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?.

input-output examples are an intuitive and simple way to specify programs and make synthesis more accessible to users with a lower level of expertise.

1.2 Problem definition

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

Unlike our synthesis procedure, all of these tools are capable of synthesising recursive programs. This is not really a limitation, since common recursive patterns can be encoded as components. For example, the program p n = foldNatNat f init n can be translated into the recursive program **TODO:** Make sure it is f (n−1) (p (n−1)) and not f n (p (n−1)), correct if needed

```
p n = match n with
  | 0 → init
  | n → f (n-1) (p (n-1))
```

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 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 on 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)

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 [7]. 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 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. For example, the polymorphic identity function is defined as Λ`X.` λ`x:X. x`. 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 [7]. The syntax is summarized below.

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

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

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

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

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

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

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

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

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

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

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.

Target language 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.

Program A program is defined as the 4-tuple $\{\Xi, \Phi, \Delta \vdash t :: T\}$, 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 unwrap the constructor only once. Therefore they are more suitable for pattern matching.

```
Int = ∀R. R → (Int → R) → R
zero = ΛR. λz:R. λs:Int → R. z
     : Int
succ = λn: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 internal language, 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 [7] 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 internal language, 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 [7]. As usual, 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 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

In order to identify the library components that can be used to instantiate a hole of a given type, we need type unification. Consider, for example, a hole of type `List Int → Int`. We want to instantiate this hole not only with components that have precisely this type, like `sum` and `prod`, but also components with a more general type that can be matched to the desired type through type application, like `head :: ∀X. List X → X` and `length :: ∀X. List X → Int`.

Alas, unification on the type system of System F is undecidable [4]. Therefore we chose to restrict our type system to universally quantified types, that is types of the form $\forall X_1.\ \ldots\ \forall X_n.\ \texttt{T}(X_1, \ldots, X_n)$ where $\texttt{T}(X_1, \ldots, X_n)$ is quantifier-free. This allows us to completely ignore universal types during

unification. Instead, we can represent $\forall X_1.\ \ldots\ \forall X_n.\ \texttt{T(}X_1,\ldots,X_n\texttt{)}$ as $\texttt{T(?}X_1\texttt{ , }\ldots\texttt{, ?}X_n\texttt{)}$, that is leave out all quantifiers and replace the bound variables with fresh type holes.

Our unification algorithm (summarised as Algorithm 1 below) is based on the unification algorithm for typed lambda calculus from [7] and slightly modified to fit our needs.

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 $\mathcal{C}$. A substitution maps holes to types.

A type hole unifies with anything. An arrow type $T_1 \to T_2$ unifies either with a type hole or with another arrow type $T_3 \to T_4$ if T_1 unifies with T_3 and T_2 with T_4. A named type applied to all of its parameters $\texttt{C}\ T_{11}\ \ldots\ T_{1k}$ unifies either with a type hole or with the same named type applied to the same number of parameters $\texttt{C}\ T_{21}\ \ldots\ T_{2k}$ if the respective parameters T_{1j} and T_{2j} unify for all $j = 1,\ldots,k$. Universal types unify only with type holes and should not appear in the set of constraints.

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

After defining the target language, the evaluation semantics, the type checking and the type unification, we are ready to formally define the problem.

Problem definition Given a library Δ, a goal type T and a list of input-output examples $[(\Phi_1, o_1), \ldots, (\Phi_N, o_N)]$, find a closed term t in the target language such that

(i) the abstraction of t over all of its type and term input variables has the goal type under an empty variable binding context and an empty hole binding context, that is $\emptyset, \emptyset, \Phi_1, \Delta \vdash t' : T$ where t' is

$$\Lambda X_1. \ \ldots \ \Lambda X_j. \ \lambda x_1. \ \ldots \ \lambda x_k. \ [I_1 \mapsto X_1, \ldots, I_j \mapsto X_j, i_1 \mapsto x_1, \ldots, x_k \mapsto x_k]t.$$

(ii) t satisfies all input-output examples, that is $\Phi_n, \Delta \vdash t \longrightarrow^* t'$ and $\Phi_n, \Delta \vdash o_n \longrightarrow^* t'$ for all $n = 1, \ldots, N$.

In Section 2.3.1 we define the search space and in Section 2.3.2 we describe the main enumeration algorithm, a standard best-first search.

2.3.1 Search space

We see the search space as a graph, where the vertices correspond to *programs*. Recall that a program is the 4-tuple $\{\Xi, \Phi, \Delta \vdash t :: T'\}$, where t is a term of the target language. The type T' can be transformed into the goal type T abstracting over all type and term input variables. That is, if Φ contains the type input variables $I_1, \ldots, I_j$ and the signatures of the term input variables $i_1 : T_1, \ldots, i_k : T_k$, then the goal type T should be equal to

$$\forall X_1. \ \ldots \forall X_j. \ [I_1 \mapsto X_1, \ldots, I_j \mapsto X_j](T_1 \to \ldots \to T_k \to T').$$

There is a directed edge between two programs $\{\Xi_1, \Phi, \Delta \vdash t_1 :: T'\}$ and $\{\Xi_2, \Phi, \Delta \vdash t_2 :: T'\}$ if and only if the judgement *derive* (defined below) $\Xi, \Phi, \Delta \vdash t_1 :: T_1 \Rrightarrow \Xi', \Phi, \Delta \vdash t_2 :: T_2$ holds between the two.

To express the rules of the derive judgement 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 $\mathcal{E}[t]$.

