ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Top-Down Inductive Synthesis with 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 **??**

- What is program synthesis
- Problem too general, needs to be restricted. In Chapter 2 we will see how other restrict the search space. Restrict to components.
- Motivate why it is still interesting to have such a synthesiser (thin interfaces and so on?)
- List of contributions
- Explain the structure of the thesis. (maybe merge with the contributions?)

for motivation you could write something about the extremely restricted list library in ocaml :)
Consider you are writing code in a functional programming language with a smaller choice of library functions than you are used to. For example (true story), if you are using OCaml and you are surprised that replicate is missing even in the more complete core library cite jane street core, you could spend a couple of minutes writing your recursive version of replicate. Or you can synthesize it with Tamandu in less than one second from other components. **TODO:** rewrite without 'you' and maybe don't mention ocaml, core and all that thing?.

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

A type-driven synthesis procedure

In this chapter we will formally define our type-directed top-down synthesis procedure. We will start with an intuitive description of the problem and move on to the formal definitions of the target programming language and the search space. Finally, we will define the synthesis procedure and present some enhancements.

2.1 The 'replicate' example

Recall the example from Chapter 1 where we wanted to synthesise `replicate` . As our synthesis procedure is type-driven and example-based, the user specifies a program by providing its type along with a few I/O-examples. Let us specify `replicate` as follows.

```
replicate :: ∀X. Int →X → List X
replicate 3 1 = [1,1,1]
replicate 2 [] = [[],[]]
```

We also need a library of components, from which we are going to compose our program. Let us assume we have the standard list combinators `map` and `foldr` with `enumTo`, the function that returns a list from 1 up to its argument, and `const`, that always returns its first argument, along with the list constructors `cons` and `[]` and the integer constructors `succ` and `0`. Even with so few library components, the search space is quite big.

The goal is to put together components from the library in order to get a list. More concretely, we fix `X` to be a fixed input type variable, we fix `n` to be an integer and `x` to be a fixed input variable of type `X`. Now the goal looks like

```
replicate n x = ?p
?p :: List X
```

, where `?p` is a *hole*, a new fresh variable whose type is known.

Which components can we use in order to get something of type `List X`? We cannot use `enumTo`, because it only produces a list of integers, but all other possibilities are open. We could either fold with an interesting function, or map some function, or even use `const` with a smart first argument. But the first and easiest thing that has the type `List X` is `[]`. That is, our first program looks as follows.

```
replicate n x = []
?p = [] :: List X
```

This is a *closed* program, that is, a program without holes that can be evaluated on the input-output examples. Alas, it does not satisfy any of them. Therefore we must try the other components. We have following possibilities to fill in the hole `?p`.

```
replicate n x = cons ?x ?xs
?p = cons ?x ?xs :: List X
?x :: X
?xs :: List X

replicate n x = map ?f ?xs
?p = map ?f ?xs :: List X
?f :: ?Y → X
?xs :: List ?Y
```

Here `?Y` is a fresh type variable that will be instantiated with something later. As of now we have no idea about the type of the argument of `?f`, we only know that it has to match the type of the elements of `?xs`.

```
replicate n x = foldr ?f ?init ?xs
?p = foldr ?f ?init ?xs :: List X
?f :: ?Y → List X → List X
?init :: List X
?xs :: List ?Y

replicate n x = const ?xs ?s
?p = const ?xs ?s :: List X
?xs :: List X
?s :: ?Y
```

Now we take the most promising program and try to fill in one of its *holes* (the fresh variables starting with `?`). Let us decide that the first one is the most promising. We now have two holes to fill in, a function that can take something and return an `X` and a list of something. Note that it has to be possible for all possible instantiations of `X` and note that we have only one value of type `X`, namely `x`, the second argument to `replicate`.

What are the possibilities to instantiate `?f`? Obviously, we cannot use `map` or `enumTo`, because they return lists. But all other possibilities are still open. We have to add following programs to our pool of possible solutions.

```
replicate n x = map (foldr ?g ?init) ?xs
?p = map ?f ?xs :: List X
?f = foldr ?g ?init :: List ?Z → X
?g :: ?Z → X → X
?init :: X
?xs :: List (List ?Z)
```

Note that we instantiated `?Y` with `List ?Z`, because `foldr` takes a list as its last argument.

```
replicate n x = map (const ?x) ?xs
?f = const ?x :: ?Y → X
?x :: X
?xs :: List ?Y
```

As you noticed, we maintain a frontier of programs with holes and expand one of the holes of the most promising program. Let us do it again. From the 6 programs generated so far we choose the most promising one. Let us decide that it is the last one. It has two holes to fill in. For the first hole, `?x`, we have only one possibility. As this hole must be of type `X`, the only thing we can take is the second argument of `replicate`, `x`.

```
replicate n x = map (const x) ?xs
?p = map ?f ?xs :: List X
?f = const ?x :: ?Y → X
?x = x :: X
?xs :: List ?Y
```

Let us directly decide that this is the most promising program so far. Later, in Section 2.4, we will define cost functions of programs and define the most promising program to be the one with the smallest cost. For now we just choose the one that will lead us to the solution.

Like in the beginning, we have to generate a list. However, since this time the type of the elements is not fixed, we cannot rule out `enumTo`. Therefore we have a lot of possibilities, starting with `replicate n x = map (const x)[]`, where we instantiate `?xs` with `[]`, and ending with

```
replicate n x = map (const x) (enumTo ?n)
?p = map ?f ?xs :: List X
?f = const ?x :: ?Y → X
?x = x :: X
?xs = enumTo ?n :: List Int
?n :: Int
```

First, we are going to evaluate the closed program `replicate n x = map ( const x)[]`. This program does not satisfy any of the input-output examples too.

The next step is to expand one of the holes of the most promising program, let us decide that it is the last one. That is, we are looking for an integer. An integer can either be the constructor `0`, or the constructor `succ` applied to some integer hole, the first argument of `replicate`, `n`, or `const` with some clever first argument applied to something.

Notice that among the new programs there are two closed programs, `replicate n x = map (const x)(enumTo 0)` and `replicate n x = map (const x)(enumTo n)`. Evaluation shows that only the second one satisfies the I/O-examples.

2.1.1 Summary

This example shows some important concepts that are defined formally in the next sections of this chapter.

Hole unknown part of a program that can be instantiated with some other programs. Only its type is known.

Closed program a program without holes that can be evaluated on the input-output examples. The terms and the types of our calculus are formally defined in Section 2.2.1.

Type-aware expansion of holes we expand holes based on their type. In Section 2.3.1 we can find the rules according to which a program is expanded.

Best first search a frontier of programs with holes is maintained, one hole of the most promising is expanded in every iteration. The best first search algorithm is defined in Section 2.3.2 and a notion of most promising program is presented in Section 2.4.

Superfluous program a program that is equivalent to a simpler program. For example, a human programmer would not instantiate a hole with `const ?x ?y`, because he could have written just `?x` instead, which is always a shorter and preferable program. Section 2.5 shows one way to rule out superfluous programs.

2.2 Calculus

In this section we will look at three different calculi.

1. System F
2. An extension of System F with holes, input variables, library components, parametric types and recursive terms and types
3. A subset of the second calculus, featuring only application of components, holes or input variables

We provide the first calculus only for the sake of completeness, as the other two calculi build upon it. The notation and the exposition follow the excellent book on type systems of Benjamin Pierce [6]. We refer to the book for a thorough introduction to System F and to type systems in general.

The third calculus is the target language of our synthesiser. However, we still need the more powerful second calculus to define the library components and evaluate them on the input-output examples.

2.2.1 Terms and Types

System F, also known as the polymorphic lambda calculus, is a calculus that, additionally to term abstraction and term application, features two new kinds of terms: Type abstraction Λ`X. t` and type application `t [T]`. This allows to express polymorphic functions. Polymorphic functions, defined as type abstractions, have a special type: The *universal* type $\forall$`X. T`. For a more detailed introduction to System F we refer to [6]. The syntax is summarized below.

