Library for Systematic Search for Expressions

SUSUMU KATAYAMA
Department of Computer Science and Systems Engineering
University of Miyazaki
1-1 W. Gakuenkibanadai
Miyazaki, Miyazaki 889-2155
JAPAN
skata@cs.miyazaki-u.ac.jp http://nautilus.cs.miyazaki-u.ac.jp/~skata

Abstract: - In our previous work we showed that systematic search approach to inductive functional programming automation makes a remarkably efficient algorithm, but the applicability of the implemented program was limited by the very poor interpreter incorporated. In this paper, we present a library-based implementation of our systematic search algorithm which is supposed to be used with a well-known Haskell interpreter.

Key-Words: - inductive program synthesis, functional programming, library design, metaprogramming, Template Haskell, artificial intelligence

1 Introduction

MagicHaskeller is our inductive programming automation system by (exhaustively-and-efficiently-)generate-and-filter scheme for the lazy functional language Haskell. Until our previous work[1][2] it has been implemented as a stand alone program including a cheap interpreter for filtration of the generated programs. In this research we implemented its API library package which is supposed to be used with Glasgow Haskell Compiler interactive (GHCi) version 6.4 and higher.

The library implementation enjoys the following effects (some of which will be discussed further in Section 4):

- user-defined data types can be used;
- the full language features and libraries that GHC provides are supported for processing generated expressions;
- the usability when one wants to choose from different primitive sets has improved; for example, users can predefine reusable chunks of primitives such as list-related one, nat-related one, etc. and append them to obtain a suitable primitive set for the synthesis to be conducted.
- it has become easier to investigate the properties of the set of all the expressions that can be generated from the given type and the given primitive component set.

2 Background

2.1 MagicHaskeller

MagicHaskeller[1][2] is a project which emerged from our question: "Is genetic programming really more efficient than elaborated systematic exhaustive search for type-correct programs? If so, when and how is it? Has someone ever compared them?" In order to do the right things in the right order, in this project we first elaborate on optimized implementation of systematic exhaustive search for type correct programs, and then, if we become certain that we need heuristic approaches and that they really help improving the efficiency, we will build heuristic approaches on the top of our research on efficient systematic search algorithms under type constraints. Using techniques from software science such as monad for breadth-first search[3], generalized trie[4], monad transformer[5], and memoization using lazy data type[2], we have already shown that systematic exhaustive search for type-correct programs is a powerful alternative to genetic programming. The implemented program and library can be downloaded and tried from `http://nautilus.cs.miyazaki-u.ac.jp/~skata/MagicHaskeller.html`.

2.2 Template Haskell

Template Haskell (TH) [6] brings template metaprogramming to Haskell. In TH, expressions, sets of declarations, and types to be spliced have type *ExpQ*, $Q\ [Dec]$, and *TypeQ* respectively, where $ExpQ = Q\ Exp$ and $TypeQ = Q\ Type$, where monad Q is a wrapper to avoid name collision. A code block can be spliced by using $\$(.)$. An expression, a set of declarations, and a type can be quoted by using $[\![.]\!]$, $[d|.]\!]$, and $[t|.]\!]$ respectively.

For each of the primitive components we need the information of both of its semantics and its syntax, i.e., the value of the component which is used for interpretation and the name of the component which is used to print the generated expression. The most important role of TH in our library is to generate both of them without re-

quiring users to write the same name twice, which will be explained further in the definition of function p in the next section.

3 Library specification

This section is devoted to the documentation of the implemented library. For each of the exported functions, its type and its short description are given.

3.1 Re-exported modules

This library implicitly re-exports the entities from **module** *Language.Haskell.TH as TH* and **module** *Data.Typeable* from the Standard Hierarchical Library of Haskell. Please refer to their documentations on types from them — in this documentation, types from TH are all qualified and the only type class used from **module** *Typeable* is Typeable.Typeable.

The following types are assigned to our internal data representations: **type** *Primitive* representing each primitive combinator and **type** *Memo* representing the memoization table.

3.2 Setting up the synthesis

Before synthesis, you have to define at least a memoization table (or you may define one once and reuse it for later syntheses). Other parameters are memoization depth and time out interval, which have default values. You may elect either to set those values to the 'global variables' using 'set*' functions, or hand them explicitly as parameters.

