ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Top-Down Inductive Synthesis with Higher-Order Functions

Master Thesis

Alexandra Maximova

August 22, 2016

Advisors: Prof. Dr. Martin Vechev, Dimitar Dimitrov
Department of Computer Science, ETH Zürich

Abstract

Library components encode well-known computational patterns that human programmers reuse in their programs. In this thesis, we investigate the benefits of providing a relatively large library of components to a program synthesiser. We target the synthesis of functional straight-line programs from input-output examples. Towards this end, we implement a basic synthesis algorithm based on best-first enumeration combined with type-based pruning. Heuristics are used to guide the search and black lists to prune the search space. We have evaluated our prototype implementation on simple algorithmic problems over a library of 37 components. Results indicate that the performance of our basic algorithm is comparable with that of much more sophisticated state-of-the-art algorithms.

Contents

Chapter 1

Introduction

In this thesis, we investigate a method for automatic program synthesis. Program synthesis strives to automatically synthesise programs given some sort of specification of their behaviour. For example, suppose we want to synthesise the function `replicate` that takes a number n and an element x of any type and returns the list $[x, \ldots, x]$ made of n copies of x.

We could specify the `replicate` function by providing a logical formula that completely describes the relation between inputs and outputs:

$$\forall n, x.\, \exists y.\ \texttt{replicate}\ n\ x = y \wedge (\text{length}(y) = n \wedge (\forall z \in y.\, z = x)).$$

Then, a logic-based program synthesiser would take such a formula and try to deduce an implementation of `replicate` that satisfies it.

Providing logical specifications, however, can be quite difficult, and we will be interested in a more natural alternative: specification by example. Here, the user has some function in mind and provides only a few input-output examples of its behaviour. From those few examples the synthesiser has to induce an implementation of the function that the user has in mind.

For our `replicate` function, we could give the following specification (in an Haskell-like syntax, which we will use through the thesis):

```
replicate 0 'A' = []
replicate 1 'A' = ['A']
replicate 2 'A' = ['A', 'A']
replicate 3 'A' = ['A', 'A', 'A']
```

The synthesiser then has to "guess", or rather, extrapolate the behaviour in the examples to unseen inputs. This guess could for example use recursion:

```
replicate 0 x = []
replicate n x = x:(replicate (n - 1) x)
```

However, synthesising a recursive definition directly is a rather difficult and unstructured problem: there are too many choices of how to invoke the function recursively. A human programmer rarely attacks such problems directly, but relies on prior knowledge of restricted computational patterns. Moreover, well-known patterns are widely implemented as reusable components (subprograms) and grouped into libraries.

For example, the computational pattern of replacing every element x in a list according to a map $x \mapsto \texttt{f}\ x$ is captured by the higher-order function `map`:

$$[x_1, \ldots, x_n] \xmapsto{\texttt{map f}} [\texttt{f}\ x_1, \ldots, \texttt{f}\ x_n].$$

Another, more general, pattern is the accumulation of the values of a list along a binary operation `f` (`` `f` `` in infix), as captured by `foldr`:

$$[x_1, \ldots, x_n] \xmapsto{\texttt{foldr f}\ x_{n+1}} x_1\ \texttt{`f`}\ (x_2\ \texttt{`f`}\ \ldots\ (x_n\ \texttt{`f`}\ x_{n+1})\ldots).$$

To illustrate how computational patterns are reused by humans, consider how to implement the `replicate` function without explicit recursion:

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

Here, `const` takes two arguments and always returns the first one; `enumTo` takes an integer n and returns the list $[1, 2, \ldots, n]$. The solution encodes the insight: to generate a list consisting of n copies of x, one can first generate an any list of length n, and then map each element to the given x. This leads us to the question: can a synthesiser produce such "insightful" programs?

1.1 Problem statement

The main goal of the thesis is to investigate whether the computational knowledge encoded in components can help an automatic synthesiser, in analogy of how a human reuses known patterns in order to solve such tasks. In particular, we investigate whether one can quickly synthesise solutions to basic algorithmic tasks in terms of standard library components. Our main hypothesis is that standard components capture widely useful patterns that one can use to find short and simple solution programs.

Towards this goal, we develop an example-based synthesis algorithm, not tailored to a specific set of components. This allows one to easily expand the "computational knowledge" of the synthesiser simply by adding new components. To simplify the matter, we focus on the synthesis of purely functional programs without any lambda expressions, explicit recursion or conditionals (i.e., only function application is allowed). This restriction is non-essential as richer constructs can be easily simulated by applications of suitable higher-order components. More importantly, it allows for a clean synthesis algorithm that focuses on how to combine components effectively.

1.2 Existing work

The past fifty years of research approached program synthesis from various points of view. For example, early methods [14] were based on automatic theorem proving and the relation between proofs and programs (the Curry–Howard correspondence). They convert a logical specification of a program into a theorem, find a proof of this theorem, and then extract a program from the proof. Such methods are mainly limited by the performance of the employed theorem prover (which, unfortunately, is not very high at present). A drawback of these approaches is that a high level of mathematical maturity is required to provide the needed logical specification.

A more accessible way to specify a program is by giving a finite number of input-output examples that the synthesised program must satisfy. In this setting, two main approaches are most popular. The first one analyses the input-output examples to derive a set of relations that capture them, and then transforms this description into a program [22, 11, 10]. Even though the techniques are quite interesting, they still need considerable advancement. The second approach is based on program enumeration [4, 2, 16], and has become more popular as processor speed increased.

Good results for the second approach were obtained by restricting the target programs to a specific domain [6, 7]. Recently a generic system for synthesis in domain-specific languages [17] was presented. This system provides a lot of flexibility to make the synthesis more tractable by employing more restrictive languages. Another way to restrict the problem is to require additional clues from the user, as in Sketch [21], where the user sketches the high-level structure of the program, while the synthesiser fills in the low-level details. To achieve efficiency, Sketch combines program enumeration with constraint-satisfaction, for which it uses off-the-shelf SAT/SMT solvers.

Recently, type theory and program verification also entered the synthesis scene [12, 9, 13, 5]. Systems like [4, 16, 19] actively use type signatures to prune the search space. This is quite reasonable, as type information is easy to provide and usually readily available. Also, type-based pruning is essential in our setting, because a large number of library components can often be combined in a small number of ways due to typing. That is why our synthesis algorithm also falls into the type-based pruning category.

1.3 Contributions

In this thesis we study whether program synthesis can benefit from access to common computational patterns. In particular, we investigate whether a library (of first and higher-order components) can guide and speed up the synthesis process. Towards this end we make the following contributions:

1. We develop a basic synthesis algorithm for purely functional programs based on exhaustive enumeration combined with type-based pruning.
2. We propose a couple of heuristics to guide the search in a best-first manner, and also how to automatically blacklist certain useless branches.
3. We develop Tamandu, a prototype implementation of our algorithm in OCaml, and perform an extensive evaluation over a library of 37 components.
4. We analyse the results of the evaluation and compare it with reported results from state-of-the-art program synthesis tools.

Our evaluation indicates that for simple algorithmic problems, our basic algorithm combined with a good enough library of components performs on par with much more sophisticated state-of-the-art algorithms.

Chapter 2

A type-driven synthesis procedure

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

2.1 The 'replicate' example

Let us illustrate our synthesis procedure with the 'replicate' example from Chapter 1. We want to synthesise a program that takes a number `n` and an element `x`, and returns the list `[x, ..., x]` that consists of `n` copies of `x`. A typical synthesis task would be specified like this:

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

Here, the user specifies the input-output signature of the program (its type), and a few input-output examples of its intended behaviour. In this case, the user desires that the program can replicate elements of any type `X`, hence the type signature is prefixed with the `∀ X` quantifier. The ability to work with any type of elements is made evident by the two examples presented after the type signature. In the first one, the number 1 gets replicated three times; in the second one, the empty list `[]` gets replicated twice.

In order to solve the synthesis task, we look for a program composed of components from a user-specified *library of components*. Let us assume that in our case the library consists of the standard list combinators `map` and `foldr`. In addition, we include the `enumTo` function that returns a list with the numbers from 1 up to its argument, and also the `const` function that always

returns its first argument. We will also need the standard list constructors `cons` and `[]`, and the standard integer constructors `succ` and `0`.

We search for the goal program by enumerating plausible programs, starting from simpler ones and moving to more complex ones (c.f., Occam's razor). In the process, we test whether the enumerated programs meet the specified input-output behaviours. Once we find such a program, we give it to the user for approval. The main difficulty with this approach is that the number of candidate programs of a given size grows exponentially, but, on the other hand, very few of them make sense for the synthesis task. For example, the program `replicate n x = n` returns the integer `n` instead of a list of elements, contradicting the specified type signature of 'replicate'.