The derive judgement can be summarised by the following four rules.
D-VarLib replaces a hole $?x$ with a type application of a library component c to the right types, if the type of c unifies with the type of $?x$. D-VarInp replaces a hole $?x$ with an input variable i from the context Φ, if it has the

right type. D-VarApp turns a hole into a term application of two fresh holes. The rule D-App chooses a hole in the program and expands it according to one of the three rules above.

TODO: do we need following paragraph? It adds nothing to the rule.

The rule D-VarLib might require a more detailed explanation. Let $\mathcal{A}$ be the class of quantifier-free types. Let `c` : $\forall X_1. \ldots \forall X_n.\ T_c(X_1, \ldots, X_n) \in \Delta$ where $T_c(X_1, \ldots, X_n)$ belongs to class $\mathcal{A}$. Let $?X_1, \ldots, ?X_n$ be fresh type holes not present in Ξ. Let σ be the unifier of T with $T_c(?X_1, \ldots, ?X_n)$. Then we can replace the hole `?x` with `c` $[\sigma(?X_1)] \ldots [\sigma(?X_n)]$.

The notation $\sigma(\Xi)$ denotes the application of the substitution σ to all types appearing in Ξ.

$$\frac{\begin{array}{c} c : \forall X_1. \ldots \forall X_n.\ T_c(X_1, \ldots, X_n) \in \Delta \\ ?X_1, \ldots, ?X_n \text{ are fresh type holes} \\ \sigma \text{ unifies } T \text{ with } T_c(?X_1, \ldots, ?X_n) \\ \Xi' = \Xi \cup \{?X_1, \ldots, ?X_n\} \setminus \{?x : T\} \end{array}}{\Xi, \Phi, \Delta \vdash ?x :: T \Rightarrow \sigma(\Xi'), \Phi, \Delta \vdash c\ [\sigma(?X_1)]\ \ldots\ [\sigma(?X_n)] :: \sigma(T)}\ \text{D-VarLib}$$

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

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

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

Note that all derived programs are well-types and, since the types of all derived programs unify with the types of their ancestors, have the right type.

2.3.2 Best-first search

Our enumeration procedure traverses the search graph defined in the previous section using standard best-first search. The algorithm maintains a frontier of candidate programs and expands one hole of the most promising program in each iteration. Closed programs are evaluated on the input-output examples. The search terminates when the first program that satisfies all input-output examples is found.

There are two points where Algorithm 2 can be tweaked.

1. Which candidate program is the most promising?
2. Which hole of the most promising program should be expanded?

The first question is addressed by the implementation of the `compare` function over programs. Section 2.4 discusses different approaches to define it.

The second question is addressed by the implementation of the `successor` function. In particular, the implementation of the D-App rule. There are two easy ways to handle this problem. The first way is to always expand the leftmost hole, the second way is to always expand the oldest hole.

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: General TODO: decide how do you want to call your cost functions. Now they are everywhere called nof-nodes, nof-nodes-simple-type and no-same-component. Maybe you want some simpler and shorter name.

The compare function in the best-first search algorithm can be defined as `cost` p_1 `-` `cost` p_2. There are different possibilities to define this cost

function. We will present four alternatives. All of them are based on the idea that shorter and simpler programs generalise better to unseen examples, along the lines of the Occam's razor principle [5]. The first three alternatives were evaluated on benchmarks and their effect on performance is discussed in Section 5.4.3.

nof-nodes The first cost function is based only on the number of nodes of the term. It prioritises shorter programs and prefers input variables over library components over holes.

$$\begin{aligned}
&\textit{nof-nodes}(c) = 1\\
&\textit{nof-nodes}(?x) = 2\\
&\textit{nof-nodes}(i) = 0\\
&\textit{nof-nodes}(t_1\ t_2) = 1 + \textit{nof-nodes}(t_1) + \textit{nof-nodes}(t_2)\\
&\textit{nof-nodes}(t\ [T]) = 1 + \textit{nof-nodes}(t)
\end{aligned}$$

nof-nodes-simple-type The second cost function also adds a factor based on the size of the types appearing in the term. It penalises thus terms with type application depending on the applied types. In particular, arrow types appearing in type applications are heavily penalised.

$$\begin{aligned}
&\textit{nof-nodes-type}(X) = 1\\
&\textit{nof-nodes-type}(?X) = 0\\
&\textit{nof-nodes-type}(I) = 0\\
&\textit{nof-nodes-type}(C\ T_1\ \ldots\ T_k) = 0\\
&\textit{nof-nodes-type}(T_1 \to T_2) = 3 + \textit{nof-nodes-type}(T_1) + \textit{nof-nodes-type}(T_2)
\end{aligned}$$

$$\begin{aligned}
&\textit{nof-nodes-term}(c) = 1\\
&\textit{nof-nodes-term}(?x) = 2\\
&\textit{nof-nodes-term}(i) = 0\\
&\textit{nof-nodes-term}(t_1\ t_2) = 1 + \textit{nof-nodes-term}(t_1) + \textit{nof-nodes-term}(t_2)\\
&\textit{nof-nodes-term}(t\ [T]) = 1 + \textit{nof-nodes-term}(t) + \textit{nof-nodes-type}(T)
\end{aligned}$$

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

no-same-component In the third cost function we additionally penalize terms that use the same component more than once.

$$\begin{aligned}
&\textit{nof-nodes-type}(?X) = 3\\
&\textit{nof-nodes-type}(I) = 0\\
&\textit{nof-nodes-type}(C\ T_1\ \ldots\ T_k) =\\
&\qquad 4 + \textit{nof-nodes-type}(T_1) + \ldots + \textit{nof-nodes-type}(T_k)\\
&\textit{nof-nodes-type}(T_1 \to T_2) = 5 + \textit{nof-nodes-type}(T_1) + \textit{nof-nodes-type}(T_2)
\end{aligned}$$

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

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

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

string-length The simplest and most imprecise method to take both the number of nodes and the complexity of the types appearing in the term into account is to define the cost of a term as the length of the string representing that term. This method also allows a simple way to weight differently the various library components by choosing a shorter or longer name. However, we decided not to use this cost function for evaluation.

2.5 Black list

Recall the 'replicate' example, where we saw the superfluous program `const [?X] ?`x_1 `?`x_2. The best-first enumeration explores many superfluous branches like `foldr [?X] [List ?X] (cons [?X]) (nil [?X]) ?xs` or `add zero ?n`. Such programs can be ruled out only based on the semantics of the library components. A simple way to prune those superfluous branches is to compile a list of undesired patterns and check each generated program against this list. This is what we call *black list pruning*.

A black list is a list of terms of the target language. Programs that contain a subterm that matches a term from the black list are removed from the candidate programs and their successors are ignored.