Note that the functions related to global variables are supposed to be used within the GHCi environment, and are not tested on other environments. They are implemented in the same way as GHCi implements global variables. For example, the global memoization table is implemented as:

```
{-# NOINLINE refmemodeb #-}
refmemodeb :: IORef Memo
refmemodeb = unsafePerformIO (newIORef defaultMD)
defaultMD = mkMemo []
```

Functions for creating memoization tables from the primitive components You can set your primitives like, e.g.,

```
setPrimitives
  $(p [|(+) :: Int -> Int -> Int, 0 :: Int, 'A', [] :: [a]|])
```

where the primitive set is consisted of $(+)$ specialized to type *Int*, 0 specialized to type *Int*, `'A'` which has monomorphic type *Char*, and $[\,]$ with polymorphic type $[a]$. As primitive components one can include any variables and constructors within the scope unless data types holding functions such as $[a \to b]$, $(Int \to Char, Bool)$, etc. are involved. However, because currently ad hoc polymorphism is not supported by this library, you may not write

```
setPrimitives $(p [|(+) :: Num a => a -> a -> a|])
```

Also, you have to specify the type unless you are using a monomorphic component (just like when using the dynamic expression of Concurrent Clean), and thus you may write `setPrimitives $(p [|'A'|])`, while you have to write `setPrimitives $(p [|[] :: [a]|])` instead of `setPrimitives $(p [|[]|])`

```
p :: TH.ExpQ -> TH.ExpQ
type Primitive = (HValue, TH.Exp, String)
```

p is used to convert your primitive component set into its internal form. Usually its argument is the quasi-quote of a tuple of primitive components, but when it is not, singleton list will be generated.

The internal representation of each item of the primitive component set has type $Primitive = (HValue, TH.Exp, String)$, where **newtype** $HValue = HV\ (forall\ a.a)$ is a universally quantified value representing the semantics of the primitive component, *TH.Exp* represents its syntactic aspect, and *String* is a string representation of its type. For example, $p\ [|((+) :: Int \to Int \to Int, 0 :: Int, \texttt{'A'}, [] :: [a])|]$ reduces to[1]

```
[(HV (unsafeCoerce#((+) :: Int -> Int -> Int)),
  TH.VarE (TH.mkName "+"),
  "(((->) Int) (((->) Int) Int))"),
(HV (unsafeCoerce#(0 :: Int)),
  TH.LitE (TH.IntegerL 0),
  "Int"),
(HV (unsafeCoerce#'A'),
  TH.LitE (TH.CharL 'A'),
  show (typeOf 'A')),
(HV (unsafeCoerce#([] :: forall a_0.[a_0])),
  TH.ConE (TH.mkName "[]"),
  "([] a_0)")]
```

Use of *HValue* and *unsafeCoerce#* for such purposes is taken from the source code of GHCi. The type consistency will not be assured statically by GHC but dynamically by our library. When the type is not given, it will be obtained via *typeOf* method as is seen in the `'A'` case of the above example, although that will cause ambiguity error if the value is polymorphic.

```
setPrimitives :: [Primitive] -> IO ()
setPrimitives = setMemo.mkMemo
mkMemo :: [Primitive] -> Memo
setMemo :: Memo -> IO ()
```

Memoization depth

```
setDepth :: Int ->     -- memoization depth
  IO ()
```

[1] Here we stripped some of the qualifications by module names in order to avoid clutter.

setDepth can be used to set the memoization depth. (Sub)expressions within this size are memoized, while greater expressions will be recomputed (to save the heap space).