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

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

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

We extend System F with holes $?x$, input variables i as well as named library components c and named types C that can take parameters $C\ [T_1, \ldots, T_k]$. The number of type parameters supported by a named type is denoted as K in its definition. The use of the names enables recursion in the definition of library components and types. Terms that do not contain holes are called *closed*. The syntax of our calculus is summarised below.

Note that we have three additional contexts. Ξ binds term holes to their types and type holes. Φ is the library of input variables. It contains one concrete instantiation of the input variables. It binds a definition and a type signature to each input term variable and a definition to each input type variable. Δ is the library of components. Each named term is bound to its definition and to its type signature and each named type is bound to ist definition and to the number of parameters it takes.

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

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

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

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

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

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

Evaluation and typing rules for this calculus can be found in the respective subsections.

The target language of our synthesiser is a subset of the previous calculus. We are only interested in term and type application of library components, input variables and holes. Therefore, the syntax is restricted as follows.

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

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

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

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

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

Since this is a proper subset of the second calculus presented in this section, we do not need separate typing and evaluation rules.

The synthesiser synthesises programs over the target language. A program is defined as a quadriplet $\{\Xi, \Phi, \Delta \vdash t :: T\}$, where t is a term of the target language. A program is called *closed*, if Ξ is empty and t and T do not contain holes.

2.2.2 Encodings

Note that in the definition of types we do not see familiar types such as booleans, integers or lists. All these types can be encoded in System F using either the Church's or the Scott's encoding [1]. We opted for the Scott's encoding because it is 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, as they unwrap the constructor only once.

```
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
```

2.2.3 Evaluation semantics

In this section we present the evaluation semantics of our extended System F, that is the second calculus introduced in Section 2.2.1. The evaluation semantics is a standard eager evaluation and we refer to the excellent book of Benjamin Pierce about type systems [6] for an introduction to the evaluation semantics of System F and evaluation rules in general.

The evaluation judgement $\Phi, \Delta \vdash t \longrightarrow t'$ means that the term t evaluates in one step to the term t' under the free variable bindings library Φ, that contains concrete instantiations for the input variables, and the component library Δ , that contains the definitions of the library components. Before listing the evaluation rules, let us define *value* v to be a term to which no evaluation rule apply.

$$\frac{c = t : T \in \Delta}{\Phi, \Delta \vdash c \longrightarrow t}\ \text{E-Lib}$$

$$\frac{i = t : T \in \Phi}{\Phi, \Delta \vdash i \longrightarrow t}\ \text{E-Inp}$$

$$\frac{\Phi, \Delta \vdash t_1 \longrightarrow t_1'}{\Phi, \Delta \vdash t_1\ t_2 \longrightarrow t_1'\ t_2}\ \text{E-App1}$$

$$\frac{\Phi, \Delta \vdash t_2 \longrightarrow t_2'}{\Phi, \Delta \vdash v_1\ t_2 \longrightarrow v_1\ t_2'}\ \text{E-App2}$$

$$\frac{}{\Phi, \Delta \vdash (\lambda x : T_{11}.\ t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12}}\ \text{E-AppAbs}$$

$$\frac{}{\Phi, \Delta \vdash (\Lambda X.\ t_2)\ [T_2] \longrightarrow [X \mapsto T_2]t_2}\ \text{E-AppAbs}$$

Rules E-Lib and E-Inp load the definitions of library components or input variables from the respective library. E-App1 and E-App2 evaluate the left hand side, respectively the right hand side, of an application. E-AppAbs and E-AppAbs get rid of an abstraction and substitute the argument into the body.
Note that E-App2 applies only if the left hand side of the application cannot be evaluated further and that E-AppAbs applies only when the argument of the lambda abstraction is a value, determining the order of evaluation.

2.2.4 Type checking

In this sections we will present the typing rules of our extended System F, that is the second calculus presented in Section 2.2.1. The typing judgement $\Gamma, \Xi, \Phi, \Delta \vdash t : T$ means the term t has type T in the contexts Γ and Ξ, binding respectively variables and holes, and Φ and Δ, containing signatures and definitions of respectively input variables and library components. The typing judgement is similar to the typing judgement of System F presented in the book about type systems of Benjamin Pierce [6] and we refer to the book for more details.

$$\frac{x : T \in \Gamma}{\Gamma, \Xi, \Phi, \Delta \vdash x : T}\ \text{T-Var}$$

$$\frac{?x : T \in \Xi}{\Gamma, \Xi, \Phi, \Delta \vdash ?x : T}\ \text{T-Hol}$$

$$\frac{i = t : T \in \Phi}{\Gamma, \Xi, \Phi, \Delta \vdash i : T}\ \text{T-Inp}$$

$$\frac{c = t : T \in \Delta}{\Gamma, \Xi, \Phi, \Delta \vdash c : T}\ \text{T-Lib}$$

$$\frac{\Gamma \cup \{x : T_1\}, \Xi, \Phi, \Delta \vdash t_2 : T_2}{\Gamma, \Xi, \Phi, \Delta \vdash \lambda x : T_1.\ t_2 : T_1 \to T_2}\ \text{T-Abs}$$

$$\frac{\Gamma, \Xi, \Phi, \Delta \vdash t_1 : T_1 \to T_2 \qquad \Gamma, \Xi, \Phi, \Delta \vdash t_2 : T_1}{\Gamma, \Xi, \Phi, \Delta \vdash t_1\ t_2 : T_2}\ \text{T-App}$$

$$\frac{\Gamma \cup \{X\}, \Xi, \Phi, \Delta \vdash t_2 : T_2}{\Gamma, \Xi, \Phi, \Delta \vdash \Lambda X.\ t_2 : \forall X.\ T_2}\ \text{T-Abs}$$

$$\frac{\Gamma, \Xi, \Phi, \Delta \vdash t_1 : \forall X.\ T_{12}}{\Gamma, \Xi, \Phi, \Delta \vdash t_1\ [T_2] : [X \mapsto T_2]T_{12}}\ \text{T-App}$$

The rules T-Var, T-Hol, T-Inp and T-Lib load the signature of variables, holes, input variables or library components from the respective context. T-Abs types a lambda abstraction as an arrow type from the type of its variable to the type of its body. T-Abs gives a type abstraction as a universal type in accordance to the type of its body. An application typechecks only if the left hand side has an arrow type and the type of the argument is equal to the type of the argument of the left hand side. Analogously, a type application typechecks only if the left hand side has a universal type.

2.2.5 Type unification

TODO: Write this

Need it to see which component can be used to expand a given hole.

Since unification on System F types is undecidable [4], we decided to do something simpler. Explain what. Boom, fertig, party.

The unification algorithm is based on the unification algorithm for typed lambda calculus from the book of Pierce **TODO:** cite the book properly!!! and slightly modified to fit our needs.

How did you do type unification? Unification of universal types is not implemented. If you need to unify to universal types, you should transform all bound variables into holes and remove the universal quantifier.

A set of constraints is a set of types that should be equal under a substitution. The unification algorithm is supposed to output a substitution σ so that $\sigma(S) = \sigma(T)$ for every constraint $S = T$ in $\mathcal{C}$.

Input: Set of constraints $\mathcal{C} = \{T_{11} = T_{12}, T_{21} = T_{22}, \ldots\}$
Output: Substitution σ so that $\sigma(T_{i1}) = \sigma(T_{i2})$ for every constraint $T_{i1} = T_{i2}$ in $\mathcal{C}$
Function *unify*($\mathcal{C}$) **is**
  if $\mathcal{C} = \emptyset$ **then** []
  else
    let $\{T_1 = T_2\} \cup \mathcal{C}' = \mathcal{C}$ in
    if $T_1 = T_2$ **then**
      unify($\mathcal{C}'$)
    else if $T_1 = ?X$ *and* $?X$ *does not occur in* T_2 **then**
      unify($[X \mapsto T_2]\mathcal{C}'$) $\circ$ $[X \mapsto T_2]$
    else if $T_2 = ?X$ *and* $?X$ *does not occur in* T_1 **then**
      unify($[X \mapsto T_1]\mathcal{C}'$) $\circ$ $[X \mapsto T_1]$
    else if $T_1 = T_{11} \to T_{12}$ *and* $T_2 = T_{21} \to T_{22}$ **then**
      unify($\mathcal{C}' \cup \{T_{11} = T_{21}, T_{12} = T_{22}\}$)
    else if $T_1 = C\ [T_{11}, T_{12}, \ldots, T_{1k}]$ *and* $T_2 = C\ [T_{21}, T_{22}, \ldots, T_{2k}]$ **then**
      unify($\mathcal{C}' \cup \{T_{11} = T_{21}, T_{12} = T_{22}, \ldots, T_{1k} = T_{2k}\}$)
    else
      fail
    end
  end
end

Algorithm 1: Type unification

2.3 Search

TODO: Write this section
how do we explore the search space (best-first search).