The relation *matches* over terms is inductively defined as follows. As you

$$\textit{matches}(?x, t)$$
$$\textit{matches}(i, t)$$
$$\textit{matches}(c, c)$$
$$\textit{matches}(t_1\ t_2, t_3\ t_4) \text{ if } \textit{matches}(t_1, t_3) \text{ and } \textit{matches}(t_2, t_4)$$
$$\textit{matches}(t_1\ [T_1], t_2\ [T_1]) \text{ if } \textit{matches}(t_1, t_2)$$

can see, holes and input variables in the black list match every subterm, a library component matches only itself, a term application matches a term application whose respective left- and right hand sides match and a type

application matches a type application if the left hand sides match. Note that the types in a type application are completely ignored.

Pruning based on black lists can be easily integrated in Algorithm 2. The result is shown in Algorithm 3, where the differences to the original best-first search are highlighted in blue.

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 $\leftarrow$ PriorityQueue.empty compare
queue $\leftarrow$ 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 $\leftarrow$ successor (PriorityQueue.top queue)
    queue $\leftarrow$ PriorityQueue.pop queue
    for *all s in successors* **do**
      queue $\leftarrow$ PrioriryQueue.push queue s
    end
  else
    queue $\leftarrow$ PriorityQueue.pop queue
  end
end
Output: PriorityQueue.top queue

Algorithm 3: Best first search with black list

In Section 5.3 we discuss how to synthesise such a black list automatically.

2.6 Templates

In this section we present a slightly different way to explore the search space. The idea is to fix all higher-order components first, producing a *template* for a program, and then fill in the remaining holes with input variables and first-order components. Since programs usually contain only a few higher-order components, the enumeration of templates takes less time than the enumeration of programs of comparable size. Moreover, templates encode well-known patterns of computation and impose meaningful constraints on the remaining holes. Therefore, it should be easy to find the desired program starting from the right template. This allows us to quickly abandon templates if no program satisfying the input-output examples is found within a short timeout.

Let us formally define a template and describe the procedure.

The library Δ is split into two contexts: Δ_h, containing all higher-order components, and Δ_f, containing the first-order ones. We also need to introduce

a new kind of term: the *delayed hole* $\underline{?x}$. This is a hole, whose instantiation is delayed to the first-order search. The context Ξ binds, additionally to normal holes, delayed holes as well. A *template* is a term in the target language that may contain delayed holes but does not contain input variables. A template is called *closed* if it does not contain holes (it may, however, contain delayed holes).

The synthesis procedure iterates between two phases: enumeration of templates and, as soon as a closed template is found, enumeration of programs as in Algorithm 2 using Δ_f for a limited period of time or until a program satisfying all input-output examples is found.

We additionally restrict the space by requiring a template to have no more than M higher-order components and no more than P delayed holes. Templates are enumerated according to the rules listed below. Those are very similar to the ones defined in Section 2.3.1, except that we cannot instantiate a hole with an input variable but we can delay a hole. 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.

Analogously to D-VarLib, the rule G-VarLib instantiates a hole with a type application of a higher-order library components to the right types, if the type of the component unifies with the type of the hole. The rule G-VarDelay delays the instantiation of a hole to the first-order search. G-VarApp replaces a hole with a term application of two fresh holes. G-App expands one of the holes of the template according to one of the three rules listed above. Note that there are no rules to expand a delayed hole.

$$\frac{\begin{array}{c}|\Xi| \leq P \text{ and } m < M \\ c : \forall X_1.\ \ldots \forall X_n.\ T_c(X_1, \ldots, X_n) \in \Delta_h \\ ?X_1, \ldots, ?X_n \text{ are fresh type holes} \\ \sigma \text{ unifies } T \text{ with } T_c(?X_1, \ldots, ?X_n) \\ \Xi' = \Xi \cup \{?X_1, \ldots, ?X_n\} \setminus \{?x : T\}\end{array}}{\Xi, \Phi, \Delta_h \vdash ?x :: T \mapsto \sigma(\Xi'), \Phi, \Delta_h \vdash c\ [\sigma(?X_1)]\ \ldots\ [\sigma(?X_n)] :: \sigma(T)}\ \text{G-VarLib}$$

$$\frac{|\Xi| \leq P \text{ and } m \leq M}{\Xi, \Phi, \Delta_h, m \vdash ?x :: T \mapsto \Xi \setminus \{?x : T\} \cup \{\underline{?x} : T\}, \Phi, \Delta_h, m \vdash \underline{?x} :: T}\ \text{G-VarDelay}$$

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

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

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), De-Bruijn indices (make sure you spell the name correctly)
- Library syntax and the type-checking when added to the library?
- eager evaluation, describe evaluator
- Table of implemented components

Chapter 4

Related Work

In this chapter we look at four state-of-the-art tools closely related to our work. They all synthesise functional programs, are based on inductive enumeration and use type information to restrict the search space. Since simple types are too ambiguous to specify a program, the tools we present either choose to complement type information with input-output examples or resort to more complex and more expressive types that can actually act as a specification, as for example the *refinement types* from [8].

4.1 SYNQUID

In [8] SYNQUID is proposed. The code can be found online[1] and there is the possibility to try it in the browser. This tool uses *refinement types* (types decorated with logical predicates) to prune the search space and to specify programs. SMT-solvers are used to satisfy the logical predicates appearing in the types.
Refinement types were already successfully used for verification. In particular, the tool builds upon the liquid types framework [9]. However, it proposes a 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.

This tool targets a language that includes lambda expressions, pattern matching, structural recursion, conditionals and fixpoint. The user can define custom functions and inductive datatypes that can be passed to the synthesiser as components.

A program is specified by providing a type signature. For example, the synthesis goal `replicate` can be specified as follows.

[1] https://bitbucket.org/nadiapolikarpova/synquid

