



CHALMERS
UNIVERSITY OF TECHNOLOGY



UNIVERSITY OF GOTHENBURG

Complementing the Digital Programming Tutor Ask-Elle with Program Synthesis

Master's thesis in Computer science and engineering

Gustav Engsmyre
Karl Wikström

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2021

MASTER'S THESIS 2021

Complementing the Digital Programming Tutor Ask-Elle with Program Synthesis

Gustav Engsmýre
Karl Wikström



UNIVERSITY OF
GOTHENBURG



CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2021

Complementing the Digital Programming Tutor Ask-Elle with Program Synthesis
Gustav Engsmysre, Karl Wikström

© Gustav Engsmysre, Karl Wikström, 2021.

Supervisor: Alex Gerdes, Department of Computer Science and Engineering
Examiner: Patrik Jansson, Department of Computer Science and Engineering

Master's Thesis 2021
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in L^AT_EX
Gothenburg, Sweden 2021

Complementing the Digital Programming Tutor Ask-Elle with Program Synthesis

Gustav Engsmysre

Karl Wikström

Department of Computer Science and Engineering

Chalmers University of Technology and University of Gothenburg

Abstract

Practising is a part of learning how to program. Usually, teachers help students with this, but digital alternatives are available. Ask-Elle is such an alternative, developed for teaching the functional programming language Haskell. Students can submit partial solutions to exercises and will receive feedback from Ask-Elle. However, it has some limitations when dealing with partial solutions that diverge from the structure of its reference solutions.

We present a proof of concept to complement Ask-Elle (called THUPY) that aims to handle solutions where students do not follow the structure of the reference solutions from Ask-Elle. We use program synthesis to generate a suggested next step for a student, with more focus on the *behaviour* rather than the structure of the reference solutions.

Our results show that this is a promising idea. Evaluating THUPY on a dataset gathered from Ask-Elle, we manage to provide feedback on 21% of student solution where Ask-Elle cