Systematic search for lambda expressions

Susumu Katayama

University of Miyazaki

Abstract

This paper presents a system for searching for desired small functional programs by just generating a sequence of type-correct programs in a systematic and exhaustive manner and evaluating them. The main goal of this line of research is to ease functional programming, along with the subgoal to provide an axis to evaluate heuristic approaches to program synthesis such as genetic programming by telling the best performance possible by exhaustive search algorithms. While our previous approach to that goal used combinatory expressions in order to simplify the synthesis process, which led to redundant combinator expressions with complex types, this time we use de Bruijn lambda expressions and enjoy improved results.

1 INTRODUCTION

Type systems are by nature tools for sound programming that constrain programs to help identifying errors. On the other hand, by exploiting strong typing, functional programming can be done in a way like solving a jigsaw puzzle, by repetition of combining unifying functions and their arguments until the programmer eventually obtains the intended program. Search-based approach to inductive program synthesis, or program synthesis from incomplete specification, can be viewed as automation of this process.

This research proposes an algorithm that searches for the type-consistent small functional programs from an incomplete specification rather in a systematic and exhaustive way, that is, by generating all the programs from small to infinitely large that match the given type and checking if they satisfy the specification. Note that due to the incompleteness in the specification the synthesized programs may not conform the users' intentions, like other inductive programming methods. The correctness of the synthesized programs could be assured through tests during or after the synthesis. Also, for the above reason it is desirable that synthesized programs are easy to understand and pretty printed.

The proposed algorithm improves the efficiency of the previous algorithm[3] by doing the following:

- it searches for de Bruijn lambda expressions, while the old one obtains combinatory expressions which is more redundant;
- the function taking the set of available atom expressions and a type `t` and returning prioritized set of synthesized expressions [1] whose type and `t` unify is now memoized;

[1]More exactly it returns prioritized set of tuples of the synthesized expressions and the unifier substitution.

- more equivalent expressions which cause redundancy in the search space and multiple counting are excluded, i.e., while the old algorithm excludes from the search space reducible expressions only in the sense of combinator reduction, the new one knows some fusion rules.

Perspective applications One obvious effect of this research is to programming. Readers may have more concrete image from Subsection 2.1, though further rearrangement of the obtained expressions may be desired.
Another application can be auto-completion in functional shells (e.g. Esther[9]).

Related work Genetic Programming (GP) also searches for functional programs under likewise conditions, which is a kind of heuristic approach to search for programs by maintaining a set of "promising programs" and by exchanging and varying their subexpressions, based on the assumption that *promising programs should include useful subexpressions*. Although GP is heuristic, however, researchers of GP tends not to compare their algorithms with non-heuristic approaches, leaving it unclear whether the heuristic works or not, and how much it improves the efficiency. This paper focuses on efficient implementation of exhaustive enumeration of expressions, and provides a basis on which to build heuristic approaches in future.

2 IMPLEMENTED SYSTEM

2.1 Overview

User interface The implemented system has a user interface that looks like Haskell interpreters. Figure 1 shows a sample interaction with the implemented system, where each > represents a command line prompt.
When the user types in a boolean typed function in Haskell as the *constraint*, the system tries to synthesize an expression which satisfies it, i.e., which makes the function return *True*. Then it prints the expression converted into usual lambda expression acceptable by Haskell interpreters and compilers. The line beginning with `:load` loads the *library file* written in Haskell subset with the Hindley-Milner type system, and during the synthesis, functions and constants in the library file are used as subexpressions of the resulting expression. Here is an example library file.

```
module Library where

zero :: Int
zero = 0
inc  :: Int→Int
inc  = λx → x+1

nat_para :: Int → a → (Int → a → a) → a
nat_para = λi x f → if i then x else f (i-1) (nat_para (i-1) x f)

nil  :: [a]
nil  = []
cons :: a → [a] → [a]
cons = (:)
```

```
list_para :: [b] → a → (b → [b] → a → a) → a
list_para = λl x f → case l of []  → x
                                a:m → f a m (list_para m x f)
```

Although the above specification may look natural, it is not common among GP systems solving the same kind of problems. Each time synthesizing a function, they require a file written, which describes:

- which primitive functions/terminals to use,
- what constraints to satisfy,
- what fitness function to use as heuristic, and
- what values of the parameters to use.

