorphism, and addition, where Int is used to represent the type of natural numbers. `mlist` is a function set related to lists, i.e. nil, cons, and curried list paramorphism. `mlistnat` is the union of `mnat` and `mlist`. `mrich` is a rich function set mainly related to lists and booleans, i.e., `mlist` plus *not*, *map*, append, *filter*, *concat*, *concatMap*, *length*, *replicate*, *take*, *drop*, *takeWhile*, *dropWhile*, *lines*, *words*, *unlines*, *unwords*, *reverse*, *and*, *or*, *any*, *all*, and *zipWith* functions plus binary append `(++)`, and `(&&)`, or `(||)`, extensional equality `(==)`, and extensional inequality `(/=)` operators, where the equality and inequality operators are defined for some types.[6]

The experiments are made reproducible by using Version 0.8.4 of MagicHaskeller.

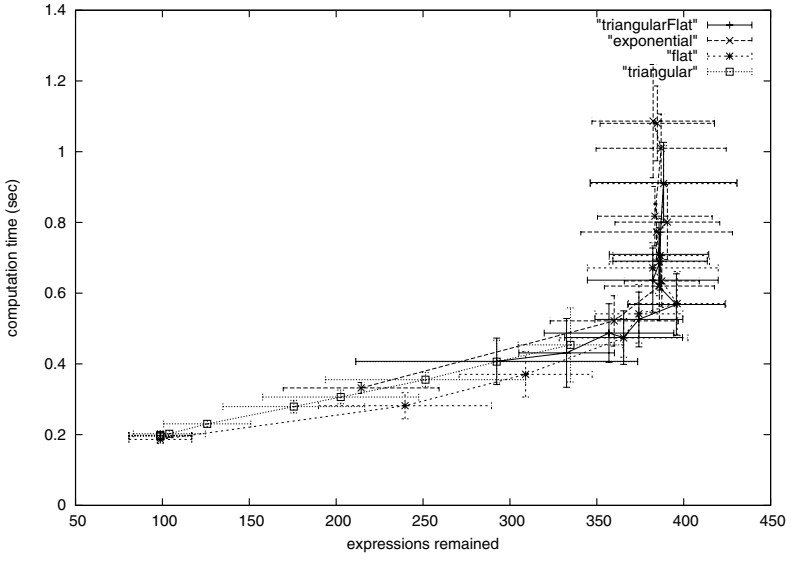
4.2 How Many Random Sample Points in Filter 1?

We use iterative deepening and the result of each iterations is filtered by Monte-Carlo method, in the way already discussed. So far so good, but there remains an issue of how many random sample points to be used at each iteration, and we do not have any conclusive theory on the strategy. In general, using more random points at each iteration means more difference to be proved and less expressions to be lost in the early iterations, but also means more execution time. One may think that more random points should be used for smaller depth bounds because small expressions are repeatedly reused everywhere. Others may think that less should be used for them because they do not directly affect the final result.

