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

TODO: write me :)

Contents

Chapter 1

Introduction

1.1 About program synthesis in general

put in some context, link 2

1.2 Problem definition

have many components, put them together into a program, no lambdas, no if-then-else, no recursion

1.3 Contributions

Evaluation, exploring the baseline algorithm, exploring the search space

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 λ^2 and generates its output in λ-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

Top down type driven synthesis

TODO: starting words In this chapter we will present and formally define all concepts and algorithms used by the synthesis system TAMANDU.

3.1 Problem

introductory example consider we want to generate

```
replicate :: ∀ X. Int → X → List X

replicate n x = map Int X (const X Int x) (enumTo n)
```

3.2 System F

The exposition will closely follow Pierce cite the book properly.

3.2.1 Terms and Types

- similar to Pierce
- except holes, components and free variables
- the search space is not the whole language but only a subset of it

Real System F

$$t ::= x \mid \lambda x : T.t \mid t\ t \mid \Lambda X.t \mid t\ [T] \qquad \textit{(terms)}$$

$$T ::= X \mid T \to T \mid \forall X.T \qquad \textit{(types)}$$

$$\Gamma ::= \emptyset \mid \Gamma \cup x : T \qquad \textit{(variable bindings)}$$

We use an extension of System F featuring holes $?x$, input variables i as well as named library components c and named types C. The use of the names enables recursion in the definition of library components and types. Named types also support type parameters. The number of type parameters supported by a named type is denoted as K in its definition.

Question: where is the output? Can you define input and output pairs? Our system

$$t ::= x \mid \lambda x : T.t \mid t\ t \mid \Lambda X.t \mid t\ [T] \mid c \mid ?x \mid i \qquad \textit{(terms)}$$

$$T ::= X \mid T \to T \mid \forall X.T \mid ?X \mid I \mid C\ [T, \ldots, T] \qquad \textit{(types)}$$

$$\Gamma ::= \emptyset \mid \Gamma \cup x : T \mid \Gamma \cup X \qquad \textit{(variable bindings)}$$

$$\Xi ::= \emptyset \mid \Xi \cup ?x : T \mid \Xi \cup ?X \qquad \textit{(hole bindings)}$$

$$\Phi ::= \emptyset \mid \Phi \cup i = t : T \mid \Phi \cup I = T : K \qquad \textit{(input variable bindings)}$$

$$\Delta ::= \emptyset \mid \Delta \cup c = t : T \mid \Delta \cup C = T : K \qquad \textit{(library components)}$$

Question: do we need the definitions of library components for synthesis? We use them only for evaluation, but we always evaluate programs during synthesis. Same for the input variables. Actually, for each I/O-example we get the pair (Φ, o), where Φ instantiates all input variables and input types of a program and o is the expected output. Subset of our system that builds the search space

$$t ::= t\ t \mid t\ [T] \mid c \mid ?x \mid i \qquad \textit{(terms)}$$

$$T ::= X \mid T \to T \mid \forall X.T \mid ?X \mid I \mid C\ [T, \ldots, T] \qquad \textit{(types)}$$

$$\Xi ::= \emptyset \mid \Xi \cup ?x : T \mid \Xi \cup ?X \qquad \textit{(hole bindings)}$$

$$\Phi ::= \emptyset \mid \Phi \cup i = t : T \mid \Phi \cup I = T : K \qquad \textit{(input variable bindings)}$$

$$\Delta ::= \emptyset \mid \Delta \cup c = t : T \mid \Delta \cup C = T : K \qquad \textit{(library components)}$$

A program is the quadriplet $\{\Xi, \Phi, \Delta \vdash t :: T\}$. A term is called *closed* if it does not contain holes. A program is closed, if Ξ is empty and t and T do not contain holes.

3.2.2 Evaluation semantics

We usually evaluate only closed programs. Eager evaluation. Rules.