2.3.1 Search space

Use only third system presented in Section System F.

Formal problem definition. Given a library Δ, a goal type T and a list of I/O-examples $[(\Phi_1, o_1), \ldots, (\Phi_N, o_N)]$, find a closed term t so that $\emptyset, \emptyset, \Phi_1, \Delta \vdash t : T$ (that is, t has the goal type under an empty variable binding context and an empty hole binding context) and t satisfies all I/O-examples, that is $\Phi_n, \Delta \vdash t \longrightarrow^* \vdash t'$ and $\Phi_n, \Delta \vdash o_n \longrightarrow^* \vdash t'$ for all $n = 1, \ldots, N$.

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.

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

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

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

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

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

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

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

2.3.2 Best-first search

TODO: Write this section

Write also about stack vs queue of open holes Say that it's standard, we just review it.

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. Different approaches to define the `compare` function are discussed in the next section. Concerning the order of expansion of the holes, we tried two strategies: the first one was maintaining a stack of holes (Ξ implemented as a stack), which leads to the expansion of the deepest hole first, the second one was maintaining a queue of holes (Ξ implemented as a queue), which leads to the expansion of the oldest hole first.

Input: goal type T, library components Δ, list of input-output examples $[(\Phi_1, o_1), \ldots, (\Phi_N, o_N)]$
Output: closed program $\{\Xi, \Phi_1, \Delta \vdash t :: T\}$ that satisfies all I/O-examples
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
return *PriorityQueue.top queue*

Algorithm 2: Best first search

2.4 Cost functions

TODO: write this section

The compare function in the best first search algorithm is `cost` p_1 `- cost` p_2. There are different possibilities to implement this cost function. We will present four alternatives.

number of nodes The first cost function is based only on the number of nodes of the term. Longer and more complicated terms are disadvantaged.

number of nodes and types The second cost function also adds a factor based on the types appearing in the term, thus penalizing terms with type application and very complicated types.

no same component In the third function we additionally penalize terms using the same component more than once.

length of the string The simplest and most imprecise method to take both the number of nodes and the complexity of the types appearing in the term

Input: term t
Output: weighted number of nodes in t
$\mathit{nof\text{-}nodes}(c) = 1$
$\mathit{nof\text{-}nodes}(?x) = 2$
$\mathit{nof\text{-}nodes}(i) = 0$
$\mathit{nof\text{-}nodes}(x) = 1$
$\mathit{nof\text{-}nodes}(\lambda x : T.\ t) = 1 + \mathit{nof\text{-}nodes}(t)$
$\mathit{nof\text{-}nodes}(t_1\ t_2) = 1 + \mathit{nof\text{-}nodes}(t_1) + \mathit{nof\text{-}nodes}(t_2)$
$\mathit{nof\text{-}nodes}(\Lambda X.\ t) = 1 + \mathit{nof\text{-}nodes}(t)$
$\mathit{nof\text{-}nodes}(t\ [T]) = 1 + \mathit{nof\text{-}nodes}(t)$

Algorithm 3: Cost function based on the number of nodes

Input: term t
Output: sum of weighted number of nodes in term t and weighted number of nodes in the types appearing in t
$\mathit{nof\text{-}nodes\text{-}type}(X) = 1$
$\mathit{nof\text{-}nodes\text{-}type}(?X) = 0$
$\mathit{nof\text{-}nodes\text{-}type}(I) = 0$
$\mathit{nof\text{-}nodes\text{-}type}(C\ [T_1, \ldots, T_k]) = 0$
$\mathit{nof\text{-}nodes\text{-}type}(T_1 \to T_2) = 3 + \mathit{nof\text{-}nodes\text{-}type}(T_1) + \mathit{nof\text{-}nodes\text{-}type}(T_2)$
$\mathit{nof\text{-}nodes\text{-}type}(\forall X.\ T) = 1 + \mathit{nof\text{-}nodes\text{-}type}(T)$

$\mathit{nof\text{-}nodes\text{-}term}(c) = 1$
$\mathit{nof\text{-}nodes\text{-}term}(?x) = 2$
$\mathit{nof\text{-}nodes\text{-}term}(i) = 0$
$\mathit{nof\text{-}nodes\text{-}term}(x) = 1$
$\mathit{nof\text{-}nodes\text{-}term}(\lambda x : T.\ t) = 1 + \mathit{nof\text{-}nodes\text{-}term}(t) + \mathit{nof\text{-}nodes\text{-}type}(T)$
$\mathit{nof\text{-}nodes\text{-}term}(t_1\ t_2) = 1 + \mathit{nof\text{-}nodes\text{-}term}(t_1) + \mathit{nof\text{-}nodes\text{-}term}(t_2)$
$\mathit{nof\text{-}nodes\text{-}term}(\Lambda X.\ t) = 1 + \mathit{nof\text{-}nodes\text{-}term}(t)$
$\mathit{nof\text{-}nodes\text{-}term}(t\ [T]) = 1 + \mathit{nof\text{-}nodes\text{-}term}(t) + \mathit{nof\text{-}nodes\text{-}type}(T)$

$\mathit{nof\text{-}nodes\text{-}and\text{-}types}(t) = \mathit{nof\text{-}nodes\text{-}term}(t)$

Algorithm 4: Cost function based on the number of nodes and types

Input: term t
Output: sum of the weighted number of nodes in term t, the weighted number of nodes in the types appearing in t and the weighted number of library components appearing more than once in t.

$$\textit{nof-nodes-type}(X) = 10$$
$$\textit{nof-nodes-type}(?X) = 3$$
$$\textit{nof-nodes-type}(I) = 0$$
$$\textit{nof-nodes-type}(C\ [T_1, \ldots, T_k]) = 4 + \textit{nof-nodes-type}(T_1) + \ldots + \textit{nof-nodes-type}(T_k)$$
$$\textit{nof-nodes-type}(T_1 \to T_2) = 5 + \textit{nof-nodes-type}(T_1) + \textit{nof-nodes-type}(T_2)$$
$$\textit{nof-nodes-type}(\forall X.\ T) = 10 + \textit{nof-nodes-type}(T)$$

$$\textit{nof-nodes-term}(c) = 3$$
$$\textit{nof-nodes-term}(?x) = 2$$
$$\textit{nof-nodes-term}(i) = 0$$
$$\textit{nof-nodes-term}(x) = 10$$
$$\textit{nof-nodes-term}(\lambda x : T.\ t) = 10 + \textit{nof-nodes-term}(t) + \textit{nof-nodes-type}(T)$$
$$\textit{nof-nodes-term}(t_1\ t_2) = 6 + \textit{nof-nodes-term}(t_1) + \textit{nof-nodes-term}(t_2)$$
$$\textit{nof-nodes-term}(\Lambda X.\ t) = 10 + \textit{nof-nodes-term}(t)$$
$$\textit{nof-nodes-term}(t\ [T]) = 5 + \textit{nof-nodes-term}(t) + \textit{nof-nodes-type}(T)$$

$$\textit{count}(t) = \sum_{c_i \text{ appears in } t} (\text{occurrences of } c_1 \text{ in } t) - 1$$

$$\textit{no-same-component}(t) = \textit{nof-nodes-term}(t) + 3\ \textit{count}(t)$$

Algorithm 5: Cost function based on the number of nodes and types penalizing the use of a library component more than once

is taking the length of the string of that term.

2.5 Black list

TODO: Write this section
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.