This usually means that synthesizing a function requires more skills and labors than those for implementing the function by hand. Unlike those systems, we make only realistic requirements aiming to obtain a useful system.

2.2 The old algorithm

This section describes the implementation used in the previous work[3], whose details have been unpublished. Section 2.2.1 and 2.2.2 are devoted to definition of the monad for doing breadth-first search and type inference at the same time. Section 2.2.3 overviews the whole algorithm

2.2.1 Monad for breadth-first search

Spivey [8] devised a monad that abstracts and thus eases implementation of breadth-first search algorithms. Its ideas are:

- the monad is defined as a stream of bags, representing a prioritized bag of alternatives, where the n-th element of the stream represents the alternatives that reside at the depth n of the search tree and have the n-th priority,
- the depth n of the direct product of two of such monads can be defined using the i-th bag of the first monad and the j-th of the second monad for all the i and j combinations such that $i+j=n$, and
- the direct sum of such monads can be defined as concatenation of bags at the same depth.

Once such monad is defined, one can easily define breadth-first algorithms using the direct sum and product, or just by replacing the monad for depth-first search in the source code of algorithms with the above one.
Preliminary experiments showed that using the Spivey's monad without change for our algorithm causes huge consumption of the heap space. Changing the definition of the monad to recompute everything as the focus goes deeper in the search tree solved the problem:

```
newtype Recomp a = Rc {unRc :: Int → Bag a}
instance Monad Recomp where
    return x   = Rc f where f 0 = [x]
                            f _ = []
    Rc f >>= g = Rc (λn → [ y | i ← [0..n]
                               , x ← f i
                               , y ← unRc (g x) (n-i) ])
instance MonadPlus Recomp where
    mzero = Rc (const [])
    Rc f `mplus` Rc g = Rc (λi → f i ++ g i)
```

2.2.2 Type inference monad

When implementing type inference, it is usually convenient to define a monad to hide states such as the current substitution and fresh variable ID and to represent error states (e.g. [2]). One of such implementations in Haskell can be as follows:

```
newtype TI a = TI (Subst → Int → Maybe (a, Subst, Int))
```

where `Subst`s represent the current substitution and `Int`s represent the ID of the next fresh variable.
In order to infer consistent types during search, we have to extend the type inference monad to represent possibilities along with their current substitutions, because the substitution differs depending on each possible node of the search tree. Such extension of the type inference monad can be defined as

```
newtype (Monad m) => TI m a = TI (Subst → Int → m (a, Subst, Int))
```

where `m` represents the monad for representing the possibilities. This monad is usually used in the form of `TI Recomp a` in our algorithm, but `TI [] a` is sometimes more efficient under uniform priorities. The conventional type inference monad can be represented as `TI Maybe a`.

2.2.3 The expression construction

The old algorithm enumerates type-correct combinator expressions that match the requested type, and for representing lambda abstractions it heavily depends on primitive *SKIBC* combinators. It works as follows: let us assume expressions that have the same type as or a more general type than, say, $\forall\, a\, b.\ [a] \rightarrow\ b \rightarrow\ Int$ are requested. Firstly, type variables, which are assumed to be universally quantified under the Hindley-Milner type system, are replaced with non-existent type constructor names, say, $G0$ and $G1$ for the above cases. Then a function named `unifyableExprs` is invoked to obtain the prioritized bags of expressions whose types and the requested type $[G0] \rightarrow\ G1 \rightarrow\ Int$ unify.
Given $[G0] \rightarrow\ G1 \rightarrow\ Int$ as an argument, internally `unifyableExprs` makes a prioritized bag of pairs of the primitive expressions that return $[G0] \rightarrow\ G1 \rightarrow\ Int$ (corresponding the head) and their argument types (corresponding the spine) along with their substitution information wrapped in `Recomp`; then for each argument type of each pair it recursively calls `unifyableExprs` with that argument type to obtain resulting argument expressions, `exprss` in the above code; finally it combines the head expression with `exprss` to return.

