Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Inductive Synthesis from Higher-Order Functions

Master Thesis

Alexandra Maximova

August 15, 2016

Advisors: Prof. Dr. Martin Vechev, Dimitar Dimitrov

Department of Computer Science, ETH Zürich

Abstract

TODO: write me :)

Contents

Chapter 1

Introduction

1.1 About program synthesis in general

put in some context, link 2

1.2 Problem definition

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

1.3 Contributions

Evaluation, exploring the baseline algorithm, exploring the search space

Chapter 2

Related Work

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

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

Nadia Polikarpova 2015
here is a talk: `http://research.microsoft.com/apps/video/default.aspx?id=255528&l=i`
and here is the code: `https://bitbucket.org/nadiapolikarpova/synquid`
In [6] Synquid is proposed. Refinement types (types decorated with logical predicates) are used to prune the search space. SMT-solvers are used to satisfy the logical predicates. The key is the new procedure for type inference (called modular refinement type reconstruction), which thank to its modularity scales better than other existing inference procedures for refinement types. Programs can therefore be type checked even before they are put together. Examples that this tool is able to synthesize include several sorting algorithms, binary-search tree manipulations, red-black tree rotation as well as other benchmarks also used by other tools (**TODO:** read about these benchmarks and write if there is something interesting). The user specifies the desired program by providing a goal refinement type.

Feser 2015
The tool proposed in [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 examples that require much more time to be synthesized than the others are dedup (remove duplicate elements from a list), droplast (drop the last element in a list), tconcat (insert a tree under each leaf of another tree), cprod (return the Cartesian product of a list of lists), dropmins (drop the smallest number in a list of lists), but all of them are synthesized under 7 minutes.

Kincaid 2013

In [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.

Osera 2015

The tool in [4] is called Myth and uses not only type information but also input-output examples to restrict the search space. The special data structure used to hold this information is the *refinement tree*. This system can synthesize higher-order functions, programs that use higher order functions and work with large algebraic data types.

There is an ML-like type system that incorporates input-output examples. Two pieces: a *refinement tree* and an enumerative search.

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

```
stutter :
```

Chapter 3

Top down type driven synthesis

TODO: starting words

In this chapter we will formally define the target programming language,

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

3.1 Problem

introductory example consider we want to generate

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

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

3.2 Calculus

The exposition will closely follow Pierce [5].

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.

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

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

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

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

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

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

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

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

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)

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

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

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

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

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

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

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

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

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

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

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

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 $\leftarrow$ 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**
    successors $\leftarrow$ successor (PriorityQueue.top queue)
    queue $\leftarrow$ PriorityQueue.pop queue
    for *all s in successors* **do**
        queue $\leftarrow$ 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.

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

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

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

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

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

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.

3.6 Templates

Top-down type-driven synthesis.

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

$\textit{nof-nodes-type}(X) = 5$
$\textit{nof-nodes-type}(?X) = 1$
$\textit{nof-nodes-type}(I) = 0$
$\textit{nof-nodes-type}(C\ [T_1, \ldots, T_k])=$
$\quad 1 + \textit{nof-nodes-type}(T_1) + \ldots + \textit{nof-nodes-type}(T_k)$
$\textit{nof-nodes-type}(T_1 \to T_2) = 3 + \textit{nof-nodes-type}(T_1) + \textit{nof-nodes-type}(T_2)$
$\textit{nof-nodes-type}(\forall X.\ T) = 5 + \textit{nof-nodes-type}(T)$

$\textit{nof-nodes-term}(c) = 1$
$\textit{nof-nodes-term}(?x) = 1$
$\textit{nof-nodes-term}(i) = 0$
$\textit{nof-nodes-term}(x) = 5$
$\textit{nof-nodes-term}(\lambda x : T.\ t) = 2 + \textit{nof-nodes-term}(t) + \textit{nof-nodes-type}(T)$
$\textit{nof-nodes-term}(t_1\ t_2) = 1 + \textit{nof-nodes-term}(t_1) + \textit{nof-nodes-term}(t_2)$
$\textit{nof-nodes-term}(\Lambda X.\ t) = 5 + \textit{nof-nodes-term}(t)$
$\textit{nof-nodes-term}(t\ [T]) = 1 + \textit{nof-nodes-term}(t) + \textit{nof-nodes-type}(T)$

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

$\textit{no-same-component}(t) = \textit{nof-nodes-term}(t) + 2\ \textit{count}(t)$

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

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.

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 6: Best first search with black list

3.6.1 Successor rules

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

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

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

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

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

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

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

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

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

Chapter 5

Evaluation

What I want to see in this chapter:

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

5.1 Set up

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

5.2 Cost functions

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

5.3 Black lists

In the presence of ”useless” functions with a high branching factor like flip, const or uncurry black list pruning is a must. But it can also be useful in other cases as well. The important thing is to find a reasonable trade off between the length of the black list (don’t forget that every subterm of every open program is matched against every item of the black list) and the degree of pruning. A longer black list prunes more of the search space, but the synthesis procedure also takes longer.

5.3.1 Benefits of black lists

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

5.3.2 Shortcomes of automatically generated black lists

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

5.4 Templates

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

It is very important to choose the examples well, because many generated programs are bad and will take forever to evaluate. And as we evaluate each closed program on the examples... We should really try to keep them small and simple. For example, I had a problem when I tried to generate dropmax using the example [1,4,3], because one of the generated programs was enumTo (prod (enumTo (prod _0))). It tried to construct a list with 479001600 elements and ran out of memory. At the same time, they should give enough information, otherwise a simpler program will be generated that, although it satisfies all given I/O-examples, it not what the user had in mind when writing the specification.

Chapter 6

Conclusions

6.1 Conclusions

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

6.2 Future Work

Templates done well, augmented examples?

Appendix A

Dummy Appendix

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

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] 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.

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

[6] 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.