3.2.3 Encodings

Note that in the definition of types we do not see familiar types such as booleans, integers, lists or trees. All these types can be encoded in System F using either the Church's or the Scott's encoding [**?**]. We opted for the Scott's encoding because it's more efficient in our case. Scott's booleans coincide with Church's booleans and are encoded as follows.

```
Bool  = ∀R. R → R → R
true  = ΛR. λx1:R. λx2:R. x1
      : Bool
false = ΛR. λx1:R. λx2:R. x2
      : Bool
if-then-else = ΛX. λb:Bool. λt:X. λf:X. b [X] t f
             : ∀X. Bool → X → X → X
```

Scott's integers differ from Church's integers as they are more suitable for pattern matching. because they don't unwrap the whole integer every time.

```
Int = ∀R. R → (Int → R) → R
zero = ΛR. λz:R. λs:Int → R. z
     : Int
succ = λ n:Int. ΛR. λz:R. λs:Int → R. s n
     : Int → Int
case = ΛR. λn:Int. λa:R. λf:Int → R. n [R] a f
     : ∀R. Int → R → (Int → R) → R
```

Analogously, Scott's lists are a recursive type and naturally support pattern matching.

```
List X = ∀R. R → (X → List X → R) → R
nil = ΛX. ΛR. λn:R. λc:X → List X → R. n
    : ∀X. List X
con = ΛX. λx:X. λxs:List X. ΛR. λn:R. λc:X → List X → R. c x xs
    : ∀X. X → List X → List X
case = ΛX. ΛY. λl:List X. λn:Y. λc:X → List X → Y. l [Y] n c
     : ∀X. ∀Y. List X → Y → (X → List X → Y) → Y
```

3.2.4 Type unification

3.2.5 Type checking

3.3 Search

how do we explore the search space (best-first search). (only priority queue)

3.3.1 Search space

(the successor rules) Use only third system presented in Section System F.

We see the search space as a graph of programs with holes (see third syntax presented in Section System F) where there is an edge between two terms t_1 and t_2 if and only if the judgement *derive* defined below $\Xi, \Phi, \Delta \vdash t_1 :: T_1 \mapsto \Xi', \Phi, \Delta \vdash t_2 :: T_2$ holds between the two.

To express the rules in a more compact form, we introduce *evaluation contexts*. An evaluation context is an expression with exactly one syntactic hole [] in which we can plug in any term. For example, if we have the context $\mathcal{E}$ we can place the term t into its hole and denote this new term by $\mathcal{E}[t]$.

A hole $?x$ can be turned into a library component c from the context Δ or an input variable i from the context Φ. The procedure fresh(T) transforms universally quantified type variables into fresh type variables $?X$ not used in Ξ. The notation $\sigma(\Delta)$ denotes the application of the substitution σ to all types contained in the context Δ.

$$\frac{c : T_c \in \Delta \qquad \sigma \text{ unifies } T \text{ with fresh}(T_c)}{\Xi, \Phi, \Delta \vdash ?x :: T \mapsto \sigma(\Xi \setminus \{?x : T\}), \Phi, \Delta \vdash c :: \sigma(T)} \text{ D-VarLib}$$

$$\frac{i : T_i \in \Phi \qquad \sigma \text{ unifies } T \text{ with fresh}(T_i)}{\Xi, \Phi, \Delta \vdash ?x :: T \mapsto \sigma(\Xi \setminus \{?x : T\}), \Phi, \Delta \vdash i :: \sigma(T)} \text{ D-VarInp}$$

A hole can also be turned into a function application of two new active holes.

$$\frac{?X \text{ is a fresh type variable} \qquad \Xi' = \Xi \setminus \{?x : T\} \cup \{?x_1 : ?X \to T, ?x_2 : ?X, ?X\}}{\Xi, \Phi, \Delta \vdash ?x :: T \mapsto \Xi', \Phi, \Delta \vdash ?x_1\ ?x_2 :: T} \text{ D-VarApp}$$

In all other cases we just choose a hole and expand it according to the three rules above.

$$\frac{\Xi, \Phi, \Delta \vdash ?x :: T_1 \mapsto \Xi', \Phi, \Delta \vdash t_1' :: T_1'}{\Xi, \Phi, \Delta \vdash t[?x] :: T \mapsto \Xi', \Phi, \Delta \vdash t[t_1] :: T[T_1 / T_1']} \text{ D-App}$$

Note that the types of all successor programs unify with the types of their ancestors. Thus, the search is type directed.

3.3.2 Exploration

Write also about stack vs queue of open holes

We explore the search graph using a best first search. We can play with the algorithm in two points (marked in blue): first, we can define which hole to expand first, second, we can choose the compare function of the priority queue.