2.3 Improvements

2.3.1 de Bruijn lambda calculus

The old primitive combinatory approach is inefficient because the polymorphism loosens the restrictions over the search domain, that is,

- expressions become redundant: polymorphic primitive combinators permit combinatory expressions such as *B C C* which can also be implemented by *I*, and such polymorphic combinators can appear everywhere including places that have nothing to do with the program structure;
- undecided type variables make the shallow nodes in the search tree branch many times: because type variables are replaced too late after being passed through computations, branching is not restricted enough while computing the program candidates, even in the cases where eventually at the leaf of the search tree the algorithm finds out that there is no type-consistent expression below that node;
- the complexity of the request type bloats rapidly as the program size increases.

A solution to the above problems could be use of director strings[4]. In this paper we go further to search over the de Bruijn lambda expressions equivalent to the expressions using director strings. One of the simplest forms of this algorithm is the following varied `unifyableExprs` function:

- the varied `unifyableExprs` takes an additional argument `avails::[`*Type*`]` which represents the types of usable de Bruijn variables, where the *n*-th of the `avails` represents the type of the de Bruijn variable x_n;
- if the requested type is a function type,
  - its argument types are piled at the beginning of `avails` in the reverse order,
  - with the resulting `avails` and the return type of the requested type as arguments, a function named `unifyableExprs'` which has the same type as `unifyableExprs` is invoked, and
  - for each of the expressions `unifyableExprs'` returns, lambda abstraction is mapped as many times as the arity of the requested type;

  otherwise, `unifyableExprs'` is invoked with the same arguments;
- `unifyableExprs'` makes a set of the expressions from the primitive set and `avails` whose return types and the requested type `reqtype` unify, and assuming the set of the expressions as the head, it forms the spine invoking `unifyableExprs` recursively using each argument type as the request type; then it returns the combination of the head with the results of the recursive calls.

Thus, as long as the return type of the head is not a type variable, `unifyableExprs'` returns η-reducible expressions, where partial application is not permitted. For example, `λxs → take 4 xs` is elected over `take 4`. This is not a problem, because we are only interested in generating only one expression from one equivalent class in order to avoid redundant search space, and the elected program need not be optimized — optimizing the resulting program could be done by compilers and is not the scope of this line of research.

The actual implementation of `unifyableExprs'` has to be more complicated than one would imagine from the above description, because the return type of the head can be a type variable, in which case the number of its arguments can be any large. Our current approach to this problem is rather naive, trying the infinite number of alternative substitutions $[X/a], [b \to X/a], [b \to c \to X/a], [b \to c \to d \to X/a], \ldots$ where X is the requested return type, a is the return type of the head, and $b, c, \ldots$ are fresh variables.
Note that the above approach is quite inefficient in some ways, preventing the algorithm from being applied to synthesis of larger programs. For example, when the type variable a is replaced with an n-ary function type, by permuting the arguments there are $n!$ equivalent expressions. Currently we are fixing this problem.
Also note that the arguments newly introduced by the substitutions have to be used. Because this can easily be tested by seeing if all the fresh variables are replaced, the idea is already implemented in the proposed algorithm.

2.3.2 Memoization

The proposed algorithm often tries to synthesize the prioritized set of subexpressions on requests of the same type. In such cases memoization often works.
Although lazy memoization that hashes the pointers of objects is quite commonly known in functional programming, this time it is not a good option, because pointer equality makes little sense here, and there is no need of lazy memoization. We implemented a trie-based memoization by

- defining the data type of lazy infinite trie indexed by the arguments of the memo function,
- putting the return value of the memo function at each leaf node of the trie, and
- looking up the trie instead of computing the function value whenever the return value for any argument is required.