Functions related to time out Because the library generates all the expressions including those with non-linear recursions, you should note that there exist some expressions which take extraordinarily long time.[2] Imagine a function that takes an integer n and increments 0 for 2^{2^n} times. Another example that has smaller program size is *blah* in

```
blah :: Int → Int → Int
blah 0 i = i
blah k i = foo (blah (k − 1)) i
foo h 0 = 2
foo h s = h (f h (s − 1))
```

Giving small values in your examples is a good policy, but even then, time out is useful. For this reason, time out is by default taken after 1 second since each invocation of evaluation. This default behavior can be overridden by the following functions.

```
setTimeout :: Int →     -- time in seconds
              IO ()
unsetTimeout :: IO ()
```

Defining generator functions automatically In this case "automatically" does not mean "inductively" but "deductively using TH".

```
define ::
  String → Integer → TH.ExpQ → TH.Q [TH.Dec]
```

define eases use of this library by automating some function definitions. For example,

```
$(define "Foo" 15 (p ⟦(1 :: Int, (+) :: Int → Int → Int)⟧))
```

is equivalent to

```
memoFoo = mkMemo (p ⟦(1 :: Int, (+) :: Int → Int → Int)⟧)
everyFoo :: Everything
everyFoo = everything 15 memoFoo
filterFoo :: Filter
filterFoo pred = filterThen pred everyFoo
where
type Everything = Typeable a ⇒ [[(TH.Exp, a)]]
type Filter =
  Typeable a ⇒ (a → Bool) → IO [[(TH.Exp, a)]]
```

3.3 The synthesizers

Now is the time for defining the synthesizer functions. Most of the functions here just filter *everything* with the predicates you provide.

Quick starters

```
findOne :: Typeable a ⇒ (a → Bool) → TH.Exp
printOne :: Typeable a ⇒ (a → Bool) → IO ()
printAny :: Typeable a ⇒ (a → Bool) → IO ()
```

findOne pred finds an expression e that satisfies *pred e* ≡ *True*, and returns it in *TH.Exp*. *printOne* prints the expression found first. *printAny* prints all the expressions satisfying the given predicate.

Incremental filtration Sometimes you may want to filter further after synthesis, because the predicate you previously provided did not specify the function enough. The following functions can be used to filter expressions incrementally.

```
filterFirst :: Typeable a ⇒
  (a → Bool) → IO [[(TH.Exp, a)]]
filterThen :: Typeable a ⇒
  (a → Bool) → [[(TH.Exp, a)]] → IO [[(TH.Exp, a)]]
```

filterFirst is like *printAny*, but by itself it does not print anything. Instead, it creates a stream of lists of expressions represented in tuples of *TH.Exp* and the expressions themselves. The stream contains expressions consisted of n primitive components at the nth element ($n = 1, 2, ...$), and thus can be viewed as Spivey[3]'s *Matrix* data type.

filterThen may be used to further filter the results.

Expression generators These functions generate all the expressions that have the type you provide.

```
everything :: Typeable a ⇒
  Int →     -- memoization depth.
  Memo →    -- memo table
  [[(TH.Exp, a)]]
getEverything :: Typeable a ⇒ IO [[(TH.Exp, a)]]
```

everything generates all the expressions that fit the inferred type, and their representations in the *TH.Exp* form.

getEverything is like *everything*, but uses the global values set with `set*` functions.

```
unifyable, matching ::
  Int → Memo → TH.Type → [[TH.Exp]]
```

These two functions are like *everything*, but take *TH.Type* as an argument, which may be polymorphic. For example, *printQ* (⟦*forall a.a → a → a*⟧ ≫= *return.unifyable* 10 *memo*) will print all the expressions using *memo* whose types unify with *forall a.a → a → a*.

Pretty printer These are utility functions for pretty printing the results:

$$pprs :: [[(TH.Exp, a)]] \rightarrow IO\ ()$$
$$printQ :: Ppr\ a \Rightarrow Q\ a \rightarrow IO\ ()$$

4 Use cases

This section presents many interesting results our library implementation has brought around. Unless otherwise specified, we assume that the Haskell source file in Table 1 is preloaded into GHCi.

User defined data types From its birth our algorithm is designed to be able to deal with user-defined data types, but such ability had been suppressed by our poor interpreter in our previous work. Now that we can use the full power of GHCi, this potential ability is uncovered.

Suppose you have a Haskell source file shown in Table 2. You may synthesize programs by using it. Table 3 shows a sample interaction.

Actually the first two of the generated expressions implement different functions. You may want to use QuickCheck[7] for random testing in order to confirm this fact. You define **instance** *Arbitrary a* $\Rightarrow$ *Arbitrary* (*Tree a*) and reload the source. Then, you filter further, as shown in Table 4. Of course, you may use *QuickCheck* to automatically filter the results further, though in this case you have to write a driver.

Using predicates in general other than examples Unlike programming by example approaches, our library accepts any predicate as well as I/O example pairs. Table 5 shows an example of finding recursive form definitions of the Fibonacci function from its closed-form solution.

Managing your primitive set Selecting your primitive set has also become easier. Access to your source file is required less frequently now, as shown in Table 6.

Browsing everything Just browsing all the expressions having the given type is also interesting (Table 7).

It has become easier to tell some properties that the syntheses have. The example in Table 8 counts the number of expressions with each size. [2]

[2]Note that these are not the 'exact' numbers of all the expressions, because our algorithm suppresses some of the duplicate expressions doing the same thing having different syntaxes, but still there remains such redundancy. Rather, you should interpret these numbers suggesting the ability of our algorithm.

5 Conclusions

We presented a library-based implementation of our systematic program generator algorithm which is supposed to be used with Glasgow Haskell Compiler interactive. By using several use cases we showed that this library expands the applicability of our algorithm.

Still, however, there are things to be done in order to enhance the usability of our algorithm:

- this algorithm can enhance Hoogle[8], that is a Haskell API search engine, by providing the ability to generate expressions;
- desirable set of primitive components (e.g. constructors and paramorphisms) for predefined types and their (better) pretty printers should be predefined, and those for user-defined types should be auto-generated;
- there could be a wrapper that automatically curry and uncurry tuple arguments

Also, support of type classes and further efficiency improvement could improve our algorithm.

References

[1] Susumu 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.

[2] Susumu Katayama. Systematic search for lambda expressions. In *Trends in Functional Programming*, volume 6. Intellect, Bristol, UK, in preparation.

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

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

[5] Sheng Liang, Paul Hudak, and Mark P. Jones. Monad transformers and modular interpreters. In *POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1995.

[6] Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. In *Haskell Workshop 2002*, October 2002.

[7] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *International Conference on Functional Programming*, pages 268–279. ACM, 2000.

[8] Neil Mitchell. Hoogle, http://www-users.cs.york.ac.uk/~ndm/projects/hoogle.php.

Table 1: Default primitive component library

```
{-# OPTIONS -fth #-}
module LibTH (module LibTH, module MagicHaskeller) where
import MagicHaskeller
initialize = do
  setPrimitives (list ++ nat ++ mb ++ bool ++ $(p [|(hd :: [a] -> Maybe a, (+) :: Int -> Int -> Int)|]))
  setDepth 15
  -- Specialized memoization tables. Choose one for quicker results.
mall  :: Memo
mall  = mkMemo (list ++ nat ++ mb ++ bool ++ $(p [|(hd :: [a] -> Maybe a, (+) :: Int -> Int -> Int)|]))
mlist :: Memo
mlist = mkMemo list
mnat :: Memo
mnat = mkMemo (nat ++ $(p [|(+) :: Int -> Int -> Int|]))
mlistnat :: Memo
mlistnat = mkMemo (list ++ nat ++ $(p [|(+) :: Int -> Int -> Int|]))
hd :: [a] -> Maybe a
hd []     = Nothing
hd (x : _) = Just x
mb = $(p [|(Nothing :: Maybe a, Just :: a -> Maybe a, caseMaybe :: Maybe b -> a -> (b -> a) -> a)|])
caseMaybe :: Maybe b -> a -> (b -> a) -> a
caseMaybe Nothing x f = x
caseMaybe (Just y) x f = f y
nat = $(p [|(0 :: Int, succ :: Int -> Int, nat_para :: Int -> a -> (Int -> a -> a) -> a)|])
  -- Nat paramorphism
nat_para :: Int -> a -> (Int -> a -> a) -> a
nat_para 0 x f = x
nat_para i x f = f (i - 1) (nat_para (i - 1) x f)
list = $(p [|([] :: [a], (:) :: a -> [a] -> [a], list_para :: [b] -> a -> (b -> [b] -> a -> a) -> a)|])
  -- List paramorphism
list_para :: [b] -> a -> (b -> [b] -> a -> a) -> a
list_para [] x f = x
list_para (y : ys) x f = f y ys (list_para ys x f)
bool = $(p [|(True, False, iF :: Bool -> a -> a -> a)|])
iF :: Bool -> a -> a -> a
iF True t f = t
iF False t f = f
```