```
n : Nat → x : α → {List α | len ν = n}
```

This is a dependent function type that denotes functions that, given a natural number n and an x of type α, return a list of α of length n. Here ν is a special logical value variable that in this case denotes the runtime return value of the functions and `len` is a measure function defined over lists.

This form of specification can be a disadvantage, since it is not that accessible to users with a lower level of expertise as input-output examples. It is not always easy to see which measure function for a custom datatype will lead to the most simple and intuitive specification. On the other hand, refinement types allow to express programs that manipulate data structures with non-trivial universal and inductive invariants in a concise way. This allows to synthesise programs on sorted lists, unique lists, binary search trees, heaps and red-black trees.

This tool synthesises simple programs over lists and integers in under 0.4 s. It can also handle more complex benchmarks that are out of the scope of this thesis, such as different sorting algorithms and manipulations of data structures with complex invariants. Various sorting algorithms over lists and trees are synthesised in under 5 s. The synthesis of the most complex benchmark, the balancing of a red-black tree, takes up to 20 s.

In contrast to our tool, the number of components provided to the synthesiser for evaluation is small.

4.2 λ^2

The tool proposed in [3] is called λ^2 and generates its output in λ-calculus with algebraic types and recursion. The target language also includes 7 higher-order combinators such as `map`, `fold` and `filter` and a flexible set of primitive operators and constants.

The user specifies the desired program providing only input-output examples. No particular knowledge is required from the user, as was demonstrated using randomly generated input-output examples. The goal type is inferred from the examples.

The synthesis algorithm is a combination of inductive generalisation, a limited form of deduction and enumerative search. First, it generates *hypotheses* in a type-aware manner, that is programs with free variables such as `λx. map ?f x` where `?f` is a placeholder for an unknown program to be synthesised. Then deduction in form of hand-coded rules about the higher-order combinators is used either to refute a hypothesis or infer new input-output examples to guide the synthesis of missing functions. For example, the hypothesis `λx. map ?f x` will be refuted if the length of the input list does not match

the length of the output length. Enumerative search is used to enumerate candidate programs to fill in the missing parts of hypotheses. Hypotheses and candidate programs are organised in a priority queue and, at each point of the search, the least-cost candidate is picked.

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. It synthesises all benchmark programs in under 7 minutes. Half of the benchmarks is synthesised in under 0.43 s. However, the synthesis of `droplast`, the program that drops the last element of a list, takes up to 320 s. The program that removes duplicates from a list, the program that drops the smallest elements of each list of a list of lists and the program that inserts a tree under each leaf of another tree take more than 100 s to synthesise.

Unlike SYNQUID and our work, this tool can only use the 7 hard-coded higher-order combinators. The extension of the set of higher-order combinators with own functions is not easily supported.

4.3 ESCHER

In [2] ESCHER is presented. This tool targets a simple untyped purely functional language consisting of constants, input variables, conditionals and library components applied to all of their arguments, including a special component `self` referring to the program being synthesised. This last component is used to synthesise recursive programs.

The user specifies the desired program as a *closed* set of input-output examples. That is, for each input-output example, all examples needed to evaluate every recursive call must be present. For example, if we want to specify `replicate` as

