ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

# Inductive Synthesis from Higher-Order Functions

Master Thesis

Alexandra Maximova

August 15, 2016

Advisors: Prof. Dr. Martin Vechev, Dimitar Dimitrov
Department of Computer Science, ETH Zürich

**Abstract**

This example thesis briefly shows the main features of our thesis style, and how to use it for your purposes.

# Contents

Chapter 1

# Introduction

This is version `v1.4` of the template.

We assume that you found this template on our institute's website, so we do not repeat everything stated there. Consult the website again for pointers to further reading about LaTeX. This chapter only gives a brief overview of the files you are looking at.

## 1.1 Features

The rest of this document shows off a few features of the template files. Look at the source code to see which macros we used!

The template is divided into TeX files as follows:

1. `thesis.tex` is the main file.
2. `extrapackages.tex` holds extra package includes.
3. `layoutsetup.tex` defines the style used in this document.
4. `theoremsetup.tex` declares the theorem-like environments.
5. `macrosetup.tex` defines extra macros that you may find useful.
6. `introduction.tex` contains this text.
7. `sections.tex` is a quick demo of each sectioning level available.
8. `refs.bib` is an example bibliography file. You can use BibTeX to quote references. For example, read if you can get a hold of it.

### 1.1.1 Extra package includes

The file `extrapackages.tex` lists some packages that usually come in handy. Simply have a look at the source code. We have added the following comments based on our experiences:

**REC** This package is recommended.

**OPT** This package is optional. It usually solves a specific problem in a clever way.

**ADV** This package is for the advanced user, but solves a problem frequent enough that we mention it. Consult the package's documentation.

As a small example, here is a reference to the Section *Features* typeset with the recommended *varioref* package:

> See Section 1.1 on the preceding page.

### 1.1.2 Layout setup

This defines the overall look of the document – for example, it changes the chapter and section heading appearance. We consider this a 'do not touch' area. Take a look at the excellent *Memoir* documentation before changing it.

In fact, take a look at the excellent *Memoir* documentation, full stop.

### 1.1.3 Theorem setup

This file defines a bunch of theorem-like environments.

**Theorem 1.1** *An example theorem.*

**Proof** Proof text goes here. □

Note that the q.e.d. symbol moves to the correct place automatically if you end the proof with an `enumerate` or `displaymath`. You do not need to use `\qedhere` as with *amsthm*.

**Theorem 1.2 (Some Famous Guy)** *Another example theorem.*

**Proof** This proof

1. ends in an enumerate. □

**Proposition 1.3** *Note that all theorem-like environments are by default numbered on the same counter.*

**Proof** This proof ends in a display like so:

$$f(x) = x^2. \qquad \square$$

### 1.1.4 Macro setup

For now the macro setup only shows how to define some basic macros, and how to use a neat feature of the *mathtools* package:

$$\lvert a \rvert, \quad \left\lvert \frac{a}{b} \right\rvert, \quad \lvert \frac{a}{b} \rvert.$$

Chapter 2

# Related Work

Try to answer the following three question for each paper read:

1. What is new in this approach? Or better, what is the approach. Describe technically the approach, so that you can answer technical questions.
2. What is the trick? (Why are they better than others?)
3. Which examples they can do really well? What kind of examples do they target? What is the most complicated thing they can generate?

Nadia Polikarpova 2015
here is a talk: `http://research.microsoft.com/apps/video/default.aspx?id=255528&l=i`
and here is the code: `https://bitbucket.org/nadiapolikarpova/synquid`
In [4] Synquid is proposed. Refinement types (types decorated with logical predicates) are used to prune the search space. SMT-solvers are used to satisfy the logical predicates. The key is the new procedure for type inference (called modular refinement type reconstruction), which thank to its modularity scales better than other existing inference procedures for refinement types. Programs can therefore be type checked even before they are put together. Examples that this tool is able to synthesize include several sorting algorithms, binary-search tree manipulations, red-black tree rotation as well as other benchmarks also used by other tools (**TODO:** read about these benchmarks and write if there is something interesting). The user specifies the desired program by providing a goal refinement type.

