McGill University COMP527 Winter2016 final project

Example-directed Program Synthesis

Review and Test Cases

Qi Nan Jin - 260531346

McGill University

qi.jin@mail.mcgill.ca

Abstract

This project closely reviews the concept of example-directed program synthesis with a main focus on the new synthesis language λ_{syn} developed by Osera and Zdancewic (2015), as well as the associated artifact: MYTH. It provides a theoretical review of the underlying concepts, a reproducibility test to the results and a sensitivity test of the synthesis language with respect to example specifications. It also proposes some improvements and additions to the existing MYTH artifact.

General Terms λ_{syn}, MYTH

Keywords Program synthesis, Example-directed

1. Introduction

For decades, program synthesis has been an active and interesting area of research for its promising potential applications (Basin et al. 2004). Since the early works of Green (1969), as well as Manna and Waldinger (1971), a close link has been established between program synthesis and formal deduction systems with heuristics. The underlying relationship between logical frameworks and programs makes program synthesis a particularly interesting topic in both computer science and logics.

In the recent years, more attentions have been attributed to the development of example-directed program synthesis. The ideal of example specifications can be related back to early works include Freeman and Pfenning (1991), which introduces the concept of refinement types. A recent paper by Osera and Zdancewic (2015) introduces a new method for type-and-example-directed program synthesis that claims to match or outperform existing gold-standard methods.

In this project, a theoretical review of the concept of program synthesis will be provided, leading to a detailed examination of the new method by Osera and Zdancewic (2015), as well as the emerging artifact, MYTH.

2. Theory

2.1 Program Synthesis in General

Works in constructive mathematics from as early as the 1930s (Kolmogorov 1932)(Kreitz 1998) laid foundation in the field of program synthesis, by suggesting the parallel between programs as proofs. With the appearance of automatic theorem provers, program synthesizers emerged shortly thereafter(Kreitz 1998).

In its early stage, program synthesis focuses mainly on proof search, based on early works of Green (1969), as well as Manna and Waldinger (1971). The method builds upon the idea that corresponding proof of the specification statement is constructive, for which the steps can be directly associated to methods to construct the program. This method mainly relies on the construction of an extended λ-calculus, following the Curry-Howard Isomorphism between proofs and terms

(Curry et al. 1972) (Tait 1967).

A second branch of program synthesis quickly emerged in the late 1970s based on program transformation (Burstall and Darlington 1977), which usually proceeds by translating the output-condition step-by-step, into logically equivalent or stronger formula (Kreitz 1998). Different branches of possible optimizations methods for transformational synthesis include Partial evaluation (Bjorner et al. 1988), Finite differencing (Paige and Koenig 1982), etc.

2.2 Example-directed program synthesis

Example-directed program synthesis aims to provide concrete input-output examples as extra information in the specification in a proof-search based program synthesis. Based on recent work from Frankle et al. (2016), examples can be interpreted as refinement types, first studies by Freeman and Pfenning (Freeman and Pfenning 1991)(Pfenning 1993). A parallel can be drown between the three fundamental types in the refinement-type system and the input-output specifications. More specifically, the input/output pairs can be identified as singleton types. The functions mapping the input type to the output type can be identified as function/arrow types. Finally, multiple input-output examples used in conjunction can, obviously, identified as the conjunction type in the refinement-type system.

The work of Osera and Zdancewic (2015), builds on this idea and formalizes a synthesis languague: λ_{syn}, as shown in Figure 1.

This language also includes full sets of typechecking rules, synthesis rules, auxiliary synthesis functions as well as evaluation and compatibility rules to form a complete framework. The details are exhaustively presented in the original paper (Osera and Zdancewic 2015).