One could use the synthesis algorithm presented in Section 2.3.2 to automatically synthesise black lists. For example, one could synthesise many programs corresponding to the identity function or to the empty list or to any other term, and add all those programs but one to the black list.

Input: goal type T, library components Δ, list of input-output examples $[(\Phi_1, o_1), \ldots, (\Phi_N, o_N)]$, black list $[b_1, \ldots, b_M]$
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**
  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 6: Best first search with black list

2.6 Templates

TODO: Write this section
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 components. The search space is thus similar to the search space described in 2.3.1, 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.

We additionally restrict the space by requiring a template to have no more than M higher-order components and no more than P closed holes.

2.6.1 Successor rules

The successor rules are very similar to the ones defined in 2.3.1, 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. All the rules are modified

to take into account the restriction on the number of components and the number of closed holes. In order to do this, we need to pass along m, the number of higher-order components in the term whose subterms we are traversing.

So we can *close* a hole.

$$\frac{\begin{array}{c}|\Xi| \leq P \text{ and } m \leq M \\ T \text{ is a type a first-order component can have}\end{array}}{\Xi, \Phi, \Delta, m \vdash ?x :: T \Rightarrow \Xi, \Phi, \Delta, m \vdash \underline{?x} :: T} \text{ G-VarClose}$$

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

$$\frac{\begin{array}{c}|\Xi| \leq P \text{ and } m < M \\ c = t_c : T_c \in \Delta \\ \sigma \text{ unifies } T \text{ with fresh}(T_c)\end{array}}{\Xi, \Phi, \Delta, m \vdash ?x :: T \Rightarrow \sigma(\Xi \setminus \{?x : T\}), \Phi, \Delta, m + 1 \vdash c :: \sigma(T)} \text{ G-VarLib}$$

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

$$\frac{\begin{array}{c}|\Xi| < P \text{ and } m \leq M \\ ?X \text{ is a fresh type hole, } ?x_1 \text{ and } ?x_2 \text{ are fresh term holes} \\ \Xi' = \Xi \setminus \{?x : T\} \cup \{?x_1 : ?X \to T, ?x_2 : ?X, ?X\}\end{array}}{\Xi, \Phi, \Delta \vdash ?x :: T \Rightarrow \Xi', \Phi, \Delta \vdash ?x_1\ ?x_2 :: T} \text{ G-VarApp}$$

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

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

Chapter 3

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
- Table of implemented components

Chapter 4

Related Work

Here discuss only synthesis of functional programs. Input-output examples are an intuitive and easy way to specify programs. Types are good to prune the search space, but simple types are not enough to specify a program. So the research went in two directions: on one side, there was the need to do something to the I/O-examples to make the search more efficient, on the other side more complex and expressive types that can act as a specification, for example the *refinement types* from [7]. Lately, there was the idea that one can automatically generate those ty[pes from I/O-examples, to have both the advantages of an intuitive and easy specification and the research done on type systems, liquid types and something like that.

Let's start. My 5 papers, organize some information about them.

- Synquid (Nadia) put the replicate example if you can (should not be that difficult). Not that "user friendly". Lambda, conditionals, recursion, components. Synthesis starting from a partial program possible. Fast, good for sorting. It is possible to download the tool (put link), try it online and there is even an emacs-mode for it.

    1. What is the specification? The refinement type of the desired program, that is a type decorated with logical predicates. Can bring the replicate example.

    2. What is the target language? Haskell, I think. I saw lambda abstractions and recursion. Can generate and use higher-order components. You can specify own datatypes and own components. Pattern matching, structural recursion, ability to use and generate polymorphic higher-order functions, reasoning about universal and recursive properties of data structures.
    The Synquid language has lambda expressions, pattern matching, conditional and fixpoint.

3. What can it do well and fast? How fast? Most advanced benchmark: generate various sorting algorithms for data structures with complex invariants (search trees, heaps, balanced trees). This is out of scope of my synthesis procedure. For the ”easy” benchmarks (what I have) everything under 0.4 s. For more complicated stuff like sorting and tree insertion under 5 seconds. For most complicated stuff (red-black trees) under 20 s.

4. What is the difficulty? What can they not generate (or take a long time to generate)? How much time do they need? It is not always easy for an inexperienced user to specify a program and to find the right 'measure' function for a datatype.
   They need up to 20 s to generate balance over red-black trees. They usually don't give more than 5 components and 6 measure functions over the needed data types.
   The specification is quite big compared to the synthesised program, half of the programs the specification is one third of the generated program or more.

5. What do they do? In [7] 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.

- Lambda square (Feser). How would you specify replicate? Give an idea of how it is generated. User friendly, as only I/O-examples need to be specified, type is reconstructed automatically.

  1. What is the specification? I/O-examples

  2. What is the target language? Their synthesis algorithm targets a functional programming language that permits higher-order functions, combinators like map, fold and filter, pattern-matching, recursion, and a flexible set of primitive operators and constants. Actually, lambda calculus with algebraid data types and recursion.

  3. What can it do well and fast? How fast? Generating programs over nested structures like trees of lists, lists of lists and lists of trees. They generate the half of the benchmarks in under 0.43 s.

4. What is the difficulty? What can they not generate (or take a long time)? How much time do they need? Only 7 higher-order combinators and not much more first-order components.
   They use relatively many examples (between 3 and 12, mostly 4 to 6).
   They need up to 320 s to generate `droplast`. Other benchmarks that take more than 100 s to synthesise are removing duplicates from a list, inserting a tree under each leaf of another tree and dropping the smallest number in a list of lists.

5. What do they do? Inductive generalization, deduction, enumerative search. First, generate *hypotheses* (that is, programs with free variables like λ`x. map ?f x` where `?f` is a placeholder for an unknown program that needs to be generated) in a type-aware manner (the type is inferred from the I/O-examples). Then deduction based on the semantics of the higher-order combinators is used either to quickly refute a hypothesis or infer new I/O-examples to guide the synthesis of missing functions. Best-first enumerative search is used to enumerate candidate programs to fill in the missing parts of hypotheses.
   The tool proposed in [3] 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 tool is able to synthesise all benchmark programs in under 7 minutes.

- Escher (Kincaid). User friendly, as untyped and only I/O-examples need to be specifies. Powerful (recursion, special if-then-else, components) Goal graph.

  1. What is the specification? I/O-examples. Oracle simulating interaction with the user.

  2. What is the target language? Recursion, special treatment for con-

ditionals, components. That's really all they have: either apply a component to its arguments (including component `self` referring to the program being generated for recursion), a constant, an input variable or a conditional. Programming language is untyped. Components may be higher-order.

3. What can they do well and fast? How fast? Synthesise recursive programs (tail recursive, divide-and-conque, mutually recursive). In the evaluation, they give 23 components to all benchmarks, none of them is higher-order. Can generate all benchmarks in under 11 s, most of them in under 1 s.

4. What is the difficulty? What can they not generate (or take a long time)? How much time do they need? The user must give a *closed* example set, that is for each example, there must be another example that can be used for a recursive call.

5. What do they do? Forward search: inductive enumeration, add a new component to an already synthesised program. Conditional inference: use the goal graph to see if you can join to synthesised programs by a conditional statement. A heuristic guides the alternation between forward search and conditional inference. Programs are associated with value vectors. Search space is also pruned based on *observational equivalence*, that is programs with equivalent value vectors are treated as equivalent programs.
   Can be formalised as a non-deterministic transition system over configurations (triples consisting of a set of synthesised programs, a goal graph and a list of input-output examples) with six transition rules: an initial rule to start the search, a terminate rule to terminate the search and four synthesis rules (one for forward search, two for conditional split and one to ask for new input-output examples to evaluate a recursive call). The recursive calls are answered by the oracle.
   Rule scheduling is added to turn this into a practical system (different heuristics).
   In [2] 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.

- Myth (Osera). Like mine, requires type and I/O-examples. Refinement tree.

  1. What is the specification? A type signature, the components and a list of input-output examples.

  2. What is the target language? pattern matching, recursion, higher-order functions in typed programming languages. Can synthesise higher-order functions, programs using higher-order functions and work with large algebraic datatypes. ML-like language with algebraic data types, match, top-level function definitions and explicitly recursive functions.