Table 2: Component library file with a **data** declaration

```
import MagicHaskeller
data Tree a = Leaf a | Branch (Tree a) (Tree a) deriving (Eq, Typeable, Show)
tree_para :: Tree a -> (a -> r) -> (Tree a -> Tree a -> r -> r -> r) -> r
tree_para (Leaf x) f g = f x
tree_para (Branch t u) f g = g t u (tree_para t f g) (tree_para u f g)
$(define "Tree" 10 (p [|(Leaf :: a -> Tree a, Branch :: Tree a -> Tree a -> Tree a,
  tree_para :: Tree a -> (a -> r) -> (Tree a -> Tree a -> r -> r -> r) -> r)|]))
```

Table 3: Sample interaction with a user-defined type

```
> ghci -v0 -fth -package MagicHaskeller TreeExample.hs
*Main>  x <- filterTree (\f -> f (Branch (Branch (Leaf 1) (Leaf 2)) (Leaf (3::Int))) == Branch
 (Leaf 3) (Branch (Leaf 2) (Leaf (1::Int))))
*Main> pprs x
\a -> tree_para a (\b -> Leaf b) (\b c d e -> Branch e d)
\a -> tree_para a (\b -> Leaf b) (\b c d e -> Branch c d)
\a -> tree_para a (\b c -> c) (\b c d e f -> Branch c (e (d b))) a
\a -> tree_para a (\b c -> c) (\b c d e f -> Branch c (d (e b))) a
\a -> tree_para a (\b c -> c) (\b c d e f -> Branch (e c) (d b)) a
\a -> tree_para a (\b c -> c) (\b c d e f -> Branch c (e (e (d b)))) a
\a -> tree_para a (\b c -> c) (\b c d e f -> Branch c (e (d (e b)))) a
\a -> tree_para a (\b c -> c) (\b c d e f -> Branch c (e (d (d b)))) a
\a -> tree_para a (\b c -> c) (\b c d e f -> Branch c (d (e (e b)))) a
\a -> tree_para a (\b c -> c) (\b c d e f -> Branch c (d (e (d b)))) a
\a -> tree_para a (\b c -> c) (\b c d e f -> Branch c (d (d (e b)))) a
\a -> tree_para a (\b c -> c) (\b c d e f -> Branch (e c) (e (d b))) a
\a -> tree_para a (\b c -> c) (\b c d e f -> Branch (e c) (d (e b))) a
\a -> tree_para a (\b c -> c) (\b c d e f -> Branch (e c) (d (d b))) a
\a -> tree_para a (\b c -> c) (\b c d e f -> Branch (e (e c)) (d b)) a
Interrupted.
*Main>
```

Table 4: Filtering further based on the results from QuickCheck

```
*Main> :reload
*Main> quickCheck (\a ->  tree_para a (\b -> Leaf (b::Int)) (\b c d e -> Branch e d) ==
 tree_para a (\b -> Leaf (b::Int)) (\b c d e -> Branch c d))
Falsifiable, after 29 tests:
Branch (Leaf (-1)) (Branch (Leaf (-2)) (Leaf 2))
*Main> (\a -> tree_para a (\b -> Leaf (b::Int)) (\b c d e -> Branch e d)) $ Branch (Leaf (-1))
 (Branch (Leaf (-2)) (Leaf 2))
Branch (Branch (Leaf 2) (Leaf (-2))) (Leaf (-1))
*Main> (\a -> tree_para a (\b -> Leaf (b::Int)) (\b c d e -> Branch c d)) $ Branch (Leaf (-1))
 (Branch (Leaf (-2)) (Leaf 2))
Branch (Branch (Leaf (-2)) (Leaf 2)) (Leaf (-1))
*Main> y <- filterThen (\f -> f (Branch (Leaf (-1)) (Branch (Leaf (-2::Int)) (Leaf 2))) ==
 Branch (Branch (Leaf 2) (Leaf (-2))) (Leaf (-1::Int))) x
*Main> pprs y
\a -> tree_para a (\b -> Leaf b) (\b c d e -> Branch e d)
\a -> tree_para a (\b c -> c) (\b c d e f -> Branch (e c) (d b)) a
\a -> tree_para a (\b c -> c) (\b c d e f -> Branch (e c) (d (d b))) a
\a -> tree_para a (\b c -> c) (\b c d e f -> Branch (e (e c)) (d b)) a
\a -> tree_para a (\b c -> c) (\b c d e f -> Branch (e c) (d b)) (Branch a a)
\a -> tree_para a (\b c -> c) (\b c d e f -> Branch (e c) (d (d (d b)))) a
\a -> tree_para a (\b c -> c) (\b c d e f -> Branch (e (e c)) (d (d b))) a
\a -> tree_para a (\b c -> c) (\b c d e f -> Branch (e (e (e c))) (d b)) a
Interrupted.
```