The formalized synthesis language λ_{syn} operates in two modes: generating type in elimiation form (E-guessing) and checking type in introduction form (I-refinement). This is achieved with the help of an new data structure named *refinement tree. A* refinement tree includes both goal nodes at which E-guessing happen, and refinement nodes at which the type-checking of I-refinement is performed. An iterative-deepening search is then applied to this structure to find a matching function that satisfies the input-output specifications (Osera and Zdancewic 2015).

$$
\begin{array}{lcl}
\tau & ::= & \tau \mid \tau_1 \rightarrow \tau_2 \\
e & ::= & x \mid C(e_1, .., e_k) \mid e_1\ e_2 \\
 & \mid & \text{fix}\, f(x : \tau_1) : \tau_2 = e \mid pf \\
 & \mid & \text{match}\, e\, \text{with}\, p_1 \rightarrow e_1 \mid .. \mid p_m \rightarrow e_m \\
p & ::= & C(x_1, .., x_k) \\
u, v & ::= & C(v_1, .., v_k) \mid \text{fix}\, f(x : \tau_1) : \tau_2 = e \mid pf \\
ex & ::= & C(ex_1, .., ex_k) \mid pf \\
pf & ::= & v_1 \Rightarrow ex_1 \mid .. \mid v_m \Rightarrow ex_m \\
E & ::= & x \mid E\, I \\
I & ::= & E \mid C(I_1, .., I_k) \mid \text{fix}\, f(x : \tau_1) : \tau_2 = I \\
 & \mid & \text{match}\, E\, \text{with}\, p_1 \rightarrow I_1 \mid .. \mid p_m \rightarrow I_m \\
\Gamma & ::= & \cdot \mid x : \tau, \Gamma \\
\Sigma & ::= & \cdot \mid C : \tau_1 * .. * \tau_n \rightarrow T, \Sigma \\
\sigma & ::= & \cdot \mid [v/x]\sigma \\
X & ::= & \cdot \mid \sigma \mapsto ex +\!\!+ X
\end{array}
$$

Figure 1. λ_{syn} syntax as formalized by Osera and Zdancewic (2015).

3. Implementation

The authors implemented a prototype synthesizer MYTH under the OCAML environment. The artifact uses a relatively naive but balanced search strategy to alternate between E-guessing and increasing the number of nodes in the refinement tree. The startegy is described in Figure 2.

```
  SynthSaturate 0.25
; SynthGrowMatches
; SynthSaturate 0.25
; SynthGrowMatches
; SynthSaturate 0.25
; SynthGrowScrutinees 5
; SynthSaturate 0.25
; SynthGrowMatches
; SynthSaturate 0.25
; SynthGrowScrutinees 5
; SynthSaturate 0.25
```

Figure 2. Synthesis strategy of MYTH by Osera and Zdancewic (2015).

where `SynthSaturate` performs a round of E-guessing with the given time constraint, `SynthGrowMatches` increases the depth of the search tree by one, and finally `SynthGrowScrutinees` increases the search scrutinee by a specified size. Here the scrutinee growth size is fixed at five (which corresponds to the size of a binary function application `f e1 e2`) (Osera and Zdancewic 2015).

Using a benchmark testing suite, Osera and Zdancewic (2015) state that MYTH matches or outperforms previous methods by synthesizing more programs as well as achieving it in shorter time on average.

4. Testing

In the context of this review project, a straightforward reproducibility test is first performed, using the method and benchmark test set from the paper, to confirm the results published by the authors. A smaller test set is then applied with minor modifications in the input-output example specifications, in order to assess the artifact's sensitivity to example specifications. Finally, a few new test cases are constructed to test the limits of MYTH.

5. Results

5.1 Reproducibility test

In order to confirm the results published from the paper, a straightforward reproducibility test is first performed using the benchmark test suite and the standard method discribed in the artifact Osera and Zdancewic (2015). The simulation is run on a Linux laptop machine with an Intel i5-4210H @ 2.90GHz and 8Gb of ram. The result of this reproducibility test is presented in Figure 3. Synthesis times using minimal context in the reproducibility test are generally longer than, but comparable to the ones published in the paper. The longer synthesis time can be explained by the mismatch in the specs of the equipments.