  3. What can they do well and fast? How fast? Recursive programs with pattern matching. It's very fast, many programs are synthesised in around 0.1 s. It can also generate larger programs (75 AST nodes) in reasonable time (3 s for calculating the set of free variables in an untyped lambda-calculus).

  4. What is the difficulty? What can they not generate (or take a long time)? How much time do they need? To generate recursive functions, they also need a closed set of examples, so that a recursive call to the function being synthesised can be answered by an input-output example. They also require relatively many examples. They use a relatively large context, but they do not say how big it is. Lacks support richer types like products and polymorphic types.

  5. What do they do? The tool in [5] 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 (push them down in the refinement tree) and guess a term of the right type that matches the examples (for one of the nodes of the refinement tree).

Chapter 5

Evaluation

The main goal of this chapter is to give some insights about the factors that affect performance and compare different variants of the synthesis procedure described in Chapter 2. The chapter also puts our synthesis procedure in relation with the related work discussed in Chapter 4.

5.1 Experimental set up

This section presents the set up of the two experiments we are going to discuss in the rest of the chapter.

The goal of the first experiment is to assess the quality and the performance of the synthesiser on standard benchmarks. The detailed set up is described in Section 5.1.1 and the results are discussed in Section 5.2.

In the second experiment the synthesiser is used to automatically generate a *black list* that can successively be used to prune the search space. We refer back to Section 2.5 for a description of pruning based on black lists. Section 5.1.2 describes how we used the synthesis procedure to generate a black list and Section 5.4 reviews the quality of the generated black list.

5.1.1 Evaluation on benchmarks

TODO: Make and insert mentioned figures: manual black list

We evaluated nine variants of our synthesis procedure, crossing the three exploration strategies with three of the cost functions described in Chapter 2. The three exploration strategies we evaluated are the following.

PLAIN implements the basic synthesis procedure based on best first search described in Section 2.3.2.

Blacklist implements the pruning of the search space based on a manually compiled black list provided in Table **??**. We refer to Section 2.5 for more details.

Template implements the double best first search introduced in Section 2.6. As you probably recall, the procedure first looks for a *template* featuring at most `nof_comp` higher-order components and at most `nof_hol` holes and as soon as such a template is found the procedure falls back on the Plain variant up to a certain depth using only the first-order components.

For each exploration strategy, we instantiated the cost function with three of the cost functions described in Section 2.4, that is with *nof-nodes*, *nof-nodes-simple-type* and *no-same-component*. We refer back to the corresponding section for more details.

We exercised the nine different variants of our synthesis procedure on a benchmark of 23 programs over lists, mostly taken from related work or standard functional programming assignments. Table **??** summarises the running times. The first three columns summarise the runtimes of the nine variants of out synthesis procedure when the synthesiser is given 36 to 37 components. Columns four to six contain, for each variant, the slowdown with respect to the minimum running time for the respective benchmark. The last three columns show the speedup we obtain if we leave only 18 to 19 components in the library. Table 5.2 lists the benchmarks along with the size of the solution generated by each of the nine variants, expressed in number of nodes.

All experiments were run on an Intel quad core 3.2 GHz with 16 GB RAM. Since the code is sequential, the performance could not benefit from the number of cores. The performance numbers are averages from 1 to 3 different executions all sharing the same specification, that is the goal type, the given examples and the set of components do not change between different executions.

In the first three columns of Table 5.1 all benchmarks except for `nth` share the same set of components. The differences in the number of components come from the fact that we took out the benchmark to synthesise, if it was one of the given components. In the last three columns of Table 5.1, in order to meet the needs of all benchmarks, we used four different sets of 19 components.

Programs are enumerated only up to a timeout based on the number of programs that have been analysed so far. For the exploration strategies Plain and BlackList the execution had been stopped after examining 2500000 programs (with or without holes). The exploration strategy Template was restricted to generate templates with at most 2 higher-order components and

at most 5 holes, the depth of the first-order search was limited to 10 calls to the PLAIN procedure. For the cost function *nof-nodes* this corresponds to circa 4 min.

Name	M.	Time			*Vs.* 37-group			*Vs.* 19-self		
		NODES	TYPES	NODUP	NODES	TYPES	NODUP	NODES	TYPES	NODUP
`append`	P	0.32	0.10	0.07	4.60	1.39	1.00	5.34	3.29	2.53
	B	0.08	0.07	0.07	1.27	1.00	1.00	2.55	2.64	2.13
	T	1.90	2.63	2.44	1.00	1.38	1.28	2.01	3.47	2.25
`concat`	P	0.19	0.04	0.24	4.32	1.00	5.51	3.08	1.50	2.29
	B	0.04	0.04	0.19	1.01	1.00	5.02	1.42	1.38	1.80
	T	1.86	2.38	2.30	1.00	1.28	1.24	2.27	3.28	2.07
`drop`	P	0.02	0.02	0.04	1.19	1.00	2.33	0.74	0.96	2.30
	B	0.04	0.05	0.05	1.00	1.40	1.20	1.19	1.83	2.13
	T	0.87	1.03	0.64	1.37	1.62	1.00	8.35	6.25	3.60
`droplast`	P	0.09	0.06	0.09	1.43	1.00	1.40	1.97	2.73	2.30
	B	0.06	0.06	0.08	1.08	1.00	1.41	2.52	3.15	2.00
	T	0.76	1.40	–	1.00	1.85	–	2.69	2.66	3.02
`dropmax`	P	1.64	0.89	0.33	4.93	2.67	1.00	8.36	7.69	7.92
	B	0.72	0.60	0.38	1.90	1.58	1.00	8.51	8.62	9.04
	T	7.58	4.98	0.58	12.98	8.54	1.00	2.99	1.70	1.99
`enumFromTo`	P	–	–	–	–	–	–	11.02	25.39	10.40
	B	204.04	242.81	–	1.00	1.19	–	32.56	29.66	19.49
	T	–	–	–	–	–	–	5.54	4.30	27.50
`enumTo`	P	0.02	0.02	0.01	2.56	2.35	1.00	0.55	0.70	0.04
	B	0.07	0.08	0.03	2.91	3.22	1.00	3.69	3.32	0.10
	T	1.67	1.04	0.49	3.38	2.10	1.00	4.27	2.37	1.52
`factorial`	P	0.00	0.00	0.02	1.00	1.03	4.42	2.09	2.05	6.65
	B	0.00	0.00	0.01	1.00	1.01	2.03	1.88	1.89	3.02
	T	209.97	168.07	0.01	$\approx$ 19K	$\approx$ 16K	1.00	12.64	11.12	1.64
`isEven`	P	0.00	0.00	0.00	1.03	1.16	1.00	1.69	1.90	1.64
	B	0.00	0.00	0.00	1.01	1.00	1.01	1.54	1.53	1.55
	T	0.00	0.00	0.00	1.00	1.00	1.09	1.39	1.39	1.51
`isNil`	P	0.00	0.00	0.01	1.02	1.00	3.68	1.84	1.79	4.07
	B	0.00	0.00	0.01	1.01	1.00	3.82	1.75	1.67	3.91
	T	0.13	0.16	–	1.00	1.21	–	11.78	12.60	9.75
`last`	P	0.00	0.00	0.01	1.00	1.29	5.74	1.58	1.67	6.20
	B	0.00	0.00	0.00	1.00	1.02	1.63	1.24	1.48	1.43
	T	0.00	0.00	0.00	1.01	1.00	1.54	1.40	1.32	1.46
`length`	P	49.00	74.95	343.56	1.00	1.53	7.01	9.02	28.21	38.90
	B	4.93	28.44	148.75	1.00	5.76	30.15	14.71	20.08	37.12
	T	16.40	15.36	6.23	2.63	2.46	1.00	5.41	3.44	1.89
`mapAdd`	P	0.42	0.27	0.56	1.57	1.00	2.12	5.76	4.66	17.03
	B	0.28	0.25	0.70	1.13	1.00	2.78	3.27	2.83	16.80
	T	2.93	1.94	2.76	1.52	1.00	1.43	2.76	2.62	8.65
`mapDouble`	P	55.90	20.45	24.21	2.73	1.00	1.18	33.84	13.03	21.42