Input: goal type T, library components Δ, list of input-output examples $[(\Phi_1, o_1), \ldots, (\Phi_N, o_N)]$
queue ← PriorityQueue.empty compare
queue ← PriorityQueue.push queue $\{\Xi, \Phi_1, \Delta \vdash ?x :: T\}$
while *not ((PriorityQueue.top queue) satisfies all I/O-examples)* **do**
  successors ← successor (PriorityQueue.top queue)
  queue ← PriorityQueue.pop queue
  for *all s in successors* **do**
    queue ← PrioriryQueue.push queue s
  end
end
Output: PriorityQueue.top queue

Algorithm 1: Best first search

3.4 Cost functions

3.4.1 nof-nodes

3.4.2 nof-nodes-simple-type

3.4.3 no-same-component

3.4.4 length of the string

3.5 Black list

automatic generation of black list discussed in evaluation A black list is a list of terms. Programs containing a black term as a subterm are not allowed to have successors. Thus, the algorithm above is modified as follows.

3.6 Templates

Top-down type-driven synthesis.

A template is a program with holes. We are interested in templates where all higher-order components are fixed and there are holes for the first-order

```
Input: goal type T, library components Δ, list of input-output
       examples [(Φ_1, o_1), ..., (Φ_N, o_N)], black list [b_1, ..., b_M]
queue ← PriorityQueue.empty compare
queue ← PriorityQueue.push queue {Ξ, Φ_1, Δ ⊢ ?x :: T}
while not ((PriorityQueue.top queue) satisfies all I/O-examples) do
    if not ((PriorityQueue.top queue) contains subterm from black list) then
        successors ← successor (PriorityQueue.top queue)
        queue ← PriorityQueue.pop queue
        for all s in successors do
            queue ← PrioriryQueue.push queue s
        end
    else
        queue ← PriorityQueue.pop queue
    end
end
Output: PriorityQueue.top queue
```

Algorithm 2: Best first search with black list

components. The search space is thus similar to the search space described in 3.3.1, with the exception that Δ contains only the higher-order components. One of the new things are *closed holes* $\underline{?x}$. Those are holes that are supposed to be filled in later with first-order components.

The idea behind the templates is that once the higher-order components are fixed, it should be easy and fast to find a first-order assignment to get the right program. So we could do a limited search from a template and if we do not find a program satisfying all of the I/O-examples we can move quickly to the next template.

3.6.1 Successor rules

The successor rules are very similar to the ones defined in 3.3.1, apart from little modifications. That is, now we have a successor rule to close a hole, and we can not instantiate a hole with an input variable any more, because that is supposed to be done in the next step.

So we can *close* a hole.

$$\frac{T \text{ is a type a first-order component can have}}{\Xi, \Phi, \Delta \vdash ?x :: T \mapsto \Xi, \Phi, \Delta \vdash \underline{?x} :: T} \text{ D-VarClose}$$

We can instantiate a hole with a (higher-order) library component.

$$\frac{c : T_c \in \Delta \qquad \sigma \text{ unifies } T \text{ with fresh}(T_c)}{\Xi, \Phi, \Delta \vdash ?x :: T \mapsto \sigma(\Xi \setminus \{?x : T\}), \Phi, \Delta \vdash c :: \sigma(T)} \text{ D-VarLib}$$

We can instantiate a hole with a function application of two fresh holes.

$$\frac{?X \text{ is a fresh type variable} \qquad \Xi' = \Xi \setminus \{?x : T\} \cup \{?x_1 : ?X \to T, ?x_2 : ?X, ?X\}}{\Xi, \Phi, \Delta \vdash ?x :: T \mapsto \Xi', \Phi, \Delta \vdash ?x_1\ ?x_2 :: T} \text{ D-VarApp}$$

We can expand one of the holes of the program according to one of the three rules above.

$$\frac{\Xi, \Phi, \Delta \vdash ?x :: T_1 \mapsto \Xi', \Phi, \Delta \vdash t_1' :: T_1'}{\Xi, \Phi, \Delta \vdash t[?x] :: T \mapsto \Xi', \Phi, \Delta \vdash t[t_1] :: T[T_1 / T_1']} \text{ D-App}$$

TODO: just cite the rules, don't copy-paste them, they are exactly the same

Chapter 4

Implementation

What could I talk about in this chapter?