We address this problem by enumerating only well-typed programs that meet the user-specified type signature. We do that by actually enumerating *partial programs*. Partial programs are just programs with *holes* that are to be filled in later. Every hole has a type that *drives* subsequent search steps: they fill the hole with a partial program matching its type. In our particular example, the search would start with the partial program:

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

The hole `?p` is treated as a typed fresh symbol that has to be filled in with another partial program. Thus, our task is to find an instantiation for all holes recursively until we end up with a *closed* program (all holes filled in) that satisfies all the input-output examples provided by the user.

We structure the recursive enumeration process as a best-first search. We maintain a set of current partial programs, the *frontier*, whose holes are to be expanded. At each step we select and remove a partial program of minimum *cost* (e.g., size) from the frontier. Then, we fill in one of the holes in the selected program according to a set of rules. The rules produce a set of *successors* that are then added to the frontier. The search continues until we select a closed program that meets all the given input-output examples.

The tricky part in the search are the rules that determine the successors of the selected program. The rules must ensure that only well-typed programs are enumerated. For example, in the above partial program we expand the lone hole `?p`. This hole has type `List X` and we need to find a component in the library that is of this type. This immediately excludes the integer constructors `succ` and `0`. However, it also excludes `enumTo`, as it produces a list of integers, while we need a list of `X`, and `X` could be any type, not necessarily `Int`. All other possibilities are open: we can fill `?p` with the result of a `fold`, a `map`, a `const`, or with the empty list `[]`. Let us begin with `[]`:

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

Since this program is closed, we can test it on the input-output examples, but it satisfies none of them. Therefore, we consider the other four possible instantiations of `?p` as well:

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

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

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

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

In these programs, `?Y` is a fresh type variable that will be instantiated later. It indicates that we do not know the type of the first argument of `?f`. The only thing we know is that it has to match the type of the elements of `?xs`.

The next step in the procedure is to expand some hole in the least-cost program. In Section 2.4 we discuss several choices for a cost function, but here we just select programs that will lead us to the solution. Let us select the last program for expansion, i.e., `map ?f ?xs`. We have two holes to fill-in: a function `?f` that takes a `?Y` and returns an `X`, and a list `?xs` of `?Y`. We decide to expand the hole `?f` first. Obviously, for that we cannot use `map` or `enumTo`, because they return lists, whereas `?f` must return an `X`. The other two possibilities are `foldr` and `const`, which we add the to the frontier:

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

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

Note that in the first case we instantiated `?Y` with `List ?Z`, because `foldr` takes a list as its last argument. This indicates that type-instantiation is a non-trivial task. In particular, it requires solving *unification* constraints between arguments and return values, as discussed in Section 2.2.5.

Now, we need to expand a program in the frontier again. Let us assume that the least-cost program is `map (const ?x) ?xs`. There are two holes to fill in: `?x` of type `X` and `?xs` of type `List ?Y`. Interestingly, we have only one option for the first hole, and so we take it: `replicate`'s second argument `x`:

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

We select this newly added program from the frontier, and fill in its only hole `?xs`. We are now in a situation similar to where the whole search started: we have to generate a list. However, this time the type of the elements is not fixed, and we cannot rule out `enumTo`. Therefore we have a lot of possibilities to instantiate this hole, starting with `[]` and ending with `enumTo ?n` where `?n` is a fresh hole of type `Int`.

One of the candidate programs, `map (const x) []`, is closed, and we evaluate it on the input-output examples. However, this program does not satisfy any of them, thus we rule it out. Several candidate solutions are added to the frontier at this step, but let us focus on the most promising one:

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

The only hole to expand is `?n` and has type `Int`. Again, we have a lot of possibilities: the number `0`, `replicate`'s first argument `n`, the constructor `succ` applied to another integer hole, or the result of invoking `const`. We again look at the closed successors, namely the first two:

```
replicate n x = map (const x) (enumTo 0)

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

The first successor does not satisfy the input-output examples. On the other hand, the second one does, and we finally arrived at a solution. Further inspection by the user shows that this program indeed captures the desired 'replicate' functionality. Our synthesis task is complete.

2.1.1 Superfluous instantiations

The many search choices that we left unexplored indicate that the search space remains quite big even if with the type-directed search that rules out ill-typed programs. That is why, in addition to the type-driven search, we include another mechanism for pruning the search space.

This mechanism (Section 2.5) blacklists *superfluous* hole instantiations that have the same or similar input-output behaviour to instantiations of lower cost. For example, let's go back to the initial partial program:

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

Recall that one possible way to instantiate `?p` is with the partial program `const ?xs ?s`. Then, we would need to instantiate `?xs` as well. But in this case we could have instantiated `?p` the same way and get exactly the same effect. Thus, we conclude that `const ?xs ?s` is superfluous and so we can safely skip it during search.

2.1.2 Summary

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

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

Successor a program derived by filling in a hole of a partial program. In Section 2.3.1 we can find the rules according to which holes are expanded.

Best-first search a frontier of programs with holes is maintained, one hole of the least-cost program is expanded in every iteration. The best-first search algorithm is defined in Section 2.3.2 and several cost functions are presented in Section 2.4.

Superfluous program a program that is equivalent to a shorter program.

2.2 Calculus

The programs targeted by our synthesiser are based on System F terms restricted to applications of library components and input variables. As we saw in the previous example, during synthesis an additional syntactic construct is used: the hole. We denote the language based on application of library components, input variables and holes as the *target language* of

our synthesiser. However, we also need another language: the language in which the library components are defined and the programs are evaluated on the input-output examples. In the rest of the chapter we will call this language the *internal language*. It extends System F with holes, parametric types and recursion.

We present therefore in Section 2.2.1 the syntax of the following three calculi:

1. System F, which we provide for the sake of completeness, since the other two calculi build upon it;
2. The internal language: an extension of System F with holes, input variables, library components, parametric types and recursive terms and types;
3. The target language: a subset of the internal language, featuring only application of components, holes and input variables.

The notation and the exposition closely follow the excellent book on type systems by Benjamin Pierce [18]. We refer to the book for a thorough introduction to System F and to type systems in general.

2.2.1 Terms and Types

This section presents the syntax of three different calculi: System F, the internal language and the target language.

System F System F, also known as the polymorphic lambda calculus, is a calculus that, additionally to term abstraction and term application, features two new kinds of terms: type abstraction `ΛX. t` and type application `t [T]`. This allows us to express polymorphic functions. For example, the polymorphic identity function is defined as `ΛX. λx:X. x`. Polymorphic functions, defined as type abstractions, have a special type: the *universal* type `∀X. T`. For a more detailed introduction to System F we refer to [18]. The syntax is summarized below.

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

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

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

Internal language We extend System F with holes $?x$, input variables i as well as named library components c and named types C that can take parameters $C\ T_1\ \ldots\ T_K$. The number of type parameters supported by a named type is denoted as K in its definition. The use of the names enables

recursion in the definition of library components and types. Terms that do not contain holes are called *closed*. The syntax of our calculus is summarised below. Evaluation and typing rules for this calculus can be found in the respective subsections.

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

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

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

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

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

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

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

Target language The target language of our synthesiser is a subset of the internal language. We already saw terms of this language in the 'replicate' example, but there all type applications were omitted for the sake of clarity. Formally, the target language restricts the internal language to term and type application of library components, input variables and holes as follows.

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

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

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

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

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

We do not need separate typing rules for this sublanguage of the internal language, since it is closed under the typing rules. We omit evaluation rules for this calculus, as our evaluator is based on the internal language.

Program A program is defined as the 4-tuple $\{\Xi, \Phi, \Delta \vdash t :: T\}$, where t is a term of the target language. A program is called *closed* if Ξ is empty and t and T do not contain holes.

2.2.2 Encodings

Familiar types such as booleans, integers or lists do not appear in the definition of the types of the internal language. All these types can be encoded in the type system of the internal language using either Church's or Scott's encoding [1]. We opt for Scott's encoding because it is more efficient in our case.

Scott's booleans coincide with Church's booleans and are encoded as follows.

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

Scott's integers differ from Church's integers as they unwrap the constructor only once. Therefore they are more suitable for pattern matching.

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

Analogously to integers, 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
```

Other algebraic datatypes, such as trees, can be easily encoded in an analogous manner.