Feser 2015
The tool proposed in [2] is called $\lambda^2$ and generates its output in $\lambda$-calculus with algebraic types and recursion. The user specifies the desired program providing input-output examples. No particular knowledge is required from the user, as was demonstrated using random input-output examples

The examples are inductively generalized in a type-aware manner to a set of hypotheses (programs that possibly have free variables). The key idea are the hard-coded deduction rules used to prune the search space depending on the semantics of some of the higher-order combinators (map, fold, filter and a few others). Deduction is also used to infer new input-output examples in order to generate the programs needed to fill in the holes in the hypotheses. This tool is able to synthesize programs manipulating recursive data structures like lists, trees and nested data structures such as lists of lists and trees of lists. The examples that require much more time to be synthesized than the others are dedup (remove duplicate elements from a list), droplast (drop the last element in a list), tconcat (insert a tree under each leaf of another tree), cprod (return the Cartesian product of a list of lists), dropmins (drop the smallest number in a list of lists), but all of them are synthesized under 7 minutes.

Kincaid 2013
In [1] Escher is presented, an inductive synthesis algorithm that learns a recursive procedure from input-output examples provided by the user. The user must provide a ”closed” set of examples, otherwise recursion cannot be handled properly The target language is untyped, first-order and purely functional. The algorithm is parametrized by components that can be instantiated differently to suit different domains. The approach combines enumerative search and conditional inference. The key idea is to use a special data structure, a *goal graph*, to infer conditional branches instead of treating `if-then-else` as a component. Observational equivalence is also used to prune the search space. Programs with the same value vectors (output of the program when applied to the inputs of the input-output examples) are considered equivalent and only one of them is synthesized. An implementation of the tool was tested on a benchmark consisting of recursive programs (including tail-recursive, divide-and-conquer and mutually recursive programs) drawn from functional programming assignments and standard list and tree manipulation programs. For all examples the same fixed set of components was used. The tool is able to synthesize all of them quickly. There is very little information on how many input-output examples were needed to synthesize the benchmarks and how difficult it is for a non-experienced user to come up with a ”closed” set of examples.

Osera 2015
The tool in [3] is called Myth and uses not only type information but also input-output examples to restrict the search space. The special data structure used to hold this information is the *refinement tree*. This system can synthesize higher-order functions, programs that use higher order functions and work with large algebraic data types.
There is an ML-like type system that incorporates input-output examples. Two pieces: a *refinement tree* and an enumerative search.

Two major operations: refine the goal type and the examples and guess a term of the right type that matches the examples.
A small example to show what does the procedure. The user specifies a goal type incorporating input-output examples as well as the "background": the types and functions that can be used.

```
stutter :
```

Chapter 3

# Benchmarks

Some programs over numbers, some over lists, some over lists of lists and some over trees (What kind of trees?). For every program, try to get a sample implementation.
Types needed: `Int, [a], Tree a`
Basic components needed: arithmetic (`+, -, *, /`), relation (`<, <=, ==, !=, >=, >`),

1. max of two numbers
   (hopefully) the easiest program

   ```
   max :: Int -> Int -> Int
   max 0 0 == 0
   max 1 0 == 1
   max 0 1 == 1
   max x y = if x > y then x else y
   ```

   What to do with conditionals?

2. square a number

   ```
   square :: Int -> Int -> Int
   square 0 == 0
   square 1 == 1
   square 2 == 4
   square 3 == 9
   square x = x * x
   ```

   That is, basic arithmetic operations like `+ - * /` should be provided

3. tetrahedral numbers

```
tetrahedral :: Int -> Int
tetrahedral 1 == 1
tetrahedral 2 == 4
tetrahedral 3 == 10
```

closed form solution

```
tetrahedral n = n * (n+1) * (n+2) / 6
```

iterative solution

```
tetrahedral n = scanl1 (+) (scanl1 (+) [0..]) !! n
```

Another iterative solution (without infinite lists)

```
tetrahedral n = foldl1 (+) (scanl1 (+) (enumFromTo 1 n))
```

Components needed: `scanl1`, `!!`
Interestingly the iterative version is much faster than the closed form solution

4. prime test
   I think this is too difficult

```
prime :: Int -> Int
prime 1 == 0
prime 2 == 1
prime 3 == 1
prime 4 == 0
prime 25 == 0
prime 29 == 1
prime n = minimum (1 : (map (mod n) (enumFromTo 2 (subtract 1 n
```

Components needed: `map`, `mod`, `minimum`, `enumFromTo`, `subtract`

5. average

```
average :: [Int] -> Int
average [1] == 1
average [1,3] == 2
average [1,2,3,6] == 6
average xs = (sum xs) `div` (length xs)
```

6. movingAverage (forward)

```
movingAverage :: Int -> [Int] -> [Int]
movingAverage 1 [1,2,3] == [1,2,3]
movingAverage 2 [1,2,3] == [2,2,3]
movingAverage 3 [3,2,4,1,5,2] == [3,2,3,2,3,2]
movingAverage n xs = map (average . take n) (init $ tails xs)
```

Components needed: `tails` from `Data.List` and `average` (one of the benchmarks), as well as `map, take` and `init` from `Prelude`.