- Programming language and compiler version
- put the type definitions and explain them (What are Fun, FUN and BuiltinFun) (built-in integers for speed)
- Library syntax and the type-checking when added to the library?
- eager evaluation, describe evaluator

Chapter 5

Evaluation

What I want to see in this chapter:

- Table of all components
- Table of synthesized programs with synthesis times for the different algorithms
- Figure of the synthesis times of synthesized programs
- Comparison to Feser
- Explain why automatic black list and templates perform so poorly
- Show advantages of using black lists (I know it's trivial)
- Talk about the trivial thing that if you have more functions with the same time you will need much more time to find the program you are looking for.
- Talk about the constants in the cost functions and how they affect the search space???
- Stack versus queue expanding → mention it in the definitions

5.1 Set up

Describe the machine and the testing set up, which components were used, what information did you give to the synthesizer, how many examples were given. What else?

5.2 Cost functions

Compare the different cost functions with each other, explain why are they good for some programs and bad for other. Table.

5.3 Black lists

5.3.1 Benefits of black lists

Table of your super manual black list. Figure that for each cost function compares time with and without black list. Trivial words about pruning search space and useless branches.

5.3.2 Shortcomes of automatically generated black lists

Maybe really bring the automatically generated table. Point at the repetitions. Say that automatically generated I/O-examples are bad and we need a lot of them. Say that you tried only identity pruning, but one could also try to generate, say, the empty list or whatever.

5.4 Templates

Short section explaining that the templates you generate are not the templates you expect and why

Chapter 6

Conclusions

6.1 Conclusions

The baseline is not that bad. Gathered some data about the search space.

6.2 Future Work

Templates done well, augmented examples?

Chapter 7

Baseline

7.1 Syntax

it's the description of the syntax of the programs in the search space as well as how to derive programs from programs.

Rewrite this without mixing syntax with semantics and with the baseline algorithm In the context Γ we store the bindings from library components and input variables to their types.

In the context Δ we store the bindings from active and inactive holes to their types.

Terms are applications of library components or input variables (denoted by x in the grammar), values (denoted by v) and holes ($?x$ and $?\underline{v}$).

We distinguish between active holes $?x$, that can be substituted with other programs, from inactive holes $?\underline{v}$, that can only be replaced with values. To determine the exact value, a heuristic using a mix of symbolic execution and brute force is applied later. Every active hole having a value type (denoted by V in the grammar) can be turned into an inactive hole.

Programs without active holes are considered *closed*. Those are the programs in which we plug in the input and execute as far as possible to see whether it agrees with the expected output.

Values are integer constants (denoted by n for brevity), lists of values and tuples of values. Does it make sense to have a special treatment for list values and tuples? Wouldn't it be better to use library components to construct them?

We use a standard type system featuring functional and universal types, as well as integers, lists and tuples. We distinguish between two kinds of type variables: $?X$ are the type variables that can be instantiated with other types

when it comes to unification, X are the type variables that are already fixed by the goal type and have to remain like that. Universal types are assumed to be used like in Haskell only as outer wrapping of the types of library components and the goal type specified by the user.

TODO: Find a way to format t t as t t and not a tt, same for VV and TT
TODO: Find a way to format it vertically and adding comments to each line. For example, write "inactive hole" on the right of $?\underline{v}$. Maybe it's better to write Γ and Δ as sets and not as grammars, as you are using them as sets. If you leave it like this, $\Delta \setminus \{?x : T\}$ would be undefined.

$$\Gamma ::= \emptyset \mid \Gamma \cup x : T \qquad \textit{(library components)}$$

$$\Delta ::= \emptyset \mid \Delta \cup ?x : T \mid \Delta \cup ?\underline{v} : T \qquad \textit{(holes)}$$

$$t ::= v \mid ?\underline{v} \mid ?x \mid x \mid tt \qquad \textit{(terms)}$$

$$v ::= n \mid v : v \mid [] \mid (v, v) \qquad \textit{(values)}$$

$$T ::= \text{Int} \mid \text{List } T \mid \text{Tuple } TT \mid T \to T \mid \forall X.T \mid X \mid ?X \qquad \textit{(types)}$$

$$V ::= \text{Int} \mid \text{List } V \mid \text{Tuple } VV \qquad \textit{(value types)}$$

It may appropriate to move this to the next section We assume that the goal program is normalized in the following sense. If the goal type specified by the user is a universal type, then we substitute each universally quantified type variable with a fresh fixed type variable X. If the goal type is a function type, then we abstract the type as much as possible and add input variables to the context Γ. For example, if the user specifies the goal program as

```
length :: ∀X. List X -> Int
length [1,2,3] == 3
length [2,2,2] == 3
length [] == 0
length [5] == 1
```

then we start our search from the partial program $\Gamma \cup \{xs : \texttt{List } X\} \vdash ?x :: \texttt{Int}$ where Γ already contains all bindings from library components to their types. Note that it must be possible to instantiate the type variable X with every possible type, therefore we do not have the right to prefer one instantiation over another and must treat X as an uninterpreted type.