2.2.3 Evaluation semantics

In this section we present the evaluation semantics of our internal language, that is the second calculus introduced in Section 2.2.1. The evaluation semantics is a standard eager evaluation and we refer to the excellent book by Benjamin Pierce about type systems [18] for an introduction to the evaluation semantics of System F and evaluation rules in general.

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

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

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

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

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

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

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

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

2.2.4 Type checking

In this section we will present the typing rules of the internal language, that is the second calculus presented in Section 2.2.1. The typing judgement $\Gamma, \Xi, \Phi, \Delta \vdash t : T$ means the term t has type T in the contexts Γ and Ξ, binding respectively variables and holes, and Φ and Δ, containing signatures and definitions of respectively input variables and library components. The typing judgement is similar to the typing judgement of System F. As usual, we refer to [18] for more details.

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

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

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

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

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

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

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

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

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

2.2.5 Type unification

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

However, unification on the type system of System F is undecidable [8]. Therefore we choose to restrict our type system to quantifier-free forms. This allows us to completely ignore universal types during unification. Nonetheless, we still want to handle universally quantified library components, that is components whose type has the form $\forall X_1$ `.` $\ldots$ $\forall X_n$ `.` `T(`$X_1, \ldots, X_n$`)` with a quantifier-free `T(`$X_1, \ldots, X_n$`)`. Towards this end, we represent types of the form $\forall X_1$ `.` $\ldots$ $\forall X_n$ `.` `T(`$X_1, \ldots, X_n$`)` as `T(?`X_1`, ..., ?`X_n`)`, that is we leave out all quantifiers and replace the bound variables with fresh type holes.

The goal of unification is to find a substitution σ that unifies all pairs of types of a set of constraints. A set of constraints $\mathcal{C}$ is a set containing pairs of types (S, T) that should be equal under a *substitution* (a mapping from holes to types). That is, for the output of the unification algorithm σ it must hold $\sigma(S) = \sigma(T)$ for every constraint (S, T) in $\mathcal{C}$.

Our unification algorithm (summarised as Algorithm 1 below) is based on the unification algorithm for typed lambda calculus from [18] and slightly modified to fit our needs.

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

2.3 Search

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

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

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

Algorithm 1: Type unification

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

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

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

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

2.3.1 Search space

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

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

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

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

The derive judgement can be summarised in four rules. D-VarLib replaces a hole $?x$ with a type application of a library component c to suitable types, if the type of c unifies with the type of $?x$. D-VarInp replaces a hole $?x$ with an input variable i from the context Φ, if its type unifies with the type of the hole. D-VarApp turns a hole into a term application of two fresh holes. The rule D-App chooses a hole in the program and expands it according to one of the three rules above.

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

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

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

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

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

All derived programs are well-typed and, since the types of all derived programs unify with the types of their ancestors, have the desired type.

2.3.2 Best-first search

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

Algorithm 2 is parametrised with respect to the following two questions.

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

The first question is addressed by the implementation of the `compare` function over programs. We compare programs based on their *cost*. Section 2.4 presents several cost functions.

The second question is addressed by the implementation of the `successor` function. In particular, the implementation of the D-App rule. There are two easy ways to handle this situation. The first way is to always expand the leftmost hole, the second way is to always expand the oldest hole. Section 4.5.4 discusses the advantages and disadvantages of these two choices.

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 {Ξ, Φ1, Δ ⊢ ?x :: T}
while not ((PriorityQueue.top queue) satisfies all I/O-examples) do
    successors ← successor (PriorityQueue.top queue)
    queue ← PriorityQueue.pop queue
    for all s in successors do
        queue ← PrioriryQueue.push queue s
    end
end
return PriorityQueue.top queue
```

Algorithm 2: Best first search

2.4 Cost functions

The compare function in the best-first search algorithm can be defined as `cost` p_1 `-` `cost` p_2. There are different possibilities to define this cost function. We will present four alternatives. All of them are based on the idea that

shorter and simpler programs generalise better to unseen examples, along the lines of the Occam's razor principle [15]. The first three alternatives are evaluated on benchmarks and their effect on performance is discussed in Section 4.5.3.

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

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

nof-nodes-simple-type The second cost function adds a factor based on the size of the types appearing in the term. It penalises thus terms with type application depending on the size of the applied types. In particular, arrow types appearing in type applications are heavily penalised. The cost function over types is inductively defined over the quantifier-free subset of the types of the target language.

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

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

$$\textit{nof-nodes-simple-type}(t) = \textit{nof-nodes-term}(t)$$

no-same-component The third cost function additionally penalises terms that use the same component more than once.

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

$$\textit{nof-nodes-type}(T_1 \to T_2) = 5 + \textit{nof-nodes-type}(T_1) + \textit{nof-nodes-type}(T_2)$$

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

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

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

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

2.5 Black list

Recall the 'replicate' example, where we encountered the superfluous program `const [?X] ?`x_1 `?`x_2. The best-first enumeration explores many superfluous branches, such as, for example:

```
foldr [?X] [List ?X] (cons [?X]) (nil [?X]) ?xs

add zero ?n
```

Such programs can be ruled out only based on the semantics of the library components. A simple way to prune those superfluous branches is to compile a list of undesired patterns and check each generated program against this list. This is what we call *black list pruning*.

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

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

$$\begin{aligned}
&\textit{matches}(?x, t)\\
&\textit{matches}(i, t)\\
&\textit{matches}(c, c)\\
&\textit{matches}(t_1\ t_2, t_3\ t_4)\ \ \textbf{if}\ \ \textit{matches}(t_1, t_3)\ \ \textbf{and}\ \ \textit{matches}(t_2, t_4)\\
&\textit{matches}(t_1\ [T_1], t_2\ [T_1])\ \ \textbf{if}\ \ \textit{matches}(t_1, t_2)
\end{aligned}$$

As you can see, holes and input variables in the black list match every subterm, a library component matches only itself, a term application matches a term application whose respective left- and right hand sides match and a type application matches a type application if the left hand sides match. Note that the types in a type application are completely ignored. As a reminder of this semantics, in the following chapters we will typeset black list patterns omitting the type applications and replacing all holes and input variables with the wild-card symbol '_'.

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

Input: goal type T, library components Δ, list of input-output examples $[(\Phi_1, o_1), \ldots, (\Phi_N, o_N)]$, black list $[b_1, \ldots, b_M]$
queue ← 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 3: Best first search with black list

In Section 4.4 we discuss how to synthesise a black list automatically.

2.6 Templates

In this section we present a slightly different way to explore the search space. The idea is to fix all higher-order components first, producing a *template* for a program, and then fill in the remaining holes with input variables and first-order components. Since programs usually contain only a few higher-order components, the enumeration of templates takes less time than the enumeration of programs of comparable size. Moreover, templates encode well-known patterns of computation and impose meaningful constraints on the remaining holes. Therefore, it should be easy to find the desired program starting from the right template. This allows us to quickly abandon tem-

plates if no program satisfying the input-output examples is found within a short timeout.

The library Δ is split into two contexts: Δ_h, containing all higher-order components, and Δ_f, containing the first-order ones. We also need to introduce a new kind of term: the *delayed hole* $\underline{?x}$. This is a hole, whose instantiation is delayed to the first-order search. The context Ξ binds, additionally to normal holes, delayed holes as well. A *template* is a term in the target language that may contain delayed holes but does not contain input variables. A template is called *closed* if it does not contain holes (it may, however, contain delayed holes).

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

We additionally restrict the space by requiring a template to have no more than M higher-order components and no more than P delayed holes. Templates are enumerated according to the rules listed below. Those are very similar to the ones defined in Section 2.3.1, except that we cannot instantiate a hole with an input variable but we can delay a hole. All the rules are modified to take into account the restriction on the number of components and the number of closed holes. In order to do this, we need to pass along m, the number of higher-order components in the term whose subterms we are traversing.

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

$$\frac{\begin{array}{c}|\Xi| \leq P \text{ and } m < M \\ c : \forall X_1.\ \ldots \forall X_n.\ T_c(X_1, \ldots, X_n) \in \Delta_h \\ ?X_1, \ldots, ?X_n \text{ are fresh type holes} \\ \sigma \text{ unifies } T \text{ with } T_c(?X_1, \ldots, ?X_n) \\ \Xi' = \Xi \cup \{?X_1, \ldots, ?X_n\} \setminus \{?x : T\}\end{array}}{\Xi, \Phi, \Delta_h, m \vdash ?x :: T \mapsto \sigma(\Xi'), \Phi, \Delta_h, m+1 \vdash c\ [\sigma(?X_1)]\ \ldots\ [\sigma(?X_n)] :: \sigma(T)}\ \text{G-VarLib}$$

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

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

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

Chapter 3

Related Work

In this chapter we look at four state-of-the-art tools closely related to our work. They all synthesise functional programs, use type information to restrict the search space and enumerate terms during search. Since simple types are too ambiguous to specify a program, the tools we present either choose to complement type information with input-output examples, as in [4, 2, 16], or resort to more complex and more expressive types that can actually act as a specification, as for example the *refinement types* in [19].

3.1 SYNQUID

In [19] SYNQUID is proposed. The code can be found online[1] and there is the possibility to try it in the browser. This tool uses *refinement types* (types decorated with logical predicates) to prune the search space and to specify programs. SMT-solvers are used to satisfy the logical predicates appearing in the types.

Refinement types were already successfully used for verification. In particular, the tool builds upon the liquid types framework [20]. However, it proposes a new procedure for type inference (called modular refinement type reconstruction), which thank to its modularity scales better than other existing inference procedures for refinement types. In contrast to existing procedures, where only complete programs could be type checked, the new procedure introduced in this paper allows one to type check partial programs as they are generated and use the type information to guide the synthesis.

This tool targets a language that includes lambda expressions, pattern matching, structural recursion, conditionals and fixpoint. The user can define cus-

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

Table 3.1: Comparison of program synthesis tools closely related to our tool Tamandu

	Type of search	Specification	Target language	Components
Synquid	type-directed enumeration combined with deduction	refinement types	typed lambda-calculus with algebraic data types and recursion	user-defined types and components, only refinement type of components must be provided
λ^2	type-directed enumeration with templates combined with manual deduction rules	input-output examples	typed lambda-calculus with algebraic data types and recursion	7 fixed higher-order components, user-defined first-order components
Escher	enumeration and conditional inference	input-output examples	untyped application of library components, input variables and the function being synthesised	user-defined first-order components
Myth	type and example-directed enumeration	input-output examples and type signature	typed lambda calculus with algebraic data types and recursion (no polymorphic types)	user-defined components and data types (no polymorphic types)
Tamandu	type-directed enumeration	input-output examples and type signature	typed application of library components and input variables	user-defined types and components

tom functions and inductive datatypes that can be passed to the synthesiser as components.

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

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

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

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

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

In contrast to our tool, the number of components provided to the synthesiser for evaluation is small. Moreover, as already mentioned earlier, it is not always easy to specify a synthesis task as a refinement type. On the contrary, in our tool the behaviour of the desired program is specified in a more intuitive way with input-output examples. Typically very few of them are required.

3.2 λ^2

The tool proposed in [4] is called λ^2 and generates its output in λ-calculus with algebraic types and recursion. The target language also includes 7 higher-order combinators such as `map`, `fold` and `filter` and a flexible set of primitive operators and constants.

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

The synthesis algorithm is a combination of inductive generalisation, a limited form of deduction and enumerative search. First, it generates *hypotheses* in a type-aware manner, that is programs with free variables such as:

```
λx. map ?f x
```

where `?f` is a placeholder for an unknown program to be synthesised. Then deduction in form of hand-coded rules about the higher-order combinators is used either to refute a hypothesis or infer new input-output examples to guide the synthesis of missing functions. For example, the hypothesis `λx. map ?f x` will be refuted if the length of the input list does not match the length of the output list. Enumerative search is used to enumerate candidate programs to fill in the missing parts of hypotheses. Hypotheses and candidate programs are organised in a priority queue and, at each point of the search, the least-cost candidate is picked.

This tool is able to synthesize programs manipulating recursive data structures like lists, trees and nested data structures such as lists of lists and trees of lists. It synthesises all benchmark programs in under 7 minutes. Half of the benchmarks are synthesised in under 0.43 s. However, the synthesis of `droplast`, the program that drops the last element of a list, takes up to 320 s. The program that removes duplicates from a list, the program that drops the smallest elements of each list of a list of lists and the program that inserts a tree under each leaf of another tree also take more than 100 s to synthesise.

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

3.3 Escher

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

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

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

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

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

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

The search is goal-directed. Programs are associated with value vectors, that is the vector of the outputs of the program on the inputs from the input-output examples. Programs sharing the same value vectors are considered equivalent, that is the search space is pruned based on observational equivalence. The algorithm alternates between two phases: forward search and conditional inference. During forward search programs are inductively enumerated by adding new components to already synthesised programs. During conditional inference a novel data structure, the *goal graph*, is used to detect when two synthesised programs can be joined by a conditional statement. The alternation between the two phases is guided by a heuristic.

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

Like our work, this tool can handle a flexible set of components. For example, a set of 23 components was used to evaluate all benchmarks. However, there were no higher-order components among them. In contrast to our tool, ESCHER requires a closed set of input-output examples. This is not always convenient for the user, as they have to anticipate the recursion in the solution.

3.4 MYTH

The tool proposed in [16] is called MYTH and targets a subset of OCaml including pattern matching, recursive functions, higher-order functions and algebraic datatypes. The prototype lacks support for polymorphic and product types, but can be extended to handle them as well.

The user specifies the desired program by providing a type signature and several input-output examples. Like in ESCHER, in order to generate recursive functions, the set of input-output examples must be *closed*. Unlike ESCHER, the user can easily extend the set of library components with their own higher-order functions and define their own algebraic datatypes.

The technique is inspired by proof search and searches for a term in the target language that has a valid typing derivation. The typing rules are

modified to incorporate input-output examples and to push them towards the leaves of the derivation tree. That is, the information contained in the input-output examples is used to prune the search space even before a program that can be evaluated on the examples is generated. The tool also makes use of well-typed term enumeration to guess a term of the goal type that is consistent with the examples. Since enumeration is used to fill in the branches of a pattern matching or the arguments to a constructor, the size of enumerated term is considerably smaller than the size of the program being generated.

This tool is able to generate structurally recursive programs that use pattern matching. The benchmarks were evaluated in the presence of a relatively large set of library components, however, the precise size of this set is not provided. The benchmarks include programs over booleans, natural numbers, lists and trees. The tool is able to synthesise all benchmark programs in less than 2.5 min, 70% of them are synthesised in less than 0.4 s. The only benchmark that takes more than 1 min to synthesise is `list_compress`.

In contrast to our tool, Myth requires a closed set of input-output examples as a specification to a synthesis task. This is an inconvenience, since the user has to anticipate the structure of the program and the recursive calls. Moreover, as opposed to our tool, Myth lacks support for polymorphic types.

Chapter 4

Evaluation

The main goal of this chapter is to give some insights about the factors that affect performance and compare different variants of the synthesis procedure described in Chapter 2. We also compare our synthesis procedure to the related work discussed in Chapter 3.

4.1 Implementation

We implemented our prototype Tamandu in OCaml 4.03.0, closely following the formalisation in Chapter 2. For our internal language, we implemented an interpreter, a type checker and unification. The types and the terms are represented by algebraic data types. For ease of implementation, we use De Bruijn indices [3] to represent bound variables in term and type abstractions. For convenience, we implemented a concrete syntax for the internal language and used it to implement most library components. A parser for this language was generated using the `menhir` package. In order to speed up the evaluation, we extended the interpreter with an interface for implementing functions in OCaml. This way, we implemented arithmetic via OCaml integers and built-in functions.

As already discussed in Section 2.2.1, the internal language supports recursion by name. In evaluation and type checking we use maps from library components to their (possibly recursive) definitions to unfold the definitions when needed. For example, to check type equality during type checking we sometimes have to unfold recursive type definitions.

Unification is implemented along the lines of the unification algorithm presented in Section 2.2.5. Constraint sets are represented by lists of pairs and substitutions by hashmaps mapping holes to types. Since we only run small instances, the overhead of not using a faster data structure like union-find is negligible.

During synthesis, we represent partial programs as maps from holes to terms (that possibly contain holes as well). Every call to the successor function adds a mapping for the current hole.

4.2 Experimental set up

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

The goal of the first experiment is to assess the quality and the performance of the synthesiser on standard benchmarks. The detailed set up is described in Section 4.3 and the results are discussed in Section 4.3.1. Section 4.5 examines the factors that affect the runtime.

In the second experiment the synthesiser is used to automatically generate a *black list* that can be used to successively prune the search space. We refer back to Section 2.5 for a description of pruning based on black lists. Section 4.4 describes how we used the synthesis procedure to generate a black list and Section 4.4.1 reviews the quality of the generated black list.

All experiments were run on an Intel quad-core at 3.2 GHz with 16 GB RAM. Since the code is sequential, the performance could not benefit from the additional 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.

4.3 Performance evaluation

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

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

Blacklist implements the pruning of the search space based on a manually compiled black list (provided in Table 4.4). We refer to Section 2.5 for more details.