```
replicate 2 'a' = ['a','a'],
```

we also need to provide the input-output examples for the possible recursive calls, that is

```
replicate 1 'a' = ['a']
replicate 0 'a' = [].
```

This is necessary because recursive programs are evaluated using the input-output examples as an oracle. However, it is not always easy for an inexperienced user to provide such a set.

The search is goal-directed. Programs are associated with value vectors, that is the vector of the outputs of the program on the inputs from the input-output examples. Programs sharing the same value vectors are considered equivalent, that is the search space is pruned based on observational

equivalence. The algorithm alternates between two phases: forward search and conditional inference. During forward search programs are inductively enumerated by adding new components to already synthesised programs. During conditional inference a novel data structure, the *goal graph*, is used to detect when two synthesised programs can be joined by a conditional statement. The alternation between the two phases is guided by a heuristic.

This tool is able to synthesise recursive programs, including tail recursive, mutually recursive and divide-and-conquer. It synthesises all benchmarks in under 11 s and all but three benchmarks in under 1 s. The benchmarks include programs on integers such as `fibonacci` and `isEven`, programs on lists such as `compress` and `insert` and programs on trees such as `nodes-at-level` and `count-leaves`.

Like our work, this tool can handle a flexible set of components. For example, a set of 23 components was used to evaluate all benchmarks. However, there was no higher-order component among them.

4.4 Myth

TODO: Write this section

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 [6] 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 5.2 and the results are discussed in Section 5.2.1. Section 5.4 discusses the factors that affect the runtime.

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 5.3 describes how we used the synthesis procedure to generate a black list and Section 5.3.1 reviews the quality of the generated black list.

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.

5.2 Performance evaluation

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 5.4. 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 5.1 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.

Components In the first three columns of Table 5.1 all benchmarks except for `nth` share the same set of components, listed in Table 5.3. For the synthesis of individual benchmarks appearing in the library as components we took the corresponding component out of the library. 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.

Timeout 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.

5.2.1 Results

Two variants of our synthesis procedure were able to synthesise all 23 benchmarks in the presence of 36 to 37 library components, all variants synthesised at least 18 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 variants that use the BlackList exploration strategy can synthesise the most number of benchmarks: for two cost functions they generate all 23 benchmark programs within the time limit, for the cost function *no-same-component* they fail to synthesise `enumFromTo`. They synthesise half of the benchmarks in under 0.2 s on average and 17 benchmark programs in under 1 s.

The variants that use the Plain exploration strategy can synthesise from 21 to 22 benchmark programs depending on the cost function. They are on average 8 times slower than the variant that combines the BlackList exploration strategy with the *nof-nodes-simple-type* cost function. However, 15 benchmarks are synthesised less than 3 times slower and for 3 of them the running times are actually lower. In the case of `enumTo` the solution found by the other variants, `enumTo n = enumFromTo (succ zero) n`, contains a pattern forbidden by the black list we used to prune the search space. The other two benchmarks have very short, simple solutions for which the overhead of checking every program against the black list is not balanced out by a substantial pruning of the search space.

The variants of our synthesis procedure that use the Template exploration strategy can synthesise only 18 to 19 benchmark programs within the time out. Moreover, they are on average 1680 times slower than the variant that uses the `BlackList` exploration strategy combined with the *nof-nodes-simple-type* cost function. However, this value is pushed up by the benchmark `factorial`. For half of the 18 benchmarks all variants can synthesise this value is less than 20 and 15 of them are no more than 90 times slower than the variant that combines BlackList with *nof-nodes-simple-types*.

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	≈ 19K	≈ 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: Runtime of nine variants of our synthesis procedure on 23 benchmarks. Each cell shows nine numbers corresponding to the different variants, organised in a 3×3 square. The rows of the square a labeled with the exploration strategies: P for PLAIN, B for BLACKLIST and T for TEMPLATE; the columns of the square are labeled with the cost functions: NODES for *nof-nodes*, TYPES for *nof-nodes-simple-type* and NODUP for *no-same-component*. The first column shows the runtimes in seconds when 36 to 37 components are provided to the synthesiser. The second column is the ratio between the running times of the first column and the minimum synthesis time for that benchmark. The third column shows the speedup with respect to the first column if we reduce the number of library components to 18-19.

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.

	library components
general functions	`const`, `flip`
booleans	`true`, `false`, `not`
integer constructors	`zero`, `succ`
integer destructors	`isZero`
integer combinators	`foldNat`, `foldNatNat`
arithmetics	`add`, `mul`, `div`, `max`, `eq`, `neq`
list constructors	`nil`, `con`
list destructors	`head`, `tail`, `isNil`
list combinators	`map`, `foldr`, `foldl`, `filter`
list functions	`length`, `append`, `reverse`, `replicate`, `concat`
list of integers functions	`sum`, `prod`, `maximum`, `member`, `enumTo`, `enumFromTo`

Table 5.3: 37 library components used for synthesis

5.3 Automatic black list

We also used our system to generate an automatic black list based on the identity function. 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. We chose to generate programs corresponding to the identity function over integers, lists of integers and lists of lists of integers.

To that end, we first used the synthesis procedure that combines the Plain exploration strategy with the *nof-nodes* cost function to synthesise the first 8 programs of type `Int`, respectively `List Int` or `List (List Int)`. For this step we provided only the constructors `con`, `nil`, `succ`, `zero` to the syn-

thesiser. We paired each synthesised program with itself to generate and input-output example.

As a second step, for each of the following goal types