Table 5: Finding recursive definitions from closed-form solution

```
LibTH> initialize
LibTH> let phi = (1 + sqrt 5)/2
LibTH> printAny (\f -> all (\n -> (f :: Int->Int) n == round ((phi^n - (1-phi)^n) / sqrt 5) ) [0..9])
\a -> nat_para a (\b c -> b) (\b c d e -> c e ((+) d e)) 0 (succ 0)
\a -> nat_para a (\b c -> b) (\b c d e -> c e ((+) e d)) 0 (succ 0)
\a -> nat_para a (\b c -> b) (\b c d e -> c ((+) d e) d) 0 (succ 0)
\a -> nat_para a (\b c -> b) (\b c d e -> c ((+) e d) d) 0 (succ 0)
\a -> nat_para a (\b c -> b) (\b c d e -> c (succ e) ((+) d e)) 0 0
\a -> nat_para a (\b c -> b) (\b c d e -> c (succ e) ((+) e d)) 0 0
Interrupted.
```

Table 6: Selecting your primitive set

(example cont'd from Table 5)

```
*LibTH> -- elect to use specialized primitive set to make your synthesis quicker
*LibTH> setPrimitives (nat ++ $(p [| (hd :: [a] -> Maybe a, (+) :: Int -> Int -> Int) |]))
*LibTH> printAny (\f -> all (\n -> (f :: Int->Int) n == round ((phi^n - (1-phi)^n) / sqrt 5) ) [0..9])
\a -> nat_para a (\b c -> b) (\b c d e -> c e ((+) d e)) 0 (succ 0)
\a -> nat_para a (\b c -> b) (\b c d e -> c e ((+) e d)) 0 (succ 0)
\a -> nat_para a (\b c -> b) (\b c d e -> c ((+) d e) d) 0 (succ 0)
\a -> nat_para a (\b c -> b) (\b c d e -> c ((+) e d) d) 0 (succ 0)
\a -> nat_para a (\b c -> b) (\b c d e -> c (succ e) ((+) d e)) 0 0
\a -> nat_para a (\b c -> b) (\b c d e -> c (succ e) ((+) e d)) 0 0
Interrupted.
```

Table 7: Browsing all the expressions

```
*LibTH> initialize
*LibTH> printQ ([t| forall a. a->a->a |] >>= return . unifyable 10 mall)
\a b -> b
\a b -> a
\a b -> 0
\a b -> True
\a b -> False
\a b -> []
\a b -> Nothing
\a b -> succ 0
\a b -> succ b
\a b -> succ a
\a b -> hd []
\a b -> Just 0
\a b -> Just True
\a b -> Just False
\a b -> Just []
\a b -> Just Nothing
\a b -> succ (succ 0)
\a b -> succ (succ b)
\a b -> succ (succ a)
Interrupted.
```

Table 8: Counting the number of expressions with each size

```
*LibTH> initialize
*LibTH> runQ [t| forall a. a->a->a |] >>= print . map length . unifyable 10 mall
[7,9,34,76,223,651,2249,8053,31169,Interrupted.
```