Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Top-Down Inductive Synthesis with Higher-Order Functions

Master Thesis

Alexandra Maximova

August 15, 2016

Advisors: Prof. Dr. Martin Vechev, Dimitar Dimitrov

Department of Computer Science, ETH Zürich

Abstract

TODO: write me :)

Contents

Chapter 1

Introduction

1.1 About program synthesis in general

put in some context, link **??**

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

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

1.2 Problem definition

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

1.3 Contributions

Evaluation, exploring the baseline algorithm, exploring the search space

Chapter 2

Related Work

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

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

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

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

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

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

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

5. What do they do? In [7] Synquid is proposed. Refinement types (types decorated with logical predicates) are used to prune the search space. SMT-solvers are used to satisfy the logical predicates. The key is the new procedure for type inference (called modular refinement type reconstruction), which thank to its modularity scales better than other existing inference procedures for refinement types. Programs can therefore be type checked even before they are put together.

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

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

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

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

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

5. What do they do? Inductive generalization, deduction, enumerative search. First, generate *hypotheses* (that is, programs with free variables like λx. map ?f x where ?f is a placeholder for an unknown program that needs to be generated) in a type-aware manner (the type is inferred from the I/O-examples). Then deduction based on the semantics of the higher-order combinators is used either to quickly refute a hypothesis or infer new I/O-examples to guide the synthesis of missing functions. Best-first enumerative search is used to enumerate candidate programs to fill in the missing parts of hypotheses.
   The tool proposed in [3] is called λ^2 and generates its output in λ-calculus with algebraic types and recursion. The user specifies the desired program providing input-output examples. No particular knowledge is required from the user, as was demonstrated using random input-output examples. The examples are inductively generalized in a type-aware manner to a set of hypotheses (programs that possibly have free variables). The key idea are the hard-coded deduction rules used to prune the search space depending on the semantics of some of the higher-order combinators (map, fold, filter and a few others). Deduction is also used to infer new input-output examples in order to generate the programs needed to fill in the holes in the hypotheses. This tool is able to synthesize programs manipulating recursive data structures like lists, trees and nested data structures such as lists of lists and trees of lists. The tool is able to synthesise all benchmark programs in under 7 minutes.

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

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

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

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

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

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

5. What do they do? Forward search: inductive enumeration, add a new component to an already synthesised program. Conditional inference: use the goal graph to see if you can join to synthesised programs by a conditional statement. A heuristic guides the alternation between forward search and conditional inference. Programs are associated with value vectors. Search space is also pruned based on *observational equivalence*, that is programs with equivalent value vectors are treated as equivalent programs.
   Can be formalised as a non-deterministic transition system over configurations (triples consisting of a set of synthesised programs, a goal graph and a list of input-output examples) with six transition rules: an initial rule to start the search, a terminate rule to terminate the search and four synthesis rules (one for forward search, two for conditional split and one to ask for new input-output examples to evaluate a recursive call). The recursive calls are answered by the oracle.
   Rule scheduling is added to turn this into a practical system (different heuristics).
   In [2] Escher is presented, an inductive synthesis algorithm that learns a recursive procedure from input-output examples provided by the user. The user must provide a "closed" set of examples, otherwise recursion cannot be handled properly The target language is untyped, first-order and purely functional. The algorithm is parametrized by components that can be instantiated differently to suit different domains. The approach combines enumerative search and conditional inference. The key idea is to use a special data structure, a *goal graph*, to infer conditional branches

instead of treating `if-then-else` as a component. Observational equivalence is also used to prune the search space. Programs with the same value vectors (output of the program when applied to the inputs of the input-output examples) are considered equivalent and only one of them is synthesized. An implementation of the tool was tested on a benchmark consisting of recursive programs (including tail-recursive, divide-and-conquer and mutually recursive programs) drawn from functional programming assignments and standard list and tree manipulation programs. For all examples the same fixed set of components was used. The tool is able to synthesize all of them quickly. There is very little information on how many input-output examples were needed to synthesize the benchmarks and how difficult it is for a non-experienced user to come up with a "closed" set of examples.

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

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

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

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

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

  5. What do they do? The tool in [5] is called Myth and uses not only type information but also input-output examples to restrict

the search space. The special data structure used to hold this information is the *refinement tree*. This system can synthesize higher-order functions, programs that use higher order functions and work with large algebraic data types.
There is an ML-like type system that incorporates input-output examples. Two pieces: a *refinement tree* and an enumerative search.
Two major operations: refine the goal type and the examples (push them down in the refinement tree) and guess a term of the right type that matches the examples (for one of the nodes of the refinement tree).