	B	14.12	13.12	17.60	1.08	1.00	1.34	13.31	18.72	21.81
	T	–	–	–	–	–	–	5.68	4.43	4.54
`maximum`	P	0.77	0.50	0.87	1.55	1.00	1.74	11.39	8.89	6.65
	B	0.32	0.24	0.61	1.31	1.00	2.54	4.32	3.70	4.52
	T	–	–	1.91	–	–	1.00	308.95	287.81	1.89
`member`	P	62.06	28.66	56.56	2.17	1.00	1.97	13.13	7.11	16.34
	B	26.95	22.67	50.65	1.19	1.00	2.23	7.22	6.48	16.41
	T	38.07	34.76	13.07	2.91	2.66	1.00	3.99	4.52	0.25
`multfirst`	P	0.18	0.08	0.13	2.34	1.00	1.72	0.22	0.24	0.46
	B	0.07	0.06	0.14	1.06	1.00	2.18	0.42	0.44	0.61
	T	13.35	1.95	0.70	19.02	2.78	1.00	2.66	2.43	2.18
`multlast`	P	7.79	1.32	1.19	6.56	1.12	1.00	0.80	0.40	0.67
	B	0.71	0.59	1.00	1.19	1.00	1.68	0.63	0.68	1.11
	T	201.81	72.08	184.20	2.80	1.00	2.56	5.05	3.34	3.68
`nth`	P	77.08	0.81	0.24	315.67	3.31	1.00	0.82	0.81	0.47
	B	0.35	0.34	0.20	1.73	1.70	1.00	0.49	0.74	0.49
	T	–	–	–	–	–	–	10.82	10.15	10.08
`replicate`	P	3.35	0.11	0.12	31.16	1.00	1.15	16.43	2.28	2.11
	B	0.09	0.08	0.14	1.08	1.00	1.63	1.48	1.74	2.05
	T	2.89	1.90	0.70	4.13	2.71	1.00	0.83	1.02	0.64
`reverse`	P	38.44	5.04	36.47	7.63	1.00	7.24	2.44	9.81	11.84
	B	1.71	0.99	17.43	1.73	1.00	17.68	4.89	3.88	11.12
	T	42.57	33.59	159.53	1.27	1.00	4.75	4.35	2.85	3.87
`stutter`	P	–	82.82	34.76	–	2.38	1.00	0.87	5.78	8.15
	B	31.91	15.23	24.59	2.10	1.00	1.61	4.84	3.46	10.18
	T	–	–	–	–	–	–	3.24	2.65	2.63
`sum`	P	0.69	0.37	0.76	1.88	1.00	2.08	13.73	11.28	11.00
	B	0.34	0.32	0.58	1.08	1.00	1.84	13.92	13.25	8.69
	T	1.91	2.11	30.20	1.00	1.11	15.85	2.76	2.98	31.26

Table 5.1: **TODO:** reformulate caption. Table of all benchmarks. The first column is the time in s of the various variants of the synthesis procedure with 37 or 36 components, The second column is the ratio between the running times of the 37 group and the minimum synthesis time for that benchmark using 36 or 37 components, the third column is the speedup we gain if we use only 19 or 18 components.

Name	Plain			Blacklist			Templates		
	Nodes	Types	NoDup	Nodes	Types	NoDup	Nodes	Types	NoDup
`append`	7	7	7	7	7	7	7	7	7
`concat`	7	7	7	7	7	7	7	7	7
`drop`	7	7	7	7	7	7	7	7	7
`droplast`	7	7	7	7	7	7	7	7	7
`dropmax`	9	9	9	9	9	9	9	9	9
`enumFromTo`	–	–	–	13	13	–	–	–	–
`enumTo`	7	7	7	7	7	9	7	7	7
`factorial`	5	5	5	5	5	5	13	13	5
`isEven`	3	3	3	3	3	3	3	3	3

isNil	5	5	5	5	5	5	5	5	5
last	5	5	5	5	5	5	5	5	5
length	9	11	9	9	11	9	9	9	9
mapAdd	7	7	7	7	7	7	7	7	7
mapDouble	11	11	11	11	11	11	–	–	11
maximum	7	7	7	7	7	7	–	–	7
member	11	11	11	11	11	11	11	11	11
multfirst	9	9	9	9	9	9	9	9	9
multlast	11	11	11	11	11	11	13	11	11
nth	13	13	13	13	13	13	–	–	13
replicate	9	9	9	9	9	9	9	9	9
reverse	9	9	9	9	9	9	9	9	9
stutter	–	13	13	13	13	13	–	–	13
sum	7	7	7	7	7	7	7	7	7

Table 5.2: Number of nodes of the synthesised solution.

5.1.2 Automatic black list generation

We also used our system to generate an automatic black list based on the identity function. Using the Plain exploration strategy and the *nof-nodes* cost function, we generated 100 programs representing the identity functions over integers, 100 over lists of integers and 100 over lists of lists of integers. The input-output examples were also generated automatically with the Plain exploration strategy and the *nof-nodes* cost function, to which we provided only constructors as library components.

We chose not to generate the polymorphic identity function. As during pruning we are ignoring types, holes and input variables, the programs that would have been generated for the polymorphic identity function are also generated for the identity over any specific type.

TODO: Either put the table of delete following sentence Table **??** summarizes 30 typical programs.

5.2 Factors affecting runtime

Two variants of our synthesis procedure were able to synthesise all 23 benchmarks in the presence of 37 library components, all variants synthesised at least 15 benchmark programs within the time limit. 78% of the benchmarks were synthesised within 1 s using BlackList as the exploration strategy and *nof-nodes-simple-type* as the cost function.

The search space is of exponential nature and depends on many factors: most notably the number of library components and the size of the solution to be synthesised. In the remainder of this section we look at these and other factors and their influence on the runtime.

5.2.1 Number of components

One well known factor that exponentially affects the runtime is the number of components provided to the synthesiser.

With 19 components we could synthesise all benchmarks with all procedures except the ones using the Template exploration strategy. With 37 components only two variants find all programs.

In particular, with 37 components, even if we provide `enumTo`, the synthesis benchmark `enumFromTo` times out for seven procedures out of nine, whereas with only 19 components this number is reduced to three. Interestingly, if we provide a 38th component, namely `drop`, then six procedures succeed in the synthesis of `enumFromTo`. This has very simple explanation: `enumFromTo` has a smaller solution that uses `drop`.

Since our synthesis procedures expand holes in a type aware manner, the number of components with the same type has an even higher impact on the running time than just the number of components. For example, if we add a constant of a new type `Foo` to the library, the running time will not increase much, because there are not many places, where we can use this component without causing a type error. On the contrary, if we would add another function from lists to lists like `tail` or `inits`, depending on the goal type we could have a considerable slowdown.

5.2.2 Size of the solution

TODO: If you have time, redo the graphics in latex, or at least find a way to move trend line in the legend In the previous sections we mentioned a second factor: the size of the solution to be synthesised. Figure 5.1 shows that the average running time for all nine variants of the synthesis procedure depends exponentially on the number of nodes of the solution found. This goes along with the intuition that a bigger program is more difficult to synthesise.

For example, if we have n possibilities to generate a program consisting of one node, that is `?x` where we have n possibilities to instantiate the hole `?x`, then we will have n^2 possibilities to generate a program with three nodes, that is $?x_1\ ?x_2$ where we have n possibilities to instantiate each hole.

TODO: Delete this question or do something with it. We cannot answer it, we do not have enough data. More surprisingly, the individual results show that there must be some other factor influencing the runtime. Take, for example, `enumFromTo`, `stutter` and `nth`. All three of them have a solution with exactly 13 nodes, but their runtimes differ at least by an order of magnitude. What does make `nth` generate in less than 1 s, `stutter` a hundred times slower and `enumFromTo` to time out in most of the cases?

In our simple intuitive explanation of the exponential dependency of the synthesis time with the size of the solution we completely ignored the contribution of types to search space pruning.