7. movingSum (backward)

```
movingSum :: Int -> [Int] -> [Int]
movingSum 1 [1,2,3] == [1,2,3]
movingSum 2 [1,2,3] == [1,3,5]
movingSum 3 [4,8,6,-1,-2,-3,-1,3,4,5] == [4,12,18,13,3,-6,-6,-1,6,12]
movingSum n xs = scanl1 (+) (zipWith (-) xs (replicate n 0 ++ xs))
```

8. matrix multiplication
   I wouldn't take this example. Matrices as lists of lists are unnatural.
   **TODO:** Ask if you really have to do it. If yes, search an implementation

9. waterflow problem
   Given an array of "wall" heights, determine the volume of the puddles that can form if it rains.

```
water :: [Int] -> Int
water [1,2,3] == 0
water [5,2,5] == 3
water [2,3,1,6,1] == 2
water h = sum $
      zipWith (-)
        (zipWith min (scanl1 max h) (scanr1 max h))
        h
```

10. horner schema to evaluate polynomials

```
horner :: [Int] -> Int -> Int
horner [1,2,3] 1 == 6
horner [1,2,3] 2 == 11
horner [4,3,2] 3 == 47
horner p x = foldl1 ((+) . (x *)) p
```

Problem: we do not generate lambda's. Do we generate functions like `(x *)`?

11. sum-under, sum all integers up to the argument

```
sum_under :: Int -> Int
sum_under 0 == 0
sum_under 1 == 1
sum_under 2 == 3
sum_under 3 == 6
sum_under 4 == 10
sum_under n = sum [1..n]
```

Components needed: `sum, enumFromTo`

12. factorial

```
factorial :: Int -> Int
factorial 0 == 0
factorial 1 == 1
factorial 3 == 6
factorial 5 == 120
factorial n = product [1..n]
```

    interesting for intermediate states

13. maximum of a list
    I don't know (yet) how to specify a "global property" like greater or smaller than all other elements in a list in Synquid. Moreover, it seems a difficult property to extract from input-output examples.

```
maximum :: [Int] -> Int
maximum [1,3,2] == 3
maximum [4,2,1] == 4
maximum [1,3,5] == 5
maximum xs = foldr max (head xs) xs
```

    Or just use the `maximum` function from `Prelude`, if it is given as a component

14. append two lists
    The specification given by Nadia does not synthesize the usual append function. Maybe it's better to let her know...
    Although it's possible to synthesize append in Synquid.
15. length of a list
    Can be also interesting for intermediate states
16. list reversal
17. bagsum: `[far,bar,gar,bar,bar,far] -> [(bar,3),(far,2),(gar,1)]`
    Seems difficult and maybe intermediate states can be helpful
18. map
    Isn't it a higher order function? I thought we synthesize only first order functions.
19. zipWith
    it's a higher order function as well
20. list drop
21. droplast, drop the last element of a list

22. dropmax, drop the greatest element of a list
    $\lambda^2$ takes much more time to synthesize droplast than dropmax. Why?
23. dedup, remove duplicates from a list
    $\lambda^2$ requires more time
24. sort by length (on lists of lists)
25. dropmins
    $\lambda^2$ required more time to synthesize it
26. lasts, last element of every list
    another program on nested lists
27. member of the tree
    Something with trees. Membership seems a difficult thing to learn from input-output examples.
28. count leaves it a tree
29. nodes at level

    Nadia has more complicated examples with Red-Black-Trees, AVL-trees and different sorting algorithms

Appendix A

# Dummy Appendix

You can defer lengthy calculations that would otherwise only interrupt the flow of your thesis to an appendix.

# Bibliography


[1] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. Recursive program synthesis. In *Proceedings of the 25th International Conference on Computer Aided Verification*, CAV'13, pages 934–950, Berlin, Heidelberg, 2013. Springer-Verlag.

[2] John K. Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 229–239, New York, NY, USA, 2015. ACM.

[3] Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 619–630, New York, NY, USA, 2015. ACM.

[4] Nadia Polikarpova and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. *CoRR*, abs/1510.08419, 2015.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

# Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

---

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

**Title of work** (in block letters):

**Authored by** (in block letters):
*For papers written by groups the names of all authors are required.*

| **Name(s):** | **First name(s):** |
|---|---|
| | |
| | |
| | |
| | |
| | |

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

| **Place, date** | **Signature(s)** |
|---|---|
| | |
| | |
| | |
| | |
| | |

*For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.*