One interesting finding is that for several tests, such as `list_append` and `list_nth`, the numbers of examples used is significantly lower than the ones reported in the paper. This is likely caused by the fact that the example set of these tests are further optimized since the paper was submitted.

Test	**#Ex**	**#N**	**Time-Min (s)**	
			test	paper
Booleans				
bool_band	4	6	0.0032	0.002
bool_bor	4	6	0.0034	0.001
bool_impl	4	6	0.0036	0.002
bool_neg	2	5	0.001	0
bool_xor	4	9	0.0036	0.002
Lists				
list_append	6	12	0.007	0.003
list_compress	13	28	0.116	0.073
list_concat	6	11	0.011	0.006
list_drop	11	13	0.023	0.013
list_even_parity	7	13	0.0086	0.004
list_filter	8	15	0.0282	0.067
list_fold	9	13	0.1986	0.139
list_hd	3	5	0.002	0.001
list_inc	4	8	0.0016	0
list_last	6	11	0.005	0
list_length	3	8	0.002	0.001
list_map	8	12	0.0184	0.008
list_nth	13	16	0.025	0.013
list_pairwise_swap	7	19	0.0146	0.007
list_rev_append	5	13	0.017	0.011
list_rev_fold	5	12	0.013	0.007
list_rev_snoc	5	11	0.01	0.006
list_rev_tailcall	8	12	0.0098	0.004
list_snoc	8	14	0.007	0.003
list_sort_sorted_insert	7	11	0.013	0.008
list_sorted_insert	12	24	0.1908	0.122
list_stutter	3	11	0.0018	0.001
list_sum	3	8	0.004	0.002
list_take	12	15	0.1406	0.112
list_tl	3	5	0.0016	0.001
Natural Numbers				
nat_add	9	11	0.004	0.002
nat_iseven	4	10	0.0016	0.001
nat_max	9	14	0.0196	0.011
nat_pred	3	5	0.001	0.001
Trees				
tree_binsert	20	31	0.4856	0.374
tree_collect_leaves	6	15	0.0266	0.016
tree_count_leaves	7	14	0.0162	0.008
tree_count_nodes	6	14	0.016	0.009
tree_inorder	5	15	0.026	0.012
tree_map	7	15	0.029	0.014
tree_nodes_at_level	11	22	1.0388	1.093
tree_postorder	9	32	1.4258	1.136
tree_preorder	5	15	0.0188	0.009

Figure 3. MYTH benchmark suite reproducibility test results using minimal context, where #Ex is the number of examples used, #N is the size of the result

Test case for `list_nth`:

```
type nat =
  | O | S of nat

type list =
  | Nil | Cons of nat * list

let list_nth : list -> nat -> nat |>
  { [] => ( 0 => 0
          | 1 => 0 )
  | [2] => ( 0 => 2
           | 1 => 0 )
  | [1; 2] => ( 0 => 1
              | 1 => 2 )
  | [1] => ( 0 => 1
           | 1 => 0 )
  | [2; 1] => ( 0 => 2
              | 1 => 1 )
  | [3; 2; 1] => ( 0 => 3
                 | 1 => 2
                 | 2 => 1 )
  } = ?
```

Output with original input:

```
let list_nth : list -> nat -> nat =
  let rec f1 (l1:list) : nat -> nat =
    fun (n1:nat) ->
      match n1 with
        | O -> (match l1 with
                  | Nil -> O
                  | Cons (n2, l2) -> n2)
        | S (n2) -> (match l1 with
                      | Nil -> O
                      | Cons (n3, l2) -> f1 l2 n2)
  in
    f1
;;
```

Output with modified input, by changing the order of the last example to `[1; 2; 3]`:

```
let list_nth : list -> nat -> nat =
  let rec f1 (l1:list) : nat -> nat =
    fun (n1:nat) ->
      match l1 with
        | Nil -> O
        | Cons (n2, l2) -> ( match f1 l2 O with
                              | O -> ( match n1 with
                                        | O -> n2
                                        | S (n3) -> O)
                              | S (n3) -> ( match n1 with
                                              | O -> n2
                                              | S (n4) -> S (n3)))
  in
    f1
;;
```

Figure 4. Difference in synthesized program given slightly modified example.

