Report on

Synthesizing Data Structure Transformations from Input-Output Examples

as part of

CS 613 Course Project

Submitted By:

Nithin S : 16305R007

Suresh Sudhakaran : 163050021

Table Of Contents

Introduction

The aim of this course project is to study the paper “Synthesizing Data Structure Transformations from Input-Output Examples” by John K Feser, Swarat Chaudhuri and Isil Dillig. We are presenting a brief overview of the paper, verifying the results and runtime guarantees, and discussing various aspects and limitations of the method used.

Overview

The paper presents a method for example-guided synthesis of functional programs over recursive data structures. Given a set of input-output examples, the method synthesizes a program in a functional language with higher order combinators like map and fold.

The approach combines 3 ideas: Inductive Generalization, deduction, and enumerative search. Generalization creates a hypotheses about the structure of the target program. For each hypotheses, Deduction infers new input/output examples for the missing subexpressions. This leads to a new subproblem where the goal is to synthesize expressions within each hypothesis. Since not every hypothesis can be realized into a program that fits the examples, a combination of best-first enumeration and deduction is used to search for a hypothesis that meets the requirements.

This method is implemented in a tool called λ^2. The authors had tested this tool on a large set of synthesis problems and recorded their results and observations. We cloned their repository[2] and have attempted to reproduce these results on an extended set of problems. We have also recorded our running times and observations on the limitations of the method.

Algorithm

The program uses inductive generalization to generate new hypothesis from the examples. If the new hypotheses is closed, it verifies if the generated concrete program satisfies the top-level input-output examples. If it does, it presents this concrete program as the output, otherwise it backtracks and pick another hypothesis from the queue.

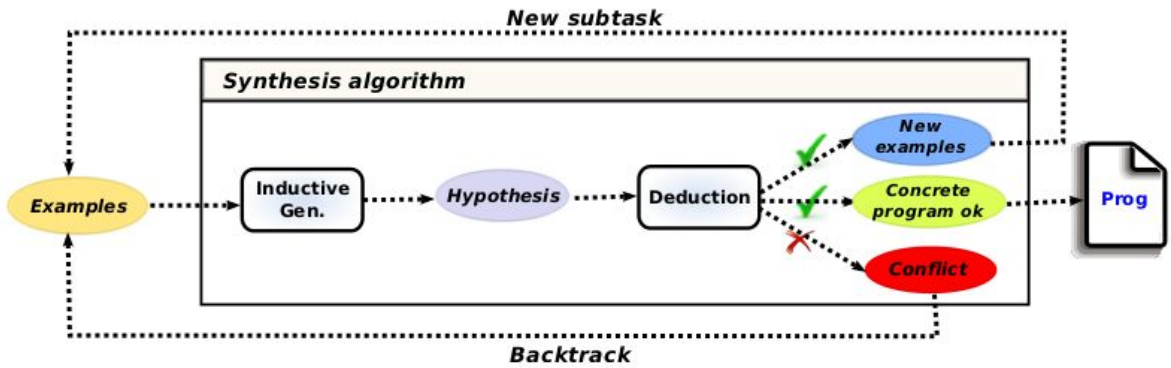

Figure 1. High-level overview of our synthesis algorithm

Inductive Generalization:

A hypothesis is a program which possibly has free variables (holes). The objective of the synthesis function is to fill the holes and generate a closed and concrete program that satisfies the top-level input/output examples. From a standard library consisting of possible hypothesis, the program selects those hypotheses which match the required type for input/output examples in increasing order of cost.

Deduction:

There are deductive rules for combinators (map, filter, foldl etc) to detect conflicts and generate new input/output examples in absence of any conflicts. For example:
Let A and B be lists and A → B be an input/output pair.

- A conflict is detected if the map combinator is used for if A and B are of different lengths.

- A conflict is detected if two equal values in A map to two different elements in B.
- If no such conflict is detected, then we now need to deduce a new hole in the combinator map ‘f’ such that f maps A_i to B_i .