Template implements the double best-first search introduced in Section 2.6. As you probably recall, the procedure first looks for a *template* featuring at most M higher-order components and at most P holes. As soon such a template is found, the procedure falls back on the Plain variant where only the first-order components are provided and explores the search space up to a certain depth.

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

We exercised the nine different variants of our synthesis procedure on a benchmark of 23 programs over lists, mostly taken from related work or standard functional programming assignments.

Table 4.1 summarises the running times. Cells are organised as 3×3-squares. Every number in such a square corresponds to one of the variants of our synthesis procedure. The first column summarises the running times of the nine variants of our synthesis procedure when the synthesiser is given 36 to 37 components. The second column contains, for each variant, the slowdown with respect to the minimum running time for the respective benchmark. The last column shows the speedup we obtain if we leave only 18 to 19 components in the library.

Table 4.2 lists the benchmarks along with the size of the solution generated by each of the nine variants, expressed in the number of nodes.

Components In the first column of Table 4.1 all benchmarks except for `nth` share the same set of components (listed in Table 4.3). For the synthesis of individual benchmarks appearing in the library as components we took the corresponding component out of the library. In the third column of Table 4.1, in order to meet the needs of all benchmarks, we used four different sets of 19 components.

Timeout Programs are enumerated only up to a timeout based on the number of partial programs synthesised so far. For the exploration strategies Plain and BlackList the execution was 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 approximately 4 min.

4.3.1 Results

Two variants of our synthesis procedure were able to synthesise all 23 benchmarks in the presence of 36 to 37 library components, all variants synthesised at least 18 benchmark programs within the time limit. 78% of the benchmarks were synthesised within 1 s using BlackList as the exploration strategy and *nof-nodes-simple-type* as the cost function.

The variants that use the BlackList exploration strategy can synthesise the most number of benchmarks: for two cost functions they generate all 23 benchmark programs within the time limit, for the cost function *no-same-component* they fail to synthesise `enumFromTo`. They synthesise half of the benchmarks in under 0.2 s on average and 17 benchmark programs in under 1 s.

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

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

Table 4.1: Runtime of nine variants of our synthesis procedure on 23 benchmarks. Each cell is organised in a 3×3 square. The rows of the square are labelled with the exploration strategies and the columns of the square are labelled with the cost functions. The first column shows the runtimes in seconds when 36 to 37 components are provided. The j-th entry in the i-th square of the second column is the ratio of the j-th entry and the minimum of the i-th square of the first column. The third column shows the speedup with respect to the first column if we provide only 18-19 library components.

Name	M.	Time			*Vs.* 37-group			*Vs.* 19-self		
		Nodes	Types	NoDup	Nodes	Types	NoDup	Nodes	Types	NoDup
`append`	P	0.32	0.10	0.07	4.98	1.51	1.08	5.34	3.29	2.53
	B	0.08	0.07	0.07	1.27	1.00	1.00	2.55	2.64	2.13
	T	1.90	2.63	2.44	29.23	40.36	37.46	2.01	3.47	2.25
`concat`	P	0.19	0.04	0.24	5.01	1.16	6.39	3.08	1.50	2.29
	B	0.04	0.04	0.19	1.01	1.00	5.02	1.42	1.38	1.80
	T	1.86	2.38	2.30	48.95	62.78	60.74	2.27	3.28	2.07
`drop`	P	0.02	0.02	0.04	1.19	1.00	2.33	0.74	0.96	2.30
	B	0.04	0.05	0.05	2.12	2.97	2.56	1.19	1.83	2.13
	T	0.87	1.03	0.64	47.01	55.79	34.42	8.35	6.25	3.60

droplast	P	0.09	0.06	0.09	1.61	1.13	1.57	1.97	2.73	2.30
	B	0.06	0.06	0.08	1.08	1.00	1.41	2.52	3.15	2.00
	T	0.76	1.40	–	13.41	24.79	–	2.69	2.66	3.02
dropmax	P	1.64	0.89	0.33	4.93	2.67	1.00	8.36	7.69	7.92
	B	0.72	0.60	0.38	2.16	1.80	1.13	8.51	8.62	9.04
	T	7.58	4.98	0.58	22.75	14.96	1.75	2.99	1.70	1.99
enumFromTo	P	–	–	–	–	–	–	11.02	25.39	10.40
	B	204.04	242.81	–	1.00	1.19	–	32.56	29.66	19.49
	T	–	–	–	–	–	–	5.54	4.30	27.50
enumTo	P	0.02	0.02	0.01	2.56	2.35	1.00	0.55	0.70	0.04
	B	0.07	0.08	0.03	7.73	8.56	2.66	3.69	3.32	0.10
	T	1.67	1.04	0.49	173.23	107.39	51.20	4.27	2.37	1.52
factorial	P	0.00	0.00	0.02	1.00	1.03	4.42	2.09	2.05	6.65
	B	0.00	0.00	0.01	1.14	1.15	2.31	1.88	1.89	3.02
	T	209.97	168.07	0.01	≈ 55K	≈ 44K	2.85	12.64	11.12	1.64
isEven	P	0.00	0.00	0.00	1.03	1.16	1.00	1.69	1.90	1.64
	B	0.00	0.00	0.00	1.13	1.12	1.13	1.54	1.53	1.55
	T	0.00	0.00	0.00	1.01	1.02	1.11	1.39	1.39	1.51
isNil	P	0.00	0.00	0.01	1.02	1.00	3.68	1.84	1.79	4.07
	B	0.00	0.00	0.01	1.09	1.08	4.15	1.75	1.67	3.91
	T	0.13	0.16	–	42.89	51.89	–	11.78	12.60	9.75
last	P	0.00	0.00	0.01	1.06	1.37	6.08	1.58	1.67	6.20
	B	0.00	0.00	0.00	1.18	1.20	1.91	1.24	1.48	1.43
	T	0.00	0.00	0.00	1.01	1.00	1.54	1.40	1.32	1.46
length	P	49.00	74.95	343.56	9.93	15.19	69.63	9.02	28.21	38.90
	B	4.93	28.44	148.75	1.00	5.76	30.15	14.71	20.08	37.12
	T	16.40	15.36	6.23	3.32	3.11	1.26	5.41	3.44	1.89
mapAdd	P	0.42	0.27	0.56	1.67	1.06	2.25	5.76	4.66	17.03
	B	0.28	0.25	0.70	1.13	1.00	2.78	3.27	2.83	16.80
	T	2.93	1.94	2.76	11.69	7.71	11.00	2.76	2.62	8.65
mapDouble	P	55.90	20.45	24.21	4.26	1.56	1.85	33.84	13.03	21.42
	B	14.12	13.12	17.60	1.08	1.00	1.34	13.31	18.72	21.81
	T	–	–	–	–	–	–	5.68	4.43	4.54
maximum	P	0.77	0.50	0.87	3.23	2.08	3.62	11.39	8.89	6.65
	B	0.32	0.24	0.61	1.31	1.00	2.54	4.32	3.70	4.52
	T	–	–	1.91	–	–	7.93	308.95	287.81	1.89
member	P	62.06	28.66	56.56	4.75	2.19	4.33	13.13	7.11	16.34
	B	26.95	22.67	50.65	2.06	1.73	3.87	7.22	6.48	16.41
	T	38.07	34.76	13.07	2.91	2.66	1.00	3.99	4.52	0.25
multfirst	P	0.18	0.08	0.13	2.82	1.20	2.06	0.22	0.24	0.46
	B	0.07	0.06	0.14	1.06	1.00	2.18	0.42	0.44	0.61
	T	13.35	1.95	0.70	207.27	30.33	10.90	2.66	2.43	2.18
multlast	P	7.79	1.32	1.19	13.15	2.24	2.00	0.80	0.40	0.67
	B	0.71	0.59	1.00	1.19	1.00	1.68	0.63	0.68	1.11
	T	201.81	72.08	184.20	340.79	121.72	311.05	5.05	3.34	3.68

`nth`	P	77.08	0.81	0.24	382.83	4.02	1.21	0.82	0.81	0.47
	B	0.35	0.34	0.20	1.73	1.70	1.00	0.49	0.74	0.49
	T	–	–	–	–	–	–	10.82	10.15	10.08
`replicate`	P	3.35	0.11	0.12	39.54	1.27	1.46	16.43	2.28	2.11
	B	0.09	0.08	0.14	1.08	1.00	1.63	1.48	1.74	2.05
	T	2.89	1.90	0.70	34.20	22.43	8.29	0.83	1.02	0.64
`reverse`	P	38.44	5.04	36.47	38.99	5.11	36.99	2.44	9.81	11.84
	B	1.71	0.99	17.43	1.73	1.00	17.68	4.89	3.88	11.12
	T	42.57	33.59	159.53	43.18	34.07	161.83	4.35	2.85	3.87
`stutter`	P	–	82.82	34.76	–	5.44	2.28	0.87	5.78	8.15
	B	31.91	15.23	24.59	2.10	1.00	1.61	4.84	3.46	10.18
	T	–	–	–	–	–	–	3.24	2.65	2.63
`sum`	P	0.69	0.37	0.76	2.16	1.15	2.39	13.73	11.28	11.00
	B	0.34	0.32	0.58	1.08	1.00	1.84	13.92	13.25	8.69
	T	1.91	2.11	30.20	5.98	6.63	94.79	2.76	2.98	31.26

Table 4.2: Number of nodes of the synthesised solution.

Name	Plain			Blacklist			Templates		
	Nodes	Types	NoDup	Nodes	Types	NoDup	Nodes	Types	NoDup
`append`	7	7	7	7	7	7	7	7	7
`concat`	7	7	7	7	7	7	7	7	7
`drop`	7	7	7	7	7	7	7	7	7
`droplast`	7	7	7	7	7	7	7	7	7
`dropmax`	9	9	9	9	9	9	9	9	9
`enumFromTo`	–	–	–	13	13	–	–	–	–
`enumTo`	7	7	7	7	7	9	7	7	7
`factorial`	5	5	5	5	5	5	13	13	5
`isEven`	3	3	3	3	3	3	3	3	3
`isNil`	5	5	5	5	5	5	5	5	5
`last`	5	5	5	5	5	5	5	5	5
`length`	9	11	9	9	11	9	9	9	9
`mapAdd`	7	7	7	7	7	7	7	7	7
`mapDouble`	11	11	11	11	11	11	–	–	11
`maximum`	7	7	7	7	7	7	–	–	7
`member`	11	11	11	11	11	11	11	11	11
`multfirst`	9	9	9	9	9	9	9	9	9
`multlast`	11	11	11	11	11	11	13	11	11
`nth`	13	13	13	13	13	13	–	–	13
`replicate`	9	9	9	9	9	9	9	9	9
`reverse`	9	9	9	9	9	9	9	9	9
`stutter`	–	13	13	13	13	13	–	–	13
`sum`	7	7	7	7	7	7	7	7	7

Table 4.3: 37 library components used for synthesis

general functions	`const`, `flip`
booleans	`true`, `false`, `not`
integer constructors	`zero`, `succ`

integer destructors	`isZero`
integer combinators	`foldNat`, `foldNatNat`
arithmetics	`add`, `mul`, `div`, `max`, `eq`, `neq`
list constructors	`nil`, `con`