How to implement the generalized trie indexed by any data type is detailed in [1]. [2]
When memoizing `unifyableExprs'`, the trie is indexed by `avails` and `reqtype`, and has a Spivey's monad instead of *`Recomp`* at each leaf node. The type variables in `avails` and `reqtype` should be normalized before the look up, or e.g. $\forall\, a.\ [a]$ and $\forall\, b.\ [b]$ would be regarded as different types.
One problem is that although the same types are often requested, the set of available functions change from time to time. Especially because the set of available functions increases monotonically as the position of the spine in question goes deeper, it is quite likely that the memoization whose argument is the simple combination of the requested return type and the set of the available expressions rarely hits.
A hint on this problem is: "when constructing a type-consistent expression we are only interested in the types of the available functions, not those functions themselves", or in other words, "available expressions of the same type are alternative in the sense of type-consistency." Thus, instead of just looking up the memo trie one

[2] Note that a lazy infinite trie instead of a resizable finite trie has to be used here, and thus unlike the implementation in [1] Patricia tree implementation is not fitted for the integer-indexed tries. However, because the only integers used as indexes are the identification numbers for type variables and type constructors, which are usually small below ten, usual lazy lists can be used instead of Patricia tree without outstanding loss of efficiency.

should do the following to make memo trie hit more often and to save the heap space for memoization:

- reorganize the set of available expressions by sorting them and dividing them into the equivalence class having the same type, and assign new variable number for each class,
- look up the return value from the trie using the reorganized list, and
- replace all the variable numbers introduced above in the obtained expressions with the available expressions.

Because memoization costs the heap space, the values that are rarely looked up should rather be recomputed than memoized. In the case of memoizing `unifyableExprs'`, it is effective to look up from the memo only small expressions because they are requested many times.

2.3.3 Excluding unoptimized expressions

Every known optimization rule suggests when equivalent programs can be synthesized, and helps to avoid such redundant synthesis. Taking a case of function fusion as an example, from the $foldr/nil$ rule we know that $foldr\ op\ x\ [] = x$, and thus we should always avoid synthesis of expressions including $foldr\ ?\ ?\ []$ pattern as a subexpression. Failure to exploit such rules can cause a tremendous loss to our algorithm because without doing so $foldr\ ?\ ?\ []$ pattern can appear everywhere even when the requested type has nothing to do with lists at all. Though such fusion points can be identified by a general rule[7], currently we use the following heuristic to capture them:

- identify the library functions that consume some data type beforehand; whether a function is a consumer or not is currently just guessed from its type;
- prohibit use of constructors when invoking `unifyableExprs` for the parameter of consumed data type.

2.3.4 Interpreter

The improved system uses an interpreter based on categorical combinatory logic (CCL) [5], while that used by the old system used the Turner's algorithm. This is a natural change because CCL has a strong tie with the de Bruijn style lambda expressions.

3 EFFICIENCY EVALUATION

Honestly speaking, the current system requires further improvements in efficiency to become very useful. Although synthesis of simple functions consisted of several primitive expressions requires only a few seconds, synthesis of functions consisted of more than twelve primitive expressions usually requires more than a minute, and may not finish the synthesis within an hour. Still, there is large room for improvements by suppressing redundant expressions before trying heuristic approaches.
In addition to the old system, this research is related to GP algorithms such as PolyGP[11],[10] that is GP under polymorphic higher-order type system, and ADATE[6] that uses monomorphic first-order type system. Although it is usually desirable to

compare the proposed method with such algorithms, here we do not compare efficiency with them for the reasons below.
Comparison with PolyGP would be unnecessary, because the comparison between PolyGP and the old system is done in [3], where the latter showed better results on all problems there, and thus if the current system shows improvements from the old system on those problems, that would be enough.
The ADATE system has two releases: version 0.3 and 0.41, but unfortunately in our modern environment both of them could not reproduce the interesting results from the literature. The version 0.3 is released only in the binary form, compiled in an outdated, specific environment, and thus is not runnable in our recent Linux systems. Although the version 0.41 seems runnable, it does not show the reported efficiency even from the sample specifications that come with the release — for example, the list sorting problem, which can reportedly be solved in 1529 seconds on 200MHz Pentium III, cannot be solved in five days on our 3GHz Pentium 4 machine. [3]
One possible reason of the above discouraging result might be as follows: although in usual GP some descendants of parent individuals who are stuck at local minima are dropped to minima at some interpolating positions by the crossover operator, since ADATE lacks in any crossover it is likely that without good luck in the random number seed it is difficult for any individual program to get out of local minima.
It is unknown how often ADATE fails, when it should be restarted if it seems to have failed, and how long it takes in average to obtain a desired result finally after the failures.