Similarly deductive rules are defined for other combinators in the file **src/l2-lib/higher_order_deduction.ml**

The deduce procedure used in this program is sound and incomplete. Soundness ensures that the program will never reject a valid open hypothesis. But even if all the subtasks generated while solving a problem have a solution, due to incompleteness it cannot be guaranteed if the solution to the original problem exists. To compensate for this, any closed hypothesis is checked for consistency with the user provided input/output examples.

Best-first enumeration:

The algorithm tries to minimize the use of conditional branches and generate the most general program that satisfies the input/output examples. We assign costs to the combinators (Refer Appendix B). Best-first enumeration comes into picture when the program picks a subtask from the queue. At this stage, it would pick the subtask with the least cost associated with it.

```
SYNTHESIZE(E_in)
 1  Q ← {(f, f, E)}        // f is a fresh variable name
 2  while Q ≠ ∅
 3  do pick (e, f, E) from Q such that e has minimal cost
 4     if e is closed
 5       then if CONSISTENT(e, E_in)
 6               then return e
 7               else continue
 8     τ ← TYPEINFER(E)
 9     H ← INDUCTIVEGEN(τ)
10     for h ∈ H
11     do e' ← e[h/f]
12        if e' is closed
13          then Q ← Q ∪ {(e', ⊥, ∅)}
14          else for f* ∈ HOLES(e')
15               do E* ← DEDUCE(e', f*, E)
16                  if E* = ⊥ then break
17                  Q ← Q ∪ {(e', f*, E*)}
18  return ⊥
```

Figure 2. Synthesis procedure.

Observations and Conclusions

We ran the programs on a Macbook Pro , 2.7GHz Intel Core i5 (2 cores) with 8gb ram. In the programs that are generated, we observed that arguments for operators such as cons, foldl and map are flipped, they are ordered differently than in Haskell. Below is the tabulation of the runtimes we observed.

Name	Runtime (paper)	Runtime in seconds (observed)	Description
add	0.04	0.185	Add a number to each element of a list.
append	0.23	0.827	Append an element to a list.
appendt	1.03	1.39	Append an element to each node in a tree of lists.
concat	0.13	0.137	Concatenate two lists together.
count_leaves	0.44	1.3	Count the number of leaves in a tree.
count_nodes	0.62	0.794	Count the number of nodes in a tree.
dedup	231.05	1.15	Remove duplicate elements from a list.
droplast	316.39	0.315	Drop the last element in a list.
dropmax	0.12	0.177	Drop the largest number(s) in a list.
dupli	0.11	0.103	Duplicate each element of a list.
evens	7.39	0.146	Remove the odd numbers from a list.
flatten	0.08	0.493	Flatten a tree into a list.
flattenl	0.08	0.656	Flatten a tree of lists into a list.
height	0.10	52	Return the height of a tree.

incrs	0.12	0.833	Increment each number in a list of lists.
incrt	0.02	0.150	Increment each node in a tree by one.
join	0.43	34	Concatenate a list of lists together.
last	0.02	0.123	Return the last element in a list.
length	0.01	0.814	Return the length of a list.
max	0.46	1.4	Return the largest number in a list.
maxt	10.59	0.925	Return the largest number in a tree.
member	0.35	0.123	Check whether an item is a member of a list.
multifirst	0.01	0.180	Replace every item in a list with the first item.
multilast	0.08	0.180	Replace every item in a list with the last item.
prependt	0.01	0.126	Prepend an element to each list in a tree of lists.
replacet	4.02	0.232	Replace one element with another in a tree of lists.
reverse	0.01	0.843	Reverse a list.
shiftl	0.89	0.813	Shift all elements in a list to the left.
sum	0.01	3.3	Return the sum of a list of integers.
sumt	0.59	2.79	Sum the nodes of a tree of integers.
sumtrees	12.10	1.5min	Return the sum of each tree in a list of trees.

For eg. Length returns the following

```
fun a -> foldl a (fun c b -> c + 1)
```

We could not get the following examples to synthesize within 20 minutes.
cprod, insertn, leaves, searchnodes, selectnodes, shiftr, tconcat.

Update : The L2 tool has been modified heavily since the paper was originally published. The runtime we have observed throughout this report are based on the latest version available as on November 2016.