TODO: Add the rules of the typing judgement The typing judgement is standard. To define the search graph we are going to explore in order to find the goal program, we also need the *derive* judgement, which says between which nodes of the graph (programs of the form $\Gamma, \Delta \vdash t :: T$) there is an

edge. To express the rules in a more compact form, we introduce *evaluation contexts*. An context is an expression with exactly one syntactic hole [] in which we can plug in any term. For example, if we have the context $\mathcal{E}$ we can place the term t into its hole and denote this new term by $\mathcal{E}[t]$.

TODO: Make a figure of all rules We can turn an active hole into an inactive hole if the active hole has a value type. do we need a premise? Wouldn't it be enough to write ?x :: V below the line?

$$\frac{\Gamma, \Delta \vdash ?x :: V}{\Gamma, \Delta \vdash ?x :: T \Rightarrow \Gamma, \Delta \setminus \{?x : T\} \cup \{?\underline{v} : V\} \vdash ?\underline{v} :: V} \text{ D-VarVal}$$

An active hole $?x$ can be turned into a library component or an input variable x from the context Γ. The procedure fresh(T) transforms universally quantified type variables into fresh type variables $?X$ not used in Δ. The notation $\sigma(\Delta)$ denotes the application of the substitution σ to all types contained in the context Δ.

$$\frac{x : T_x \in \Gamma \qquad \sigma \text{ unifies } T \text{ with fresh}(T_x)}{\Gamma, \Delta \vdash ?x :: T \Rightarrow \Gamma, \sigma(\Delta \setminus \{?x : T\}) \vdash x :: \sigma(T)} \text{ D-VarLib}$$

An active hole can also be turned into a function application of two new active holes.

$$\frac{?X \text{ is a fresh type variable}}{\Gamma, \Delta \vdash ?x :: T \Rightarrow \Gamma, \Delta \setminus \{?x : T\} \cup \{?x_1 :?X \to T, ?x_2 :?X\} \vdash ?x_1\ ?x_2 :: T} \text{ D-VarApp}$$

In all other cases we just choose an active hole and expand it according to the three rules above.

$$\frac{\Gamma, \Delta \vdash ?x :: T_1 \Rightarrow \Gamma, \Delta' \vdash t_1' :: T_1'}{\Gamma, \Delta \vdash t[?x] :: T \Rightarrow \Gamma, \Delta' \vdash t[t_1] :: T[T_1 / T_1']} \text{ D-App}$$

7.2 Baseline Algorithm

The breadth (or best) first search of the search graph, nothing fancy.

The baseline algorithm is a simple best first search implemented using a priority queue. One possibility is to order the enqueued elements according to the number of library components and active holes.

The root node has the form $\Gamma, \{?x : T\} \vdash ?x :: T$ where T is neither functional nor universal type and Γ contains the input variables along with the library components. The successors of a node are the nodes reachable in one step of the derive judgement. That is, $\Gamma, \Delta' \vdash t' :: T'$ is a successor of $\Gamma, \Delta \vdash t :: T$ if it holds $\Gamma, \Delta \vdash t :: T \Rightarrow \Gamma, \Delta' \vdash t' :: T'$.

We don't need to explore nodes that are equivalent up to alpha conversion to already visited nodes.

A term is considered *closed* if it does not contain active holes. Closed terms are tested on the input-output examples. Programs with inactive holes are symbolically executed on the input-output examples and, if possible, the concrete value of the inactive hole is determined. Otherwise we try to solve it by brute force with a small timeout. Does it make sense in terms of performance? Usually the values we need are really simple like [], 0, 1. Wouldn't it be more efficient to try some simple values first?

The input-output examples are given as a vector of inputs I and the vector of corresponding expected outputs O.

TODO: Typeset in a nicer way, for example with a vertical line instead of all those 'end' and nicer keyword formatting.

```
BFS(root, I, O)
        queue ← {root}
        visited ← {}
        while(timeout not reached)
                current ← queue.dequeue
                if (current.closed)
                        if (current.test(I) == O)
                                return current
                        end
                else
                        for (s in current.successors)
                                if (!visited.alphacontains(s))
                                        queue ← queue.push(s)
                                end
                        end
                end
        end
end
```

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.