5.2 Sensitivity test

Several test cases are slightly modified by changing the order in which the examples are presented, in order to test MYTH's sensitivity to example specifications. While programs dealing with simply data types such as `bool` or `nat` are synthesized identically to the original cases, MYTH fails to synthesize the same program for more complex data types such as `list`. An example is shown in Figure 4. This finding will be further discuss in Section 6.1.

5.3 New test cases

Five new test cases, as listed below, are successfully synthesized:

- `bool_iff : bool -> bool -> bool`, which checks the *if and only if* condition of two `bool`.
- `nat_eq : nat -> nat -> bool`, which checks if two `nat` numbers are equal.
- `list_has : nat -> list -> bool`, which checks if a `list` contains a specific `nat` number.
- `list_index : nat -> list -> nat`, which returns the index of the first instance of a specific `nat` number in a `list`, returns the index of the last element if specified element not found in list.
- `list_eq : list -> list -> bool`, which checks whether two lists contain the exact same elements (in same order).

`bool_iff` and `nat_eq` are quite trivial, whereas the three test cases of `list` type require more careful tuning of the example specification and definition of helper functions. The synthesis data is listed in Figure 5. The detailed findings will again be further discussed in Section 6.2.

Test	#Ex	#N	Time (s)
nat_eq	5	16	0.0104
list_eq	16	23	13.6328
list_index	10	19	1.751
list_has	7	19	0.638
bool_band	4	9	0.0038

Figure 5. MYTH test results using new test cases, where #Ex is the number of examples used, #N is the size of the result

6. Discussion

6.1 Sensitivity to example specifications

The most important finding from the tests is that, although not an issue with more premitive data types (`bool`, `nat`), MYTH is extremely sensitive to example specifications for programs with more complex types. Sometimes, a very slight change in the example specification could result in vastely different results or even failure in synthsis. In some extreme cases, the synthesis changes significantly even by simply adding an extra redundant example. Osera and Zdancewic (2015) did make a note of this issue in their discussion, stating that the examples should be specified in such way that they imitate each recursive call of the target function, which is usually not case in real-life input-output examples available to be used for program synthesis. This is likely going to be the most important challenge to overcome in order for this method to have real-life applications.

6.2 Context size and helper functions

Some of the new test cases require a larger context, *i.e.*, predefined helper functions. This allows the program to be synthesized correctly but significantly increased the synthsis time. This finding is in agreement of some cases presented by Osera and Zdancewic (2015) in the original publication. The authors attributed this issue to the significantly larger set of possible matches for E-guessing given by the helper function. They also proposed possible ways of mitigate this problem, such as optimizing search heuristic.

It was pointed out by Osera and Zdancewic (2015) that the synthesis tend to be driven into a counter-intuitive pattern: *inside-out recursion*, where the synthesized function tends to match for output types from recursive calls itself, rather than input types. Although there are successful cases where a synthesized function with an *inside-out recursion* does perform the correct task, it is observed during the testing that most of these functions are too specific to satisfying the given set of examples and generally correct. It is also observed, however, that extra context and helper functions tend to decrease the chance of getting an *inside-out recursion*, at the cost of synthsis time.

6.3 Other findings

As already pointed out in Section 6.2, one possible immediate improvement to this synthesis method is to further optimize the search strategy and/or heuristic. The current setting of MYTH has a rather naive, but balanced alternation between term generation and refinement, with a generation timeout of 2.5ms. Based on the description by Osera and Zdancewic (2015) as well as the traces in the source code, it appears that the authors have tested several different possibilities and found the current setting to be the most stable one. During the testing, a few other search strategies, such as a longer timeout for term generation and increasing the size of scrutinee growth, have been attempted. They all prove to be less optimal than the existing stable strategy. Due to time and resource constraint, the different possible search strategies cannot be exhaustively tested, but this should definitely be the main concentration of future works on this topic.