```
Int → Int
List Int → List Int
List (List Int) → List (List Int)
```

we used the synthesis procedure that combines the PLAIN exploration strategy with the *nof-nodes* cost function to synthesise the first 100 programs that satisfy the 8 input-output examples of the corresponding type generated in the previous step. This time we provided the synthesiser with the 37 components listed in Table 5.3.

After removing duplicates and the classical program corresponding to the identity function, that is `id x = x`, we got 212 black list patterns. Section 5.3.1 reviews the quality of the generated black list.

5.3.1 Results

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.

5.4 Factors affecting runtime

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.

5.4.1 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 a 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.

5.4.2 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.

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

5.4.3 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* with three different exploration strategies. 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, under the hypothesis that smaller programs generalise better to 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 instantiate their type variables with 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 `enumTo (prod (enumTo (prod xs)))`, 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 do not contain any type applications, like `enumTo (mul (succ (div n m)) (succ ?x))`.

5.4.4 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 as a stack, as opposed to the second one, where the open holes are kept in a queue.

Organising the open holes of a program as a stack leads to the expansion of the holes from left to right. To give some intuition, we provide a derivation of `mapAdd` showing 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 leads to faster synthesis, because leftmost holes usually have 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 `?`x_3 with `map [Int]` imposes some constraints on the types of `?`x_4 and `?`x_2.

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 `?`x_3 `?`x_4 `?`x_2 we would now try to expand the hole `?`x_2, 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 superfluous 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.

5.4.5 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`. For example, if we provided `enumFromTo` only with examples that result in a list of length three, we got the program that simply concatenated the first input with its successor with the second input.

```
enumFromTo m n
    ≠ con [Int] m (con [Int] (succ m) (con [Int] n (nil [
   Int])))
```

If we provided `enumFromTo` only with examples starting with 1, the synthesised solution was just a call to `enumTo` with the second input variable as

argument or, depending on the components we gave, the program corresponding to `enumTo n`.

```
enumFromTo m n
    ≠ foldNatNat [List Int] (con [Int]) (nil [Int]) n
```

And for the two examples `enumFromTo 1 2` and `enumFromTo 2 4` that we carefully chose so that the output lists had different lengths and so that the first arguments were different, we got the program that completely ignores the second argument, as it assumes that the length of the resulting list is the successor of the first argument.

```
enumFromTo m n
    ≠ con [Int] m (map [Int] [Int] (b_add m) (enumTo m))
```

5.4.6 Blacklist

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 `→ ?`X_2 `→` X_3`] (nil [?`X_1 `→ ?`X_2 `→` 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, in our implementation black list pruning is extremely expensive. Each element of the black list is matched against every subterm 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 5.4 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 rapidly increasing functions. For example, following program computing tetration[1] represents a problem for our evaluator.

```
tetration a n =
    foldNat [Int] (foldNat [Int] (mul a) 1) 1 n
```

```
 append nil                    head (enumFromTo _ _)
```

[1] Tetration, written as ${}^{n}a$ or $a \uparrow\uparrow n$, is the operation defined as

$$\underbrace{a^{a^{\cdot^{\cdot^{a}}}}}_{n \text{ times}}.$$

```
append _ nil                    head (enumTo _)
add _ zero                      head (map _ _)
add zero                        head nil
div _ zero                      head (replicate _ _)
div _ (succ zero)               isNil nil
div zero                        length (con _ _)
div (succ zero)                 length (enumFromTo _ _)
foldNat _ _ zero                length (enumTo _)
foldNat succ zero               length (map _ _)
foldNatNat (foldNatNat _ _ _)   length nil
foldNatNat _ _ zero             length (reverse _)
isZero zero                     map _ nil
max zero zero                   maximum nil
mul (succ zero)                 not (not _)
mul _ (succ zero)               prod (con _ nil)
mul _ zero                      prod (con zero _)
mul zero                        prod nil
sub _ zero                      prod (reverse _)
sub zero                        replicate zero
concat nil                      reverse (con _ nil)
const _ _                       reverse (map _ (reverse _))
enumFromTo (succ zero)          reverse nil
enumTo zero                     reverse (reverse _)
enumTo (prod _)                 sum (con _ nil)
flip _ _ _                      sum nil
foldl _ _ nil                   sum (reverse _)
foldr con nil                   tail (con _ nil)
foldr _ _ nil                   tail (enumFromTo _ _)
head (con _ _)                  tail nil
```

Table 5.4: Manually compiled black list patterns used for evaluation in the BlackList exploration strategy.

In Table 5.1 we see that the runtime profits the most from the introduction 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 polymorphic functions that apply in many cases but are rarely needed, 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 synthesise functions

that need them.

5.4.7 Templates

In Table 5.1 we see that the synthesis procedures that use the Template exploration strategy fail more often to find a solution within the timeout. Moreover, even if they find a solution, they tend to be 10 times slower than the other synthesis procedures, `length`, `member`, `replicate` and `reverse` being an exception.

The main reason for this slowdown resides in our implementation. In particular, in the successor rules we presented in Section 2.6. Consider following derivation of a template for `replicate`, where `X` represent the input type variable, `n` is the first argument and `x` is the second. The list on the right of each program shows the type of its not delayed holes.

```
(?x0,          [?x0 :: List X]) ⟶
(?x1 ?x2,        [?x1 :: ?Y → List X, ?x2 :: ?Y]) ⟶
(?x1 ?x2,         [?x2 :: ?Y]) ⟶
(?x1 (foldr [?Z1] [?Z2]),    [])
```

Note that the only thing we can do with $?x_1$ is to delay it, because in our library there is no higher-order component that takes only one argument. On the other hand, $?x_2$ can be instantiated with every higher-order function of the library, because delaying $?x_1$ does not constrain its type in any way. This means that, before having a chance to explore a sensible template with 4 leaves like `foldl [?X] [?Y] ?f ?init ?xs`, the synthesiser must explore up to a certain depth many non that sensible but smaller templates like `?x ( foldr [?X] [?Y])`.

The solutions synthesised by the synthesis procedures that use the Template exploration strategy tend to contain more higher-order components and in two cases are surprisingly long. In Table 5.2 we can see that two variants of our synthesis procedure synthesise a program with 13 nodes for `factorial`, whereas all other variants find one with only 5 nodes.

Despite of those drawbacks, the Template exploration strategy is still interesting: it is more resilient to the choice of input-output examples compared to the other two. For example, if we provide a slightly larger input-output example for `dropmax`, six variants of our synthesis procedure run out of memory, whereas the three variants that use the Template exploration strategy still find the solution in under 8 s.

5.4.8 Unknown factors

Individual results show that there must be other factors 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 for the synthesis procedures that do not time out on `stutter`. 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?

5.5 Synthesised solutions

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)

replicate [X] n x
    = 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

multfirst [X] 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.

5.6 Comparison to related work

We compare our synthesis procedure that uses the Blacklist exploration strategy combined with the *nof-nodes-simple-type* cost functions with the state-of-the-art tools reported in Chapter 4. The results are summarised in Table 5.5, where for each tool we provide the specification size (expressed in number of examples for example-based tools and in AST nodes for Synquid) and the runtime.

Since the running times were taken from the respective papers and were not run on the same machine, we cannot directly compare performance. However, the case of `droplast`, where λ^2 takes around 300 s and our tool only 0.06 s, cannot be explained only by the difference in the hardware. Providing the right components, in this case `reverse` and `tail`, helps the synthesis tool to find a solution faster.

Another thing that impedes direct comparison of the running time is that the other tools generate a richer class of programs. All of them are capable of generating recursive functions and most of them have support for conditionals and pattern matching, whereas the target language of our tool includes only application of components and input variables. On the other hand, components can encode well-known recursion patterns. For example, programs that use the component `foldNat` naturally translate to a recursive program, as we see from the example of `isEven`.

```
isEven n = foldNat not true n

isEven n = match n with
  | 0 → true
  | 1 + m → not (isEven m)
```

It might be a consequence of the simplicity of our target language, but as we can see in Table 5.5, our tool needs less input-output examples to synthesise the benchmarks than the other example-based tools. Synquid relies on a different specification that requires a higher level of expertise, which makes direct comparison difficult.

The number of provided components also changes across the tools. With 36 to 37 components, we provide the largest library to our tool. On the second place, Escher uses a library of 23 components to evaluate all benchmarks. Other tools handle less components. For example Synquid does not provide more than 5 components and 6 measure functions over the needed data types to any of its benchmarks.

Considering that our tool is just standard type-aware best-first enumeration, it is surprising that only four benchmarks show a considerably worse (about two orders of magnitude) performance than the state-of-the-art.

	Our tool		Synquid		λ^2		Escher		Myth	
name	spec	time	spec	time	spec	time	spec	time	spec	time
append	2	0.07	8	0.05	3	0.23			12	0.01
concat	2	0.04	5	0.05	5	0.13	–	0.06	6	0.02
drop	2	0.05	14	0.29			–	0.02	13	1.29
droplast	2	0.06			6	316.39				
dropmax	2	0.6			3	0.12				
isEven	2	0					–	0.02	4	0.01
isNil	2	0	6	0.02						
last	2	0			4	0.02	–	0.02	6	0.09
length	2	28.44	4	0.1	4	0.01	–	0.01	3	0.02
mapAdd	2	0.25			5	0.04				
mapDouble	3	13.12	7	0.06	3	0.11				
maximum	3	0.24			7	0.46				
member	7	22.67	6	0.03	8	0.35				
multfirst	2	0.06			4	0.01				
multlast	2	0.59			4	0.08				
nth	3	0.34							24	0.96
replicate	3	0.08	7	0.06						
reverse	2	0.99	11	0.29	4	0.01	–	0.22		
stutter	2	15.23					–	0.55	3	0.02
sum	2	0.32			4	0.01	–	0.06	3	0.01

Table 5.5: Comparison of our tool with the state-of-the-art. For our tool, λ^2 and Myth the *spec* column shows the number of examples, for Synquid it shows the specification size in AST nodes. For all tools the *time* columns show the runtime in seconds. The cells corresponding to benchmarks that were not tested with the respective tool are left empty.

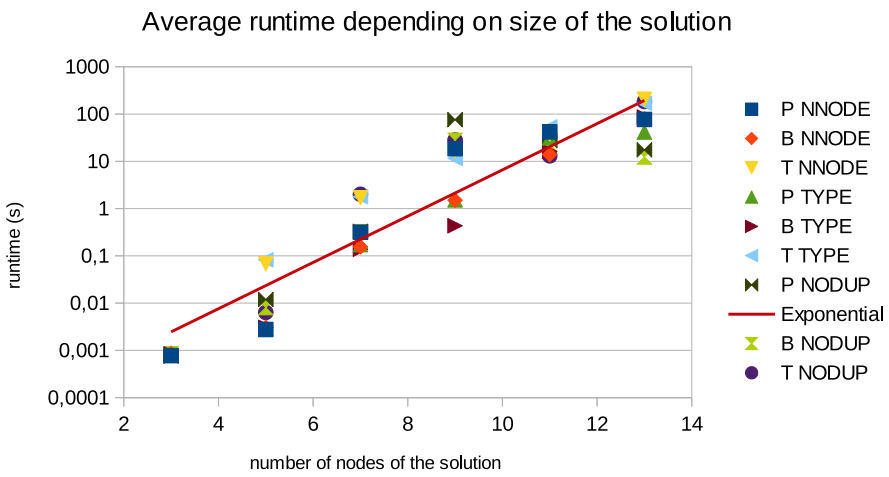

Figure 5.1: Average running time of the variants of the synthesis procedure depending on the number of nodes of the solution.

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] Umesh V. Vazirani Michael J. Kearns. *An introduction to computational learning theory*. MIT Press, 1994.

[6] 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.

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

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

[9] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Lan-*

guage Design and Implementation, PLDI '08, pages 159–169, New York, NY, USA, 2008. ACM.

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.