The developer of L2 has released a version tagged “PLDI_SRC” which generated the results provided in the paper. We were unable to get this version working since some dependencies seem to have been deprecated.

We wrote the following examples to test the synthesis method.

nth	0.235	Return the nth element in a list.
removek	8.5 min	Remove the kth element from list.
replicate	0.894	Duplicate each list element n times.
split	timeout	Split the list into 2: first n and rest.
sumodd	timeout	Find the sum of odd elements in a list

We observed that for some combinations of higher order functions, the program times out. For example, a program like sumodd can be intuitively written by a programmer using sum, filter and odd functions, but the sumodd test case we wrote timed out.

We also noticed that the program managed to catch patterns the user may not have intended. This is to be expected from a synthesis program.
For eg. While writing sumoddp testcases, we wrote the following as our input/output example

```
{
  "name": "sumoddp",
  "description": "",
  "kind": "examples",
  "contents": {
    "examples": [
      "(sumoddp []) -> 0",
      "(sumoddp [1]) -> 1",
      "(sumoddp [2 1]) -> 2",
```

```
      "(sumoddp [3 2 4]) -> 7"
    ],
    "background": []
  }
}
Output: fun a -> sum (intersperse a (neg 1))
```

Our intentions was to find the sum of elements at odd indices. But the program managed to find another pattern that we overlooked.

Our belief is that this occurs because the default cost of all functions in stdlib.ml is 1 unless otherwise specified. This gives undue preference functions like intersperse (cost:1) over combinators such as map (cost: 2) and filter (cost:2)

Bibliography

[1] John K. Feser , Swarat Chaudhuri , Isil Dillig, Synthesizing data structure transformations from input-output examples
[2] https://github.com/jfeser/L2.git

Appendix A

1.stdlib.ml: This is a file containing components to be used in synthesis. We can use another file in place of this using command line arguments.

The contents of stdlib.ml is given below:

```
builtin +, -, /, *, %, =, <>, <, <=, >, >=, &&, ||, not, if,
::, car, cdr, tree, value, children
let inf = 4611686018427387903
let rec foldr = fun l f i ->  if l = [] then i else  f (foldr
(cdr l) f i) (car l)
let rec foldl = fun l f i -> if l = [] then i else foldl (cdr
l) f (f i (car l))
let rec map = fun l f -> if l = [] then [] else (f (car l)) ::
map (cdr l) f
let rec filter = fun l f -> if l = [] then [] else let rest =
filter (cdr l) f in
          if f (car l) then (car l) :: rest else rest
```

On top of these, stdlib.ml contains the following functions:

```
filteri, mapi, mapt, foldt, merge, take , zip, intersperse,
append, reverse, concat, drop, sort, dedup, len, nth, exists,
split_n, unzip, last, count, range, sub, list_and, list_or,
repeat, delete_first, delete_all, union, intersect, replace,
sum, mean, median, min, max, product, pow, neg, fact, abs,
even, odd.
```

In case we want to deduct a program without using some of these functionalities, there is an option to blacklist a set of these functions in the input json file.

Appendix B

default_cost.json:This file contains the default cost for functions used in synthesis. The file contents are.

```
{
    "num" : 1,
    "bool": 1,
    "hole": 0,
    "lambda": 1,
    "_let": 1,
    "list": 1,
    "tree": 1,
    "var ": 1,
    "call": {
        "foldr": 3,
        "foldl": 3,
        "foldt": 3,
        "map": 2,
        "mapt": 2,
        "filter": 2
    },
    "call_default": 1
}
```

The default value has been set to 1. Thus the predefined functions in stdlib takes preference over higher order combinators like map and foldr.

Appendix C

Running Instructions:

- Instructions to set up L2 (from github wiki): https://github.com/jfeser/L2/wiki
- Running Command:
  ./l2.native synth -dd higher_order -l components/stdlib.ml specs/example.json

    - Here the stdlib.ml file has the predefined functions to be used in synthesis. We can provide our own library in place of stdlib.ml .
    - example.json should contain the input/output examples to be used in synthesis. The format of the json file can be found in any of the examples in specs folder.