Improvements from the old algorithm Table 1 shows the execution time of synthesizing the same functions in the same environment as in [3], i.e. on Pentium4 2.00GHz machine with the Glasgow Haskell Compiler ver. 6.2 on Linux 2.4.22, with the `-O` optimization flag.
For the new algorithm an adapted version of the original library file excluding *SKIBC* combinators and specialized variants of paramorphisms that are no longer necessary is used, i.e.:

```
module Library where

zero :: Int
zero = 0
inc  :: Int→Int
inc  = λx → x+1
nat_para :: Int → a → (Int → a → a) → a
nat_para =  λi x f → if i then x else f (i-1) (nat_para (i-1) x f)
dec :: Int → Int
dec = λx → if x ≡ 0 then 0 else x-1

nil  :: [a]
nil  = []
cons :: a → [a] → [a]
cons = (:)
list_para :: [b] → a → (b → [b] → a → a) → a
```

[3]In addition, when given other problems than the samples, it aborts because of lack in error handling.

TABLE 1. Computation time (sec.) of the old and the new algorithms.

	nth	map	length
computation time for the old system (real)	5.3	2.2	0.03
(user)	5.1	2.2	0.02
	nth	map	length
computation time for the new system (real)	0.8	1.9	0.03
(user)	0.6	1.2	0.02

```
list_para =  λl x f → case l of []  → x
                                a:m → f a m (list_para m x f)
hd :: [a] → a
hd = head
tl :: [a] → [a]
tl = tail
```

In all the experiments performance improvements are observed.

4 CONCLUSIONS

An algorithm that searches for the type-consistent functional programs from an incomplete set of constraints in a systematic and exhaustive way is proposed. It improves the efficiency of the previous algorithm by using de Bruijn lambda calculus at the back end, memoization, and some known fusion rules to avoid multiple counting of the equivalent expressions.

REFERENCES

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

[2] M. P. Jones. Typing Haskell in Haskell. In *Haskell Workshop*, September 1999.

[3] S. Katayama. Power of brute-force search in strongly-typed inductive functional programming automation. In *PRICAI 2004: Trends in Artificial Intelligence, 8th Pacific Rim International Conference on Artificial Intelligence, LNAI 3157*, pages 75–84, August 2004.

[4] R. Kennaway and R. Sleep. Director strings as combinators. *ACM Transactions on Programming Languages and Systems*, 10(4):602–626, 1988.

[5] R. D. Lins. A new formula for the execution of categorical combinators. In *Proceedings of 8th International Conference on Automated Deduction, LNCS 230*, pages 89–98, 1986.

[6] R. Olsson. Inductive functional programming using incremental program transformation., 1994.

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

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

[9] A. van Weelden and R. Plasmeijer. A functional shell that dynamically combines compiled code. In *Proceedings on 15th international workshop on the implementation of functional languages*, Scotland, September 2003. Springer Verlag.

[10] T. Yu. Polymorphism and genetic programming. In *Proceedings of Fourth European Conference on Genetic Programming*, 2001.

[11] T. Yu and C. Clack. PolyGP: A polymorphic genetic programming system in Haskell. In *Genetic Programming 1998: Proc. of the Third Annual Conference*, pages 416–421, 1998.

```
> :set --quiet
> \f -> f ["sldkfj", "", "324oj", "wekljr3","43sld"] == "s3w4"
The inferred type is: (([] ([] Char)) -> ([] Char)).
Looking for the correct expression.
current program size = 1
current program size = 2
current program size = 3
current program size = 4
current program size = 5
current program size = 6
current program size = 7
current program size = 8
current program size = 9
Found!
(\ a -> list_para a nil (\ b c d -> list_para b d (\ e f g -> cons e d)))
1 sec in real,
0.83 seconds in CPU time spent.
> :load LibList
> \f -> f "sldkjf" == "fjkdls"
The inferred type is: (([] Char) -> ([] Char)).
Looking for the correct expression.
current program size = 1
current program size = 2
current program size = 3
current program size = 4
current program size = 5
current program size = 6
current program size = 7
current program size = 8
Found!
(\ a -> list_para a (\ b -> b) (\ b c d e -> d (cons b e)) nil)
1 sec in real,
0.35 seconds in CPU time spent.
>
```

FIGURE 1. Sample user interaction.