Another minor finding of the MYTH prototype is an issue in output formatting. Although MYTH is not compatible with premitive data types in OCaml, such as `int` or `string`, the authors added an extra layer of parser to translate normal repersentation of a natural number or a list (*e.g.*, `0, 1, [0,1]` etc) into algebraic types `nat` and `list` recognizable by MYTH. This is certainly done for convenience in specifying input-output examples. During the output step of the synthesized program, the authors reformat the `nat` and `list` types back into a normal reprsentation. Althought this is more advantageous from an aesthetics point of view, it prevents the synthesized code from being run directly in the context of OCaml. This may potentially cause issues for future wrapper methods for test/application purposes. An adapted implementaion of MYTH is created to resolve this issue by keeping the data types as they are during the output formatting step.

7. Conclusion

This project provides an overview of the underlying theories of program synthesis with a concentrated attention on the new synthesis language λ_{syn} - a proof-theory-based, example-and-type-directed synthesis algorithm - developped by Osera and Zdancewic (2015). Reproducibility as well as sensitivity tests are performed on the prototype of λ_{syn}'s artifact: MYTH, revealing strengths and weakenesses of this method. Five additional test cases are also successfully synthesized using the MYTH artifact. Interesting results are discovered during the testing and discussed in this report. Optimization of the search strategy has been identified during this project to be the key component that needs to be improved in future works, if time and resources permit. An optimal search strategy will allow this method to accomodate for more formats of example-specification while keeping the synthesis search space reasonably small. Once achieved, this synthesis method has great potential of becoming a usable tool with broader applications.

A. Electronic Appendix

All work done for this project, including an copy of the MYTH artifact with customized implementation (output format fix and new test cases), as well as a copy of this report and its source code, can be found in a repository dedicated to this project at https://github.com/Hachiko99/Project-527.

References

D. Basin, Y. Deville, P. Flener, A. Hamfelt, and J. Fischer Nilsson. *Program Development in Computational Logic: A Decade of Research Advances in Logic-Based Program Development*, chapter Synthesis of Programs in Computational Logic, pages 30–65. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. ISBN 978-3-540-25951-0.

D. Bjorner, N. D. Jones, and A. Ershov. *Partial Evaluation and Mixed Computation: Proceedings of the IFIP TC2 Workshop, Gammel Avernaes, Denmark, 18-24 Oct., 1987.* Elsevier Science Inc., 1988.

R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM (JACM)*, 24(1):44–67, 1977.

H. B. Curry, R. Feys, W. Craig, J. R. Hindley, and J. P. Seldin. Combinatory logic. 1972.

J. Frankle, P.-M. Osera, D. Walker, and S. Zdancewic. Example-directed synthesis: a type-theoretic interpretation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 802–815. ACM, 2016.

T. Freeman and F. Pfenning. *Refinement types for ML*, volume 26. ACM, 1991.

C. Green. Application of theorem proving to problem solving. Technical report, DTIC Document, 1969.

A. Kolmogorov. The theory of functions of a real variable. *Science in the USSR during fifteen years: Mathematics*, 1932.

C. Kreitz. Program synthesis. In *Automated DeductionA Basis for Applications*, pages 105–134. Springer, 1998.

Z. Manna and R. J. Waldinger. Toward automatic program synthesis. *Communications of the ACM*, 14:151–165, 1971.

P.-M. Osera and S. Zdancewic. Type-and-example-directed program synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 619–630. ACM, 2015.

R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):402–454, 1982.

F. Pfenning. Refinement types for logical frameworks. In *Informal Proceedings of the Workshop on Types for Proofs and Programs*, pages 285–299, 1993.

W. W. Tait. Intensional interpretations of functionals of finite type i. *The journal of symbolic logic*, 32(02):198–212, 1967.