list destructors	`head`, `tail`, `isNil`
list combinators	`map`, `foldr`, `foldl`, `filter`
list functions	`length`, `append`, `reverse`, `replicate`, `concat`
list of integers functions	`sum`, `prod`, `maximum`, `member`, `enumTo`, `enumFromTo`

4.4 Automatic black list generation

We also used our system to generate an automatic black list based on the identity function. We chose not to generate the polymorphic identity function. As during pruning we are ignoring types, holes and input variables, the programs that would have been generated for the polymorphic identity function are also generated for the identity over any specific type. We choose to generate programs corresponding to the identity function over integers, lists of integers and lists of lists of integers.

Towards this end, we first use the synthesis procedure that combines the PLAIN exploration strategy with the *nof-nodes* cost function to synthesise the first 8 programs of type `Int`, `List Int` and, respectively, `List (List Int)`. For this step we provided to the synthesiser only the constructors `con`, `nil`, `succ`, `zero`. We paired each synthesised program with itself to generate an input-output example.

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

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

we used the synthesis procedure that combines the PLAIN exploration strategy with the *nof-nodes* cost function to synthesise the first 100 programs that satisfy the 8 input-output examples of the corresponding type generated in the previous step. This time we provided the synthesiser with the 37 components listed in Table 4.3.

As a third step, we removed duplicates, the classical program corresponding to the identity function, that is `id x = x` and all programs that use the same input variable more than once. This way we got 212 black list patterns. Section 4.4.1 reviews the quality of the generated black list.

4.4.1 Results

We were able to automatically synthesise 300 programs corresponding to the identity function for three different types using automatically generated input-output examples in under 2 s and extract 212 black list patterns out of them. The number of automatically generated examples, 8, is higher than the number of manual input-output examples we need to generate the same programs. Since programs are generated incrementally, the smaller ones first, the first 7 automatically generated lists of lists of integers do not contain anything except for `nil` and among the first 7 automatically generated lists of integers there are no lists of length more than one. 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 patterns in the automatically generated black list correspond to the identity function, whereas superfluous programs do not necessarily involve application of the identity function. As an example where no identity function is applied, consider `sum nil`, a longer synonym for `zero`.

2. Many automatically generated black list patterns are subsumed by shorter patterns. For example, `append (append nil nil) _` is not needed if the black list already contains `append nil _`.

3. The patterns we extract contain only one wild-card symbol, while patterns with two or more wild-cards would be useful as well. For example, the pattern `foldr _ _ nil` encodes the insight that folding over the empty list is no different than taking the initial value, no matter which function is used for folding.

4.5 Factors affecting runtime

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

4.5.1 Number of components

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

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

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

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

4.5.2 Size of the solution

In the previous sections we mentioned a second factor: the size of the solution to be synthesised. Figure 4.1 shows that the average running time for all nine variants of the synthesis procedure depends exponentially on the number of nodes of the solution found. This goes along with the intuition that a bigger program is more difficult to synthesise.

For example, if we have n possibilities to generate a program consisting of one node, that is `?x` where we have n possibilities to instantiate the hole `?x`, then we will have n^2 possibilities to generate a program with three nodes, that is $?x_1\ ?x_2$ where we have n possibilities to instantiate each hole. However, this simple intuitive explanation does not take into account the contribution of types to search space pruning.

4.5.3 Cost functions

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

nof-nodes prioritises shorter programs and prefers input variables to library components to holes, under the hypothesis that smaller pro-

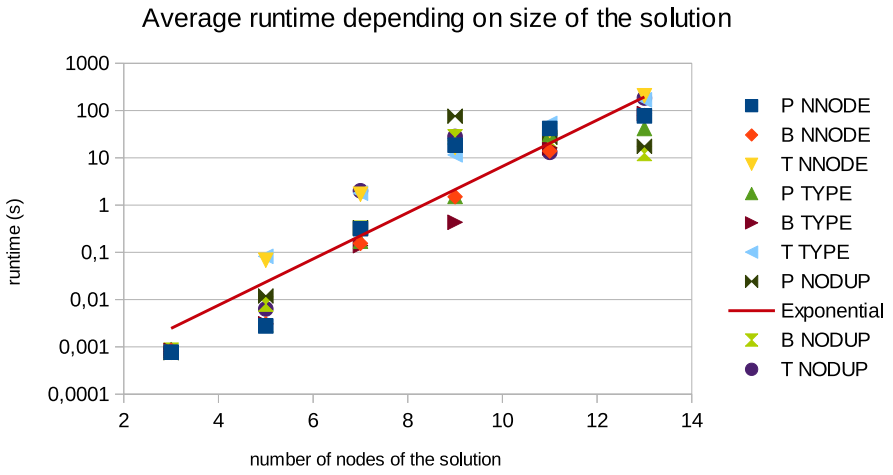

Figure 4.1: Average running time of the variants of the synthesis procedure depending on the number of nodes of the solution.

grams generalise better to unseen examples (c.f. Occam's razor). However, this cost function gives the same cost to the following two programs.

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

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

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

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

instead of the smaller

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

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

The impact on the runtime of this cost function is comparable to the introduction of pruning based on black lists. This has to do with the fact that polymorphic functions that apply in many cases but are rarely

needed, like `const` and `flip`, tend to instantiate their type variables with long arrow types. In the BlackList exploration strategy those programs are filtered by the black list, the *nof-nodes-simple-type* cost function assigns them a higher cost because of their types.

no-same-component prioritises smaller programs with simpler types and additionally penalises the use of the same component more than once. The hope is that without having to inspect programs that take a long time to evaluate like `enumTo (prod (enumTo (prod xs)))`, running times will sink. Against expectations, this is usually not the case. The reason could be that the synthesis procedures that use this cost function examine many larger programs that do not contain any type applications, like `enumTo (mul (succ (div n m)) (succ ?x))`.

4.5.4 Stack vs Queue expansion

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

1. which program should be expanded first
2. which hole of this program should be expanded 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 choose to discuss two. In the first one the holes of a program are organised as a stack, as opposed to the second one, where the holes are kept in a queue.

Organising the holes of a program as a stack leads to left-to-right expansion. To give some intuition, we provide a derivation of `mapAdd`, the function that takes an integer `n` and a list of integers `xs` and adds `n` to each element of `xs`. We show the stack of holes on the right of each partial program. The top of the stack is on the left.

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

Left-to-right expansion often leads to faster synthesis, because leftmost holes usually have more constraints on their type. Consider the program `?x3 ?x4 ?x2` from the derivation of `mapAdd`. We know more about `?x3` than about `?x2`:

the first one must be a function that takes two arguments of some type and returns a list of integers, whereas the second hole could be anything. Furthermore, the instantiation of $?x_3$ with `map [Int]` imposes some constraints on the types of $?x_4$ and $?x_2$.

Keeping the holes of a program in a queue leads to the expansion of the hole with the smallest depth first. This could be useful to control the depth of a program, but in practice it has a substantial drawback. Consider again the derivation of `mapAdd`. The first three steps are the same, but in the program $?x_3$ $?x_4$ $?x_2$ we would now try to expand the hole $?x_2$, that we have absolutely no information about. Every library component and every input variable are valid instantiations of this hole. Thus, this expanding strategy leads to a higher branching factor and explores many superfluous programs like $?x_3$ $?x_4$ `map` and `map (`$?x_5$ `foldr) xs`.