5.2.3 Cost functions

Cost functions are an instrument to prioritize some programs over others and as such have an impact on the running time. We extensively evaluated three of the cost functions described in Section 2.4: *nof-nodes*, *nof-nodes-simple-type* and *no-same-component*. In the following we will see how they affect the runtime and which programs they prefer.

nof-nodes prioritises shorter programs and prefers input variables to library components to holes, in the hope that smaller programs generalise better the examples. However, this cost function gives the same cost to the following two programs.

```
head [List Int → List Int] (nil [List Int → List Int])
    ?xs
map Int Int succ ?xs
```

It seems therefore natural that paired with the PLAIN strategy it usually leads to higher running times than other cost functions.

nof-nodes-simple-type additionally penalises arrow types appearing in type applications. We can see it in the solution for `length`. Two of the synthesis procedures that use this cost function find the larger solution

```
length [X] xs
    = sum (map [X] [Int] (const [Int] [X] (succ zero))
        xs)
```

instead of the smaller

```
length [X] xs
    = foldr [X] [Int] (const [Int → Int] [X] succ)
       zero xs,
```

because the second one contains an arrow type in a type application.

The impact on the runtime of this cost function is comparable to the introduction of pruning based on black lists. This has to do with the fact that polymorphic functions that apply in many cases but are rarely needed, like `const` and `flip`, tend to have long arrow types. In the BLACKLIST exploration strategy those programs are filtered by the black list, the *nof-nodes-simple-type* cost function assigns them a higher cost because of their types.

no-same-component prioritises smaller programs with simpler types and additionally penalises the use of the same component more than once. The hope is that without having to inspect programs that take a long time to evaluate like **TODO:** put an example of foldnat foldnat foldnat mul (mul 3 3) 3 or `enumTo prod enumTo prod`, running times will sink. Against expectations, this is usually not the case. The reason could be that the synthesis procedures that use this cost function examine many larger programs that to not contain any type applications, like **TODO:** put an example of succ add sub mul prod.

5.2.4 Examples

Another factor that greatly impacts on performance is the choice and the number of provided input-output examples. As our procedure evaluates every closed program it synthesises on at least the first input-output example, we must make sure that the first input-output example is

a. small enough, so that also undesirable programs like `enumTo (prod ( enumTo (prod xs)))` do not get stuck or run out of memory trying to construct a list with 479001600 elements, which happens already for the at first sight innocent input `[2,2,3]`.

b. expressive enough to rule out many programs, so that there is no need to fall back on the other, often bigger, input-output examples.

Clearly, using as few and as small input-output examples as possible has a positive effect on performance. On the other side, too few and too general input-output examples can lead to the synthesis of the wrong program, that is a program that satisfies all provided input-output examples but that does not generalise in the expected way. This was especially a problem with `enumFromTo` and `member`. **TODO:** Put a concrete example with enumFromTo `con Int _0 (drop Int _0 )enumTo _1))` well, this is a real solution
`filter Int (b_leq _0)(con Int b_zero (enumTo _1))`
On input `enumFromTo 1 3 = [1,2,3]` and `enumFromTo 2 4 = [2,3,4]` we get the program `con Int _0 (con Int (b_succ _0)(con Int _1 (nil Int)))`

for (1,2) and (1,3) we get `enumTo _1` or `b_foldNatNat (List Int)(con Int)( nil Int)_1` depending on the components we give.

for (1,2) and (2,3) we get `con Int _0 (con Int _1 (nil Int))`

for (1,2) and (2,4) we get `con Int _0 (map Int Int (b_add _0)(enumTo _0))`

5.2.5 Blacklist

TODO: Make and insert mentioned figures: the manual black list
The search space abounds of superfluous programs that are equivalent to

smaller ones. In Section 2.5 we introduced a way to leverage this inconvenience: Pruning based on black lists. This approach allows us not to explore further programs that will surely lead to a solution bigger than the optimal one, like `append [X] (nil [X])`?xs, or not lead to a solution at all, like `(head [`?X_1 $\rightarrow$?X_2 $\rightarrow$ X_3`] (nil [`?X_1 $\rightarrow$?X_2 $\rightarrow$ X_3`]))` ?x_1 ?x_2.

A longer black list allows to prune more superfluous programs and sinks considerably the number of programs our synthesis procedure needs to consider before finding a solution. However, black list pruning is extremely expensive in our implementation. Each element of the black list is matched against every subtree of every program with holes that is generated. That is, there is a trade-off between the length of the black list and the gain in performance that we can get.

Figure **??** shows the black list we used to evaluate the benchmarks. We manually compiled it combining unwanted patterns often seen in the search space with some carefully chosen automatically generated identity functions. We also added some extremely increasing functions like `foldNat [Int] ( foldNat [Int] (mul n)1)1 m` that represented a problem for our evaluator. **TODO:** If you have time, try to figure out what mathematical function it corresponds to.

In Table **??** we see that the runtime profits the most from the introducing of black list pruning when we use the cost function *nof-nodes*. The running time drops less significantly if we use other cost functions. A possible explanation of this behaviour could be the fact that other cost functions give a higher cost to those programs that are filtered with our black list.

We could also empirically see that pruning using black lists is very helpful in the presence of "useless" functions that increase the branching factor, for example `flip`, `const` or `uncurry`. Forbidding a fully applied `flip`, `const` or `uncurry` has a comparable effect on performance to taking those components out of the library. However, since we are not taking those components out of the library, we are still able to synthesis functions that need them.

5.2.6 Templates

TODO: Write this
Explain why templates perform so poorly (hm... because they don't do BFS, that is they have to explore every branch to the "end"?)

Maybe show "strange" solutions? But we already have them in the section about solutions.

Short section explaining that the templates you generate are not the templates you expect and why I found one example, where templates help! For `dropmax` I had a run out of memory exception with plain enumeration and I

could synthesise it in 7 seconds with templates! (Ok, I modified templates to put in only really higher-order components and close every hole, no matter the type). More resistant to the choice of examples.

5.2.7 Stack vs Queue expansion

As already mentioned in Section 2.3.1, we have two open questions in our best first search:

a. what program to expand next

b. which hole of this program to expand first

In the previous section we addressed the first question with different cost functions. In this section we focus on the second one.

Among all possible heuristics to determine which hole of the least-cost program to expand next, we chose to discuss two. In the first one the open holes of a program are organised in a stack, as opposed to the second one, where the open holes are kept in a queue.

Organising the open holes of a program into a stack leads to the expansion of the holes from left to right. To give some intuition we provide a derivation of `mapAdd` featuring the stack of open holes on the right of the program, where `xs` represents the input list and `n` the amount of the increment.

```
(?x0, [x0]) ⟶
(?x1 ?x2, [?x1, ?x2]) ⟶
(?x3 ?x4 ?x2, [?x3, ?x4, ?x2])) ⟶
(map [Int] ?x4 ?x2, [?x4, ?x2]) ⟶
(map [Int] (?x5 ?x6) ?x2, [?x5, ?x6, ?x2]) ⟶
(map [Int] (add ?x6) ?x2, [?x6, ?x2]) ⟶
(map [Int] (add n) ?x2, [?x2]) ⟶
(map [Int] (add n) xs, [])
```

Left-to-right expansion often lead to faster synthesis, because leftmost holes have usually more constraints on their type. Consider the program `?x3 ?x4 ?x2` from the derivation of . We know more about `?x3` than about `?x2`: The first one must be a function that takes two arguments of some type and returns a list of integers, whereas the second hole could be anything. Furthermore, the instantiation of `?x3` with `map [Int]` imposes some constraints on the types of `?x4` and `?x2`.

Keeping the open holes of a program in a queue leads to the expansion of the hole with the smallest depth first. This could be useful to control the depth of a program, but in practice it has a substantial drawback. Consider again the derivation of `mapAdd`. The first three steps are the same, but in the program `?x3 ?x4 ?x2` we would now try to expand the hole `?x2`, that we have absolutely no information about. Every library component and every input

variable are valid instantiations of this hole. Thus, this expanding strategy leads to a higher branching factor and explores many useless programs like `?`x_3 `?`x_4 `map` and `map (?`x_5 `foldr) xs`.