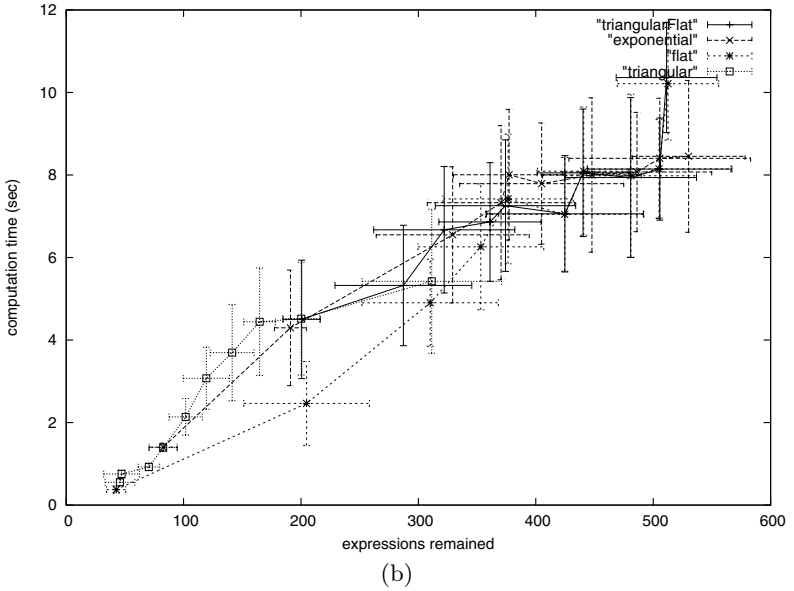
Figure 1 depicts the experimental results of the trade-off lines for some simple strategies. Since less time and more expressions are desirable provided that those expressions are proved to be different, strategies near the down-right corner should be good strategies. Table 3 defines the strategy families we tried. Strategies which appear in Table 3 but not in the legend of Fig. 1 are those which are known to perform very poorly and omitted to avoid clutter. For each strategy family, points and error bars for $n = 1...10$ are plotted, which represent the averages and standard deviations of 5 runs.

Graphs in Fig. 1 suggest the simple flat strategy is nearly the pareto optimal in both graphs; according to Fig. 1a the optimal parameter n is around 4 to 6, while according to Fig. 1b the optimal n is around 8 to 9, though in the latter case the graph has not converged yet.

[6] Currently MagicHaskeller does not support type classes.



(a)



(b)

Fig. 1. Trade-off lines between the computation time and the number of remaining generated expressions within finite depth.
(a) number of functions with type $String \to String$(*1) until the depth bound $t = 7$, generated from the `mrich` primitive set, (b) number of functions with type $Int \to Int$ until the depth bound $t = 10$, generated from the `mnat` primitive set.
(*1) In Haskell, *String* is an alias to [*Char*].

Table 3. Strategy names

strategy family name	# of random points (d: depth bound)	example of $n = 2$, $t = 10$
delta	n if $d = 0$; 1 otherwise	[2,1,1,1,1,1,1,1,1,1,1]
exponential	$\lceil 2^{4-d/n} \rceil$	[16,12,9,6,5,3,3,2,2,1,1]
flat	n	[2,2,2,2,2,2,2,2,2,2,2]
trapezoidal	$\lfloor 3.5 + n + (8 - 2n)d/t \rfloor$	[5,5,6,6,7,7,7,8,8,9,9]
triangular	$\max\{1, t - d - (n - 2)\}$	[10,9,8,7,6,5,4,3,2,1,1]
triangularFlat	$\max\{n, t - d\}$	[10,9,8,7,6,5,4,3,2,2,2]
steepTriangular	$\max\{1, n(t - d)\}$	[20,18,16,14,12,10,8,6,4,2,1]

Table 4. Time spent for generating all the possible expressions within 8 function applications. The "# of exprs" column shows the number of expressions generated at each depth. (NB: this does not mean "within each depth-bound", i.e., this is the differentiated value.) "h/e" means out of memory.

primitive set	query type	not filtered time (sec)	not filtered # of exprs
mnat	$Int \rightarrow Int$	0.70	[2,2,6,22,78,324,1492,7726,42994]
mlistnat	$Int \rightarrow String \rightarrow String$	0.58	[2,0,0,14,22,74,492,3030,14776]
mrich	$String \rightarrow String$	h/e	[2,5,42,225,1755,12226,98008,771208,
mrich	$(Char \rightarrow Int) \rightarrow String \rightarrow [Int]$	10.78	[1,2,12,63,415,2736,20393,155031,1240668]

	with Filter 2 time (sec)	with Filter 2 # of exprs	with Filters 1,2 time (sec)	with Filters 1,2 # of exprs
cont'ed	6.06	[2,2,3,4,9,20,44,98,286]	3.20	[2,2,3,4,12,14,53,119,251]
	2.60	[2,0,0,3,0,3,13,10,107]	2.35	[2,0,0,0,0,1,7,11,60]
	h/e	[2,1,3,13,19,101,304,1087,	55.11	[2,1,3,12,19,112,265,918,3793]
	298.92	[1,0,0,3,1,3,21,22,120]	10.25	[1,0,0,3,0,0,22,13,128]

4.3 Efficiency

Table 4 shows the time spent for computation and the number of generated programs without/with our Monte-Carlo filters. Based on the observation seen in the last section, we used the flat strategy with $n = 5$.