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

4.5.5 Examples

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

a. small enough, so that also undesirable programs such as

   ```
   enumTo (prod (enumTo (prod xs)))
   ```

   do not get stuck or run out of memory trying to construct a list with 479001600 elements, which happens already for the at first sight innocent input `[2,2,3]`;

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

Clearly, using as few and as small input-output examples as possible has a positive effect on performance. On the other side, too few and too general input-output examples can lead to the synthesis of the wrong program, that is a program that satisfies all provided input-output examples but that does not generalise in the expected way. This was especially a problem with `enumFromTo` and `member`. For example, if we provided `enumFromTo` only with examples that result in a list of length three, we got the program that simply concatenated the first input with its successor with the second input.

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

If we provided `enumFromTo` only with examples starting with 1, the synthesised solution was just a call to `enumTo` with the second input variable as argument or, depending on the components we gave, the program corresponding to `enumTo n`:

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

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

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

4.5.6 Blacklist

The search space abounds of superfluous programs that are equivalent to smaller ones. In Section 2.5 we introduced a way to leverage this inconvenience: pruning based on black lists. This approach allows us to avoid exploring 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 [?`$X_1 \to$ `?`$X_2 \to X_3$`] (nil [?`$X_1 \to$ `?`$X_2 \to X_3$`])) ?`x_1 `?`x_2.

A longer black list allows to prune more superfluous programs and considerably reduces the number of programs our synthesis procedure needs to consider before finding a solution. However, in our implementation black list pruning is extremely expensive. For each pattern in the black list and for every subterm of every partial program that is generated, we check, whether the relation *matches* holds. That is, there is a trade-off between the length of the black list and the gain in performance that we can get.

Figure 4.4 shows the black list we used to evaluate the benchmarks. We manually compiled it combining unwanted patterns often seen in the search space with some carefully chosen automatically generated identity functions. We also added some rapidly increasing functions. For example, the following program computing tetration[1] represents a problem for our evaluator.

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

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

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

Table 4.4: Manually compiled black list patterns used for evaluation in the BlackList exploration strategy.

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

In Table 4.1 we see that the runtime profits the most from the introduction of black list pruning when we use the cost function *nof-nodes*. The running time drops less significantly if we use other cost functions. A possible explanation of this behaviour could be the fact that other cost functions give a higher cost to those programs that are filtered with our black list.

We could also empirically see that pruning using black lists is very helpful in the presence of polymorphic functions that apply in many cases but are rarely needed, for example `flip`, `const` or `uncurry`. Forbidding a fully applied `flip`, `const` or `uncurry` has a comparable effect on performance to taking those components out of the library. However, since we are not taking

those components out of the library, we are still able to synthesise functions that need them.

4.5.7 Templates

In Table 4.1 we see that the synthesis procedures that use the Template exploration strategy fail more often to find a solution within the timeout. Moreover, even if they find a solution, they tend to be 10 times slower than the other synthesis procedures, `length`, `member`, `replicate` and `reverse` being an exception.

The main reason for this slowdown resides in our implementation. In particular, in the successor rules we presented in Section 2.6. Consider following derivation of a template for `replicate`, where `X` represents the input type variable, `n` the first argument and `x` the second. The list on the right of each program shows the type of its non-delayed holes.

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

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

The solutions synthesised by the synthesis procedures that use the Template exploration strategy tend to contain more higher-order components and in two cases are surprisingly long. In Table 4.2 we can see that two variants of our synthesis procedure synthesise a program with 13 nodes for `factorial`, whereas all other variants find one with only 5 nodes.

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

4.5.8 Unknown factors

Individual results show that there must be other factors influencing the runtime. Take, for example, `enumFromTo`, `stutter` and `nth`. All three of them have a solution with exactly 13 nodes, but their runtimes differ at least by an order of magnitude for the synthesis procedures that do not time out on `stutter`. What makes `nth` generate in less than 1 s, `stutter` a hundred times slower and `enumFromTo` to time out in most of the cases?

4.6 Synthesised solutions

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

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

replicate [X] n x
    = foldNat [List X] (con [X] x) (nil [X]) n
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Analogously, a "wrong" solution for `member` turned out to be a clever implementation of `isEven`, namely `foldNat not true n`. Folding over an integer is similar to recursion over that integer. In this case the base case of the recursion is `true` and in the inductive case we negate `isEven (n-1)`.

Another unexpectedly clever solution is due to the absence of a polymorphic equality function. We only have equality over integers, thus the benchmark `member` is not polymorphic but has the type `Int` → `List Int` → `Bool`. This allowed the synthesiser to generate, along with the expected solution

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

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

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

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

4.7 User-synthesiser interaction

In our experiments we also explored some rudimentary ways of user-synthesiser interaction in order to help the synthesiser to avoid the three classes of undesired programs listed in Section 5.1.1.

The simplest way to help the synthesiser is to restrict the library to the components needed for the synthesis task at hand. For example, the user might expect that `enumFromTo` can be synthesised by putting together `enumTo`, `map`, `add`, `sub` and the integer and lists constructors. However, in our experience it is easy to miss some of the components.

Another way to help the synthesiser is to generate helper functions first and add them to the library before the synthesis of the desired program. For example, to speed up the synthesis of `enumFromTo`, we first generated `enumTo` from the components we had. This increases the size of the library, but reduces the size of the generated solution.

The last way to help the synthesiser that we tried is to specify a starting template. For example, the user might have the intuition that `enumFromTo` can be synthesised by adding some integer to each element of a list of consecutive numbers. They can encode this intuition into the template:

```
enumFromTo m n = map [Int] [Int] (add ?i) (enumTo ?j)
```

and get the desired implementation:

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

However, user-input is usually not needed to synthesise the benchmarks in less than a minute, `enumFromTo` being the only exception.

4.8 Comparison to related work

We compare our synthesis procedure that uses the BLACKLIST exploration strategy combined with the *nof-nodes-simple-type* cost functions with the state-of-the-art tools reported in Chapter 3. The results are summarised in Table 4.5, where for each tool we provide the specification size (expressed in number of examples for example-based tools and in AST nodes for SYNQUID) and the runtime.

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

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

```
isEven n = foldNat not true n
```

```
isEven 0 = true
isEven n → not (isEven (n-1))
```

It might be a consequence of the simplicity of our target language, but as we can see in Table 4.5, our tool needs less input-output examples to synthesise the benchmarks than the other example-based tools. SYNQUID relies on a different specification that requires a higher level of expertise, which makes direct comparison difficult.

The number of provided components also changes across the tools. With 36 to 37 components, we provide the largest library to our tool. On the second

Table 4.5: Comparison of our tool with the state-of-the-art. For our tool, λ^2 and Myth the *spec* column shows the number of examples, for Synquid it shows the specification size in AST nodes. For all tools the *time* columns show the runtime in seconds. The cells corresponding to benchmarks that were not tested with the respective tool are left empty.

	Tamandu		Synquid		λ^2		Escher		Myth	
Name	spec	time	spec	time	spec	time	spec	time	spec	time
`append`	2	0.07	8	0.05	3	0.23			12	0.01
`concat`	2	0.04	5	0.05	5	0.13	–	0.06	6	0.02
`drop`	2	0.05	14	0.29			–	0.02	13	1.29
`droplast`	2	0.06			6	316.39				
`dropmax`	2	0.60			3	0.12				
`isEven`	2	0.00					–	0.02	4	0.01
`isNil`	2	0.00	6	0.02						
`last`	2	0.00			4	0.02	–	0.02	6	0.09
`length`	2	28.44	4	0.10	4	0.01	–	0.01	3	0.02
`mapAdd`	2	0.25			5	0.04				
`mapDouble`	3	13.12	7	0.06	3	0.11				
`maximum`	3	0.24			7	0.46				
`member`	7	22.67	6	0.03	8	0.35				
`multfirst`	2	0.06			4	0.01				
`multlast`	2	0.59			4	0.08				
`nth`	3	0.34							24	0.96
`replicate`	3	0.08	7	0.06						
`reverse`	2	0.99	11	0.29	4	0.01	–	0.22		
`stutter`	2	15.23					–	0.55	3	0.02
`sum`	2	0.32			4	0.01	–	0.06	3	0.01

place, Escher uses a library of 23 components to evaluate all benchmarks. The components provided to Myth are not listed in the paper, but the library is similar to Escher's and we believe that it is not larger. Synquid does not provide more than 5 components and 6 measure functions over the needed data types to any of its benchmarks.

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

Chapter 5

Conclusions

5.1 Conclusions

Type-driven synthesis of functional programs from input-output examples strives to automatically generate well-typed programs that satisfy the given input-output examples and generalise well to unseen input-output pairs. When solving similar tasks, human programmers rely on well-known computational patterns.

The main goal of the thesis was to study how well first- and higher-order components can guide and speed up the synthesis process.

In order to achieve this goal, we implemented a prototype of a simple synthesis procedure based on program enumeration in OCaml. Extensive evaluation of the prototype on benchmarks showed that it can compete with related state-of-the-art tools. We believe therefore that synthesis from library components is a promising direction and needs to be explored further.

The prototype implements a synthesis procedure based on best-first enumeration of partial programs. We experimented with different cost functions and with different strategies to expand a partial program.

5.1.1 Observations about the search space

During experimentation we faced different classes of programs that hinder the synthesis process.

First of all, we discovered that the search space abounds of closed programs that represent a challenge for our evaluator, even on examples as small as `[2,2,3]` and `3`. This was our main motivation in reducing the size and the number of input-output examples.

Next, we noticed that every hole can be filled in with `head [?X] (nil [?X])`, where `?X` has to be instantiated with the type of the hole to expand.

Many partial programs containing this pattern were enumerated just to be ruled out as soon as all holes were filled in and the closed programs were evaluated. To prune whole branches that will surely not lead to a solution, we introduced a black list containing the undesired patterns. As a next step, we used the synthesiser to generate such a black list automatically.

A related problem are partial programs that can be ruled out based on the semantics of the library component and the expected behaviour of the target program. For example, when trying to synthesise `enumFromTo`, no human programmer would insist on taking `enumTo` as the first component. The partial program

```
enumFromTo m n = enumTo ?x
```

will not lead to the solution, because every list returned by `enumTo`, no matter the argument, starts with 1, whereas the list returned by `enumFromTo` should start with `m`, which is not necessarily 1. Our synthesis procedure, on the contrary, treats this partial program and its successors as very promising candidates. This motivated us to try a slightly different synthesis procedure: the one that fixes the higher-order components first (generating a *template*) and then fills in the remaining holes with input variables and first-order components.

Empirical evaluation showed that reducing the size of the examples and blacklisting undesired patterns improves the performance, as expected. On the other hand—against expectations—the introduction of templates leads to a significant slowdown. We believe, however, that this could be changed with a more sophisticated implementation and should be investigated further.

5.2 Future Work

As anticipated in the previous section, the information about well-known computational patterns encoded as library components can be successfully reused in program synthesis. However, in order to reach a publishable unit, this promising direction needs further exploration. In this section we list some possibilities for future work that follow from the limitations of our system.

5.2.1 Automatic black list generation

Pruning based on black lists helps to reduce synthesis time. However, manually compiled black lists are inconvenient when users are allowed to add their own components to the library. Automatically synthesising a black list from the components in the given library would alleviate this inconvenience.

In Section 4.4 we discussed how we used our synthesis procedure to automatically generate a black list. However, we arrived at the conclusion that our method is too primitive to compete with a manually compiled black list. In particular, the generated patterns have only one wild-card and correspond to the identity function. The former restriction leads to the generation of patterns like

```
foldr append _ nil
foldr con _ nil
foldr drop nil
```

that we would like to replace with the one pattern `foldr _ _ nil`. The more general pattern not only helps keeping the size of the black list small but also rules out more of the superfluous programs. The latter restriction does not allow us to generate longer synonyms for terms like `nil` and `zero` that do not involve the application of the identity function, as for example `enumTo zero` and `sum nil`.

In general, a good black list contains only the shortest most general patterns that rule out superfluous programs. It is a question of interest how to generate such a black list automatically. We believe that a good result could be achieved by exploring patterns with two of more holes and setting other functions (not only the identity) and a synthesis goal.

TODO: Write me Generalise identity function to projections of the form `proj(x1, x2, ... xi) = xi`, where the types of the arguments are free in the sense that they can be instantiated with anything. Might be not worth it. Since we need a short list, it could make sense to concentrate on some concrete variants of the projection function.

5.2.2 Templates

Another interesting question that our thesis does not answer is whether fixing the higher-order components first in form of templates helps to guide and speed up synthesis. As discussed in Section 4.5.7, the successor rules currently used in our prototype implementation generate a lot of undesirable templates such as:

```
?x (foldr [?X] [?Y])
?x (map [?X1] [?Y1]) (foldr [?X2] [?Y2])
```

Rewriting the successor rules or pruning templates according to some heuristic could shed light on the practical impact of this approach. In particular, we believe that limiting the size of the programs being synthesised instead of the depth of the first-order search could be a first step towards a more efficient implementation. The reason for that is that undesirable templates

tend to have very few successors in the first-order search. As a result, partial programs being explored are typically quickly growing applications of holes.

5.2.3 Harder examples

TODO: write me More experimental work has to be done. Harder benchmarks have to be synthesised. We encourage to define trees and experiment with programs over trees and compound data structures.

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] N.G de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381 – 392, 1972.

[4] 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.

[5] Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. Example-directed synthesis: A type-theoretic interpretation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 802–815, New York, NY, USA, 2016. ACM.

[6] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 317–330, New York, NY, USA, 2011. ACM.

[7] Sumit Gulwani, Vijay Anand Korthikanti, and Ashish Tiwari. Synthesizing geometry constructions. In *Proceedings of the 32Nd ACM SIGPLAN*

Conference on Programming Language Design and Implementation, PLDI '11, pages 50–61, New York, NY, USA, 2011. ACM.

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

[9] Jeevana Priya Inala, Xiaokang Qiu, Ben Lerner, and Armando Solar-Lezama. Type assisted synthesis of recursive transformers on algebraic data types. *CoRR*, abs/1507.05527, 2015.

[10] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 215–224, New York, NY, USA, 2010. ACM.

[11] Emanuel Kitzelmann. Logic-based program synthesis and transformation. chapter Analytical Inductive Functional Programming, pages 87–102. Springer-Verlag, Berlin, Heidelberg, 2009.

[12] 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.

[13] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Complete functional synthesis. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 316–329, New York, NY, USA, 2010. ACM.

[14] Zohar Manna and Richard Waldinger. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1):90–121, January 1980.

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

[16] 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.

[17] Daniel Perelman, Sumit Gulwani, Dan Grossman, and Peter Provost. Test-driven synthesis. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 408–418, New York, NY, USA, 2014. ACM.

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

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

[20] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 159–169, New York, NY, USA, 2008. ACM.

[21] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 404–415, New York, NY, USA, 2006. ACM.

[22] Phillip D. Summers. A methodology for lisp program construction from examples. *J. ACM*, 24(1):161–175, January 1977.

ETH

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):

Top-Down Inductive Synthesis with Higher-Order Functions

Authored by (in block letters):

For papers written by groups the names of all authors are required.

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

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)**
Zürich, 22.08.2016	A. Максимова

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