We used the stack-based expansion strategy throughout all runtime evaluations of the benchmarks.

5.3 Synthesised solutions

TODO: Rewrite programs in normal style and put listing blocks with programs instead of inlining everything

Most of the synthesised solutions are precisely the ones we would have written by hand. For some programs different variants of the synthesis procedure find two different valid programs of the same size. For example, for `replicate` we find following two solutions.

```
replicate [X] n x
    = map Int [X] (const [X] Int x) (enumTo n)
    = foldNat [List X] (con [X] x) (nil [X]) n
```

For the few benchmarks that can use `foldl` and `foldr` interchangeably, like `concat`, `maximum` and `sum`, different variants find different programs. The different variants of the synthesis procedure do not show clear preference for the one or the other. They can use `foldr` for one of such programs and `foldl` for the other.

Interesting is the case of `multfirst` and `multlast`, where following two solutions are found.

```
multfirst [X] xs
    = map [X] [X] (const [X] [X] (head [X] xs)) xs
    = replicate [X] (length [X] xs) (head [X] xs)
```

We omit the analogous solutions for `multlast` for brevity. The different synthesis procedures show a clear preference for the one or the other. For example, all synthesis procedures using the Template exploration strategy seem to prefer the use of the higher-order `map` to the first-order `replicate`. This has to do with the depth of the first-order search. The search from a particular template (in this case the template with no higher-order functions) times out before reaching programs with three components. On the other hand, the search starting from the template `map [?Y] [X] (const [?Y] [X] ?`x_1`) ?`x_2 succeeds within the timeout.

The preference of the Template exploration strategy for solutions containing higher-order functions leads to unexpectedly large programs. For example, one of the solutions for `factorial` is

```
factorial n
    = prod (foldr [List Int] [List Int]
        (foldl [List Int] [Int] (const [List Int] [Int]))
        (enumTo n)
        (nil [List Int]))
```

instead of the much simpler `prod (enumTo n)`. Note that since we are folding over an empty list, the two programs are completely equivalent.

There is a tendency to represent the constant integer 1 as `prod (nil [Int])` instead of `succ zero`. And even if we forbid with a black list the patterns

```
enumFromTo (succ zero) _
prod nil
```

the synthesis procedure with the BlackList exploration strategy still finds a way to express `enumTo` using `enumFromTo`: It simply falls back to

```
enumTo n
    = enumFromTo (div n n) n.
```

Of course, if we take the component `enumFromTo` out of the library we can generate the desired program

```
enumTo n
    = foldNatNat [List Int] (con [Int]) (nil [Int]) n.
```

Small programs are not always efficient. For example, for `enumFromTo` we find the solution

```
enumFromTo m n
    = con [Int] m (foldNat [List Int] (tail [Int]) (enumTo
        n) m)
```

that corresponds to generating a list from 1 to the second input and then dropping the first part of the list. The more efficient solution

```
enumFromTo m n
    = con [Int] m (map [Int] [Int] (add m) (enumTo (sub n
       m)))
```

is larger and thus it is generated only if we take `foldNat` out of the library.

Sometimes the solution found by the synthesiser suggested other benchmarks we could try to synthesise. For example, after examining the aforementioned solution for `enumFromTo` we realized that `drop` can be implemented as

```
drop [X] n xs
    = foldNat [List X] (tail [X]) xs n
```

Analogously, a "wrong" solution for `member` turned out to be a clever implementation of `isEven`, namely `foldNat not true n`. Folding over an integer is similar to recursion over that integer. In this case the base case of the recursion is `true` and in the inductive case we negate `isEven (n-1)`.
Another unexpectedly clever solution is due to the absence of a polymorphic equality function. We only had equality over integers, thus the benchmark `member` is not polymorphic but has the type `Int → List Int → Bool`. This allowed the synthesiser to generate, along with the expected solution

```
member n xs
    = not (is_nil [Int] (filter [Int] (eq n) xs)),
```

a special version that makes use of the fact that the product of a list is 0 if the list contains at least one 0 and that two numbers are equal if their difference is 0.

```
member n xs
    = is_zero (prod (map [Int] [Int] (sub n) xs))
```

This works in our implementation because our built-in integers can have negative values.

5.4 Automatic black list

We were able to automatically synthesise 300 programs corresponding to the identity function for three different types using automatically generated input-output examples in under 2 s. Because of their incremental nature, we need 8 automatically synthesised input-output examples as opposed to the 2-3 manual ones that would have been enough. Below that number there is no list of lists of integers that contains something but `nil` or no list of length two. This implies that synthesis using automatically synthesised input-output examples is slower than synthesis using manual ones.

For the evaluation of the benchmarks we preferred compiling a manual black list mainly because of three reasons:

1. All programs in the automatically generated black list correspond to the identity function.
2. Many automatically generated black list programs are unnecessary. For example, `append (append nil nil)_` and `concat (append nil _)` are not needed if the black list already contains `append nil _`.
3. The automatically generated programs are all closed programs and as such they are too concrete. For example, instead of `foldNatNat max _ zero`, `foldNatNat const _ zero` and `foldNatNat drop _ zero` we could just have the one program `foldNatNat _ _ zero` that generalises all the

programs with the idea that folding over the integer 0 is the same as taking the initial value, no matter which function is used for folding.

The first two points can be addressed with small modifications to the experimental set up: generate `nil`, `zero`, `undefined` and other constants as well for the first and prune the black list after or during generation for the second. The third point is way more complex. Partial evaluation of programs with holes could help to some extent, but at the end it is about the ability to abstract and generalise over programs.

5.5 Comparison to related work

TODO: Put the table in LaTeX
TODO: Write this Comparison to Feser, to Nadia, to Escher and to Myth. Say that you cannot really compare the numbers because they weren't run on the same machine. Say the thing with the size of the specification and the number of components. Say you have components capturing (tail?) recursive behaviour (fold over lists and fold over integers).

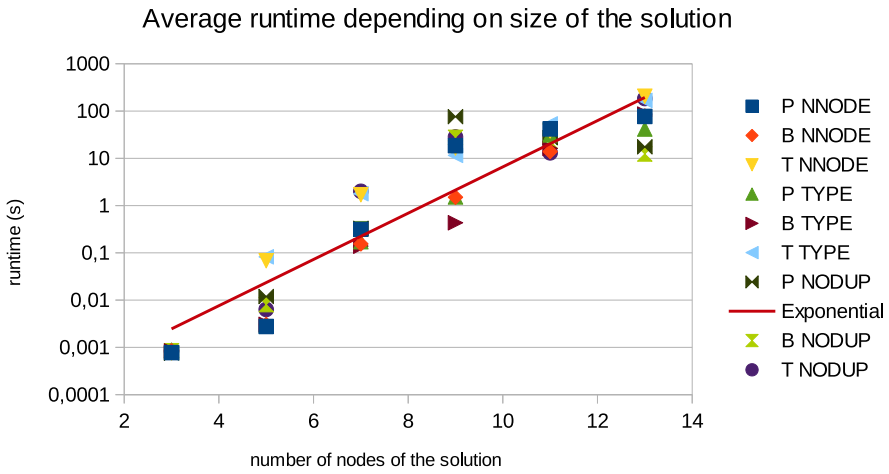

Figure 5.1: Average running time of the variants of the synthesis procedure depending on the number of nodes of the solution.

Chapter 6

Conclusions

6.1 Conclusions

The baseline is not that bad. Gathered some data about the search space. Incremental development (synthesise `enumTo` before synthesising `enumFromTo` and use it as a component).

6.2 Future Work

Templates done well, augmented examples?

Bibliography

[1] M Abadi, L Cardelli, and G Plotkin. Types for the scott numerals, 1993.

[2] 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.

[3] 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.

[4] Gérard P. Huet. Unification in typed lambda calculus. In *Proceedings of the Symposium on Lambda-Calculus and Computer Science Theory*, pages 192–212, London, UK, UK, 1975. Springer-Verlag.

[5] 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.

[6] B.C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[7] 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.