By comparing the number of expressions after applying only Filter 2 and that after applying Filters 1,2, one can tell that few different programs are lost by using Filter 1 except for the case of generating $Int \rightarrow String \rightarrow String$. Also, applying Filter 1 always reduces the computation time, especially when applied to results generated using the `mrich` primitive set.

5 Conclusions

We presented an algorithm for stripping mathematically equivalent functions from a prioritized infinite bag of functions, which is obtained as a search result. We implemented a function that takes such a prioritized bag as an argument and returns its complete set of representatives as a prioritized infinite set of functions. Our algorithm does not require that the equivalence between each function in the prioritized set is explicitly defined, but that its parameter values can be generated randomly, and that equivalence between return values is explicitly defined. Also, we applied the proposed algorithm to removing duplicates in sets of subexpressions during program generation by MagicHaskeller. Our experimental

results show that it is effective for restraining the exponential bloat to some extent when using a relatively large primitive set. This means that our algorithm is useful in practical cases where we want to generate expressions consisted of standard library functions rather than reinventing well-known toy functions from scratch.

References

1. Olsson, R.: Inductive functional programming using incremental program transformation. Artificial Intelligence 74(1), 55–81 (1995)
2. Yu, T.: Polymorphism and genetic programming. In: Miller, J., Tomassini, M., Lanzi, P.L., Ryan, C., Tetamanzi, A.G.B., Langdon, W.B. (eds.) EuroGP 2001. LNCS, vol. 2038, pp. 218–233. Springer, Heidelberg (2001)
3. Schmid, U.: Inductive Synthesis of Functional Programs – Learning Domain-Specific Control Rules and Abstract Schemes. Springer, Heidelberg (2001); Habilitation thesis
4. Kitzelmann, E.: Data-driven induction of recursive functions from input/output-examples. In: AAIP 2007: Proceedings of the Workshop on Approaches and Applications of Inductive Programming, pp. 15–26 (2007)
5. Katayama, S.: Power of brute-force search in strongly-typed inductive functional programming automation. In: Zhang, C., Guesgen, H.W., Yeap, W.-K. (eds.) PRICAI 2004. LNCS (LNAI), vol. 3157, pp. 75–84. Springer, Heidelberg (2004)
6. Katayama, S.: Library for systematic search for expressions and its efficiency evaluation. WSEAS Transactions on Computers 12(5), 3146–3153 (2006)
7. Katayama, S.: Systematic search for lambda expressions. In: Trends in Functional Programming, Intellect, vol. 6, pp. 111–126 (2007)
8. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: ICFP 2000: Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming, pp. 268–279. ACM, New York (2000)
9. Spivey, M.: Combinators for breadth-first search. Journal of Functional Programming 10(4), 397–408 (2000)
10. Spivey, M.: Algebras for combinatorial search. In: Workshop on Mathematically Structured Functional Programming (2006)
11. Hudelmaier, J.: Bounds on cut-elimination in intuitionistic propositional logic. Archive for Mathematical Logic 31, 331–354 (1992)
12. Dyckhoff, R.: Contraction-free sequent calculi for intuitionistic logic. Journal of Symbolic Logic, 795–807 (1992)
13. Katayama, S.: MagicHaskeller (2005), `http://nautilus.cs.miyazaki-u.ac.jp/~skata/MagicHaskeller.html`