Chapter 3

Top down type driven synthesis

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.

3.1 Problem

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, foldr, foldl 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.

- Try to put components together in order to get a list. Fix X as a type variable and fix n to be an Int and x to be an X. Try to synthesise a List X. Which components can you use in order to get a List X? You 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 [].

```
replicate n x = []
```

Cool, we can evaluate this program straight away... and see that is does not satisfy the I/O-examples. It means, we must try further.

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

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

?Y is a fresh variable. We omit foldl for brevity.

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

replicate n x = const ?xs ?s
?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 (note that it has to be possible for all possible instantiations of X and note that we have only one value of type X: x, the second argument to replicate.
What are the possibilities to instantiate ?f? Obviously, we cannot use map or enumTo, because they return lists. But all other possibilities are still open (again, we leave out foldl for brevity).

```
replicate n x = map (foldr ?g ?init) ?xs
?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
```

Again, from all programs generated so far (6 in total) 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
?f = const ?x :: ?Y → X
?x = x :: X
?xs :: List ?Y
```

Let us directly decide this is the most promising program so far (how to decide which program are we going to expand next is explained in Section 3.4). Like at the beginning, we have to generate a list. But this time we can choose what the elements of the list are, therefore we cannot rule out enumTo. As you see, 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)
?f = const ?x :: ?Y → X
?x = x :: X
?xs = enumTo ?n :: List Int
?n :: Int
```

First we are going to evaluate the *closed* program (that is, without holes) replicate n x = map (const x) []... and see that it doesn't satisfy the I/O-examples. Then, as always, we take the most promising program and expand one of its holes. Let us decide that the most promising program is the last one. That is, we are searching 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. This time, we got two closed programs that we are going to evaluate, replicate n x = map (const x) (enumTo 0) and replicate n x = map (const x) (enumTo n). Evaluation shows that only the second one satisfies the I/O-examples.

- Important difference between X and ?X. The first one is fixed and cannot be instantiated with anything, the second one is a free variable that can be instantiated with anything we like.

This example show following concepts

- Hole: unknown part of a program, for which we only know the type and that we can expand
- we maintain a frontier of programs with holes

- from this frontier, we always chose the most promising one. What program is the most promising can be determined by a cost function (Section 3.4)
- some programs are not worth expanding. For example, we see straight away that is does not make much sense to instantiate a hole with const ?x ?y, because we could as well write ?x for it, which is always a shorter and preferable program. That is, some programs can be ruled out semantically (Section 3.5)
- The search space is quite big even with so few components. Exponential.

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

3.2 Calculus

In this section we will introduce three different calculus.

- System F as in the excellent book of Pierce [6]
- The system we derive from System F and in which the components are defined (includes holes, input variables and recursion for terms and types)
- A subset of the second system, used for generation (only application of components, holes or input variables is allowed).

The exposition will closely follow Pierce [6].

Let's start with the standart System F. Our calculus is based on System F. A subset of our calculus is used for generation.

- generate programs from restricted calculus
- calculus itself is still powerful, because we use it to define the library components as well.

3.2.1 Terms and Types

- similar to Pierce
- except holes, components and free variables
- recursion
- recursive types that can take parameters
- the search space is not the whole language but only a subset of it

Real System F

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.2.2 Encodings

Note that in the definition of types we do not see familiar types such as booleans, integers, lists or trees. All these types can be encoded in System F using either the Church's or the Scott's encoding [1]. We opted for the Scott's encoding because it's more efficient in our case. Scott's booleans coincide with Church's booleans and are encoded as follows.

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

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

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

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

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

3.2.3 Evaluation semantics

We usually evaluate only closed programs. Eager evaluation. Rules. Judgement $\Phi, \Delta \vdash t \longrightarrow t'$.

Define *value* v to be a term to which no evaluation rule apply.

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

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

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

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

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

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

3.2.4 Type checking

Question: why do I mathematically need that many contexts? How can I summarize T-Var, T-Hol, T-Inp, T-Lib in one rule?
Question: Should I type System F or only ”programs”? Answer: System F, of course. In the code you type library components as well.

I moved this section because type checking does not need unification.

The typing judgement $\Gamma, \Xi, \Phi, \Delta \vdash t : T$. Based on the book of Pierce.

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

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

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

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

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

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

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

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

3.2.5 Type unification

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

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

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

3.3 Search

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

3.3.1 Search space

Use only third system presented in Section System F.

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

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

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(C) is
  if C = ∅ then []
  else
    let {T1 = T2} ∪ C' = C in
    if T1 = T2 then
      unify(C')
    else if T1 = ?X and ?X does not occur in T2 then
      unify([X ↦ T2]C') ∘ [X ↦ T2]
    else if T2 = ?X and ?X does not occur in T1 then
      unify([X ↦ T1]C') ∘ [X ↦ T1]
    else if T1 = T11 → T12 and T2 = T21 → T22 then
      unify(C' ∪ {T11 = T21, T12 = T22})
    else if T1 = C [T11, T12, ..., T1k] and T2 = C [T21, T22, ..., T2k]
     then
      unify(C' ∪ {T11 = T21, T12 = T22, ..., T1k = T2k})
    else
      fail
    end
  end
end
```

Algorithm 1: Type unification

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

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

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

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

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

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

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

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

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

3.3.2 Exploration

Write also about stack vs queue of open holes

We explore the search graph using a best first search. We can play with the algorithm in two points (marked in blue): first, we can define which hole to expand first, second, we can choose the compare function of the priority queue. Different approaches to define the compare function are discussed in the next session. Concerning the order of expansion of the holes, we tried two strategies: the first one was maintaining a stack of holes (Ξ implemented as a stack), which leads to the expansion of the deepest hole first, the second one was maintaining a queue of holes (Ξ implemented as a queue), which leads to the expansion of the oldest hole first.

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

Algorithm 2: Best first search

3.4 Cost functions

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

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

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

Algorithm 3: Cost function based on the number of nodes

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

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

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

3.5 Black list

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

One could use the synthesis algorithm presented in Section 3.3.2 to automatically synthesise black lists. For example, one could synthesise many programs corresponding to the identity function or to the empty list or to any other term, and add all those programs but one to the black list.

Input: term t
Output: sum of weighted number of nodes in term t and weighted number of nodes in the types appearing in t

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

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

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

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

3.6 Templates

Top-down type-driven synthesis.

A template is a program with holes. We are interested in templates where all higher-order components are fixed and there are holes for the first-order components. The search space is thus similar to the search space described in 3.3.1, with the exception that Δ contains only the higher-order components. One of the new things are *closed holes* $\underline{?x}$. Those are holes that are supposed to be filled in later with first-order components.

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

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

3.6.1 Successor rules

The successor rules are very similar to the ones defined in 3.3.1, apart from little modifications. That is, now we have a successor rule to close a hole,

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

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

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

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

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

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

and we can not instantiate a hole with an input variable any more, because that is supposed to be done in the next step. All the rules are modified to take into account the restriction on the number of components and the number of closed holes. In order to do this, we need to pass along m, the number of higher-order components in the term whose subterms we are traversing.

So we can *close* a hole.

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

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

Input: goal type T, library components Δ, list of input-output examples $[(\Phi_1, o_1), \dots, (\Phi_N, o_N)]$, black list $[b_1, \dots, b_M]$
queue ← PriorityQueue.empty compare
queue ← PriorityQueue.push queue $\{\Xi, \Phi_1, \Delta \vdash ?x :: T\}$
while *not ((PriorityQueue.top queue) satisfies all I/O-examples)* **do**
  if *not ((PriorityQueue.top queue) contains subterm from black list)* **then**
    successors ← successor (PriorityQueue.top queue)
    queue ← PriorityQueue.pop queue
    for *all s in successors* **do**
      queue ← PrioriryQueue.push queue s
    end
  else
    queue ← PriorityQueue.pop queue
  end
end
Output: PriorityQueue.top queue

Algorithm 6: Best first search with black list

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

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

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

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

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

Chapter 4

Implementation

What could I talk about in this chapter?

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

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 3. The chapter also puts our synthesis procedure in relation with the related work discussed in Chapter 2.

5.1 Experimental set up

TODO: Beschtreibung der Tabelle zur Tabelle anpassen

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

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

In the second experiment the synthesiser is used to automatically generate a *black list* that can be successively be used to prune the search space. We refer back to Section 3.5 for a description of pruning based on black lists. Section 5.1.2 describes how we used the synthesis procedure to generate a black list and Section 5.4 reviews the quality of the generated black list.

5.1.1 Evaluation on benchmarks

We evaluated 9 variants of our synthesis procedure, crossing the 3 exploration strategies with 3 of the cost functions described in Chapter 3. The three exploration strategies we evaluated are the following.

plain implements the basic synthesis procedure based on best first search described in Section 3.3.2.

blacklist implements the pruning of the search space based on a manually compiled black list provided in Table **??**. We refer to Section 3.5 for more details.

template implements the double best first search introduced in Section 3.6. 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 3 of the cost functions described in Section 3.4, 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 9 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 **??** and Table **??** list the benchmarks along with the specification size, that is the number of input-output examples we used to generate them, and the number of library components we provided to the synthesiser. The other columns report for each variant the running time, the ratio to the minimum running time for that benchmark and the size of the generated solution expressed in number of nodes.

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

In Table **??** all benchmarks except for `nth` share the same set of components, from which we took out the benchmark to synthesise, if it was one of the components. In Table **??**, in order to meet the need of all benchmarks, we used four different sets of 19 components.

Programs are enumerated only up to a timeout based on the number of programs that have been analysed so far. For the exploration strategies **plain** and **blacklist** the execution had been stopped after examining 2500000 programs (with or without holes). The exploration strategy **template** was restricted to generate templates with at most 2 higher-order components and at most 5 holes, the depth of the first-order search was limited to 10 calls to the plain procedure. For the cost function *nof-nodes* this corresponds to circa 4 min.

5.1.2 Automatic black list generation

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

We chose not to generate the polymorphic identity function. As during pruning we are ignoring types, holes and input variables, the programs that would have been generated for the polymorphic identity function are also generated for the identity over any specific type. Table **??** summarizes 30 typical programs.

5.2 Factors affecting runtime

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

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

5.2.1 Size of the solution

TODO: If you have time, redo the graphics in latex, or at least move trend line in the legend Figure 5.1 shows that the average running time for all nine variants of the synthesis procedure depends exponentially on the number of nodes. 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 the hole.

TODO: Delete this question or do something with it. We cannot answer it, we do not have enough data. More surprisingly, the individual results show that there must be some other factor influencing the runtime. Take, for example, `enumFromTo`, `stutter` and `nth`. All three of them have a solution with exactly 13 nodes, but their runtimes differ at least by an order of magnitude.

What does make `nth` generate in less than 1 s, `stutter` a hundred times slower and `enumFromTo` to time out in most of the cases?

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

5.2.2 Number of components

TODO: enumFromTo
Discuss that enumFromTo can be generated with 19 but not with 37 components even if we provide `enumTo`. That providing only the components it needs, `enumFromTo` can be synthesised pretty quickly also without providing `enumTo`.

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

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

Clearly, using as few and as small input-output examples as possible has a positive effect on performance. On the other side, too few and too general input-output examples can lead to the synthesis of the wrong program, that is a program that satisfies all provided input-output examples but that does not generalise in the expected way. This was especially a problem with `enumFromTo` and `member`. **TODO:** Put a concrete example with enumFromTo

5.2.4 Blacklist

The search space abounds of superfluous programs that are equivalent to smaller ones. In Section 3.5 we introduced a way to leverage this inconvenience: Pruning based on black lists. This approach allows us not to explore further programs that will surely lead to a solution bigger than the optimal one, like `append [X] (nil [X])`$?xs$, or not lead to a solution at all, like `(head [`$?X_1 \rightarrow ?X_2 \rightarrow X_3$`] (nil [`$?X_1 \rightarrow ?X_2 \rightarrow X_3$`]))` $?x_1$ $?x_2$.

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

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

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

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

5.2.5 Templates

5.2.6 Cost functions

5.2.7 Stack vs Queue expansion

As already mentioned in Section 3.3.1, we have two open questions in our best first search:

a. what program to expand next

b. which hole of this program to expand first

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

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

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

$(?x_0,\ [x_0]) \longrightarrow$
$(?x_1\ ?x_2,\ [?x_1,\ ?x_2]) \longrightarrow$
$(?x_3\ ?x_4\ ?x_2,\ [?x_3,\ ?x_4,\ ?x_2])) \longrightarrow$
$(\texttt{map [Int]}\ ?x_4\ ?x_2,\ [?x_4,\ ?x_2]) \longrightarrow$
$(\texttt{map [Int]}\ (?x_5\ ?x_6)\ ?x_2,\ [?x_5,\ ?x_6,\ ?x_2]) \longrightarrow$
$(\texttt{map [Int] (add}\ ?x_6)\ ?x_2,\ [?x_6,\ ?x_2]) \longrightarrow$
$(\texttt{map [Int] (add n)}\ ?x_2,\ [?x_2]) \longrightarrow$
$(\texttt{map [Int] (add n) xs},\ [])$

Left-to-right expansion often lead to faster synthesis, because leftmost holes have usually more constraints on their type. Consider the program $?x_3$ $?x_4$ $?x_2$ from the derivation of . We know more about $?x_3$ than about $?x_2$: 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 useless programs like $?x_3$ $?x_4$ `map` and `map` $(?x_5$ `foldr`) `xs`.

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

5.3 Interesting unexpected solutions

TODO: explain the thing with the different solutions
TODO: discuss interesting solutions, isEven, member
TODO: polymorphic equality

5.4 Automatic black list

We were able to automatically synthesise 300 programs corresponding to the identity function for three different types using automatically generated input-output examples **TODO:** time it. Because of their incremental nature, we need 6 to 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's about the ability to abstract and generalise over programs.

5.5 Comparison to related work

TODO: Put the table in LaTeX

—————————————- What I want to see in this chapter:

- Comparison to Feser, to Nadia, to Escher and to Myth (yes, we have their running times for some of the benchmarks. Say that you cannot really compare the numbers because they weren't run on the same machine).

- Explain why templates perform so poorly (hm... because they don't do BFS, that is they have to explore every branch to the "end"?)
- How number of components affects synthesis time
  - Short section explaining that the templates you generate are not the templates you expect and why I found one example, where templates help! For `dropmax` I had a run out of memory exception with plain enumeration and I could synthesise it in 7 seconds with templates! (Ok, I modified templates to put in only really higher-order components and close every hole, no matter the type). More resistant to the choice of examples.
  - In particular, not only the number but also the number of components of the same type. If you have more functions with the same type you will need much more time to find the program you are looking for (more possible successors).
- Talk about the constants in the cost functions and how they affect the search space. How cost functions influence the search space (`(head (^26 →^27 →Int)(nil (^26 →^27 →Int)))?2 ?3` has too complicated types appearing in the term, so it will have a higher cost in some of the cost functions. `foldnat foldnat foldnat mul (mul 3 3)3` takes a lot of time to evaluate, it will have a higher cost in no-same-component).
  - Idea behind nof-nodes: smaller programs generalize better to the examples
  - Idea behind nof-nodes-simple-types: programs with smaller types are less "useless" (example with `head nil`). Favor the use of input variables. But now a lot of "simple" but difficult to evaluate programs are synthesised, like the `enumTo prod enumTo prod`.
  - Idea behind no-same-component: there are components that are usually used only once in a program, like `foldr`, `foldNat`. Another thing is that we penalize also complicated types of the form `List (List (^5 →List ^4))` that had no additional cost in nof-nodes-simple-types. But now types overweight the number of nodes and simple programs like `foldr add zero _0` weight more than programs like `add mul sub prod enumTo....`
  - Idea behind no-same-component-bigger-constants: make all constants bigger so that we can differentiate more between the constants and so that nodes count more than types.

– Idea behind no-same-component-even-bigger-constants: see above. (well, then you failed. your constants for types are pretty much as big as those for terms...). Why it's bad? See above.

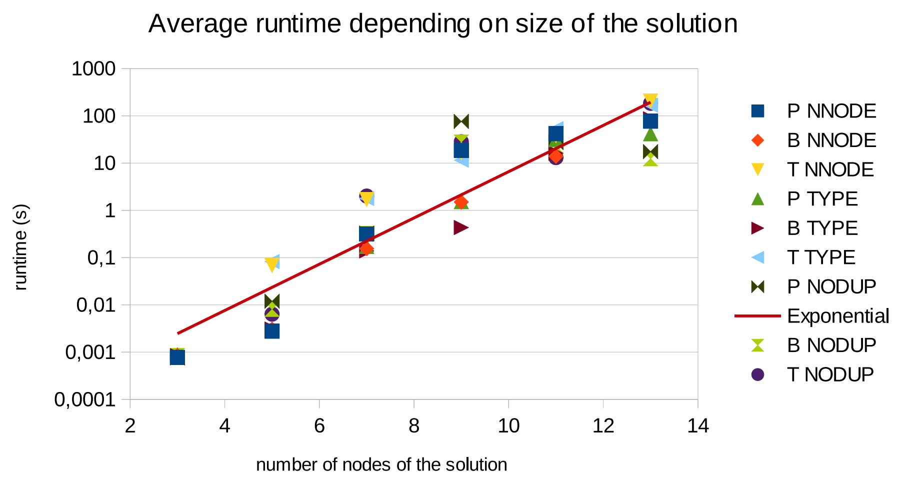

Figure 5.1: Average running time of the variants of the synthesis procedure depending on the number of nodes of the solution.

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] Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. Synthesis modulo recursive functions. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 407–426, New York, NY, USA, 2013. ACM.

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

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

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

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

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

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

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

Title of work (in block letters):

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

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

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

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

Place, date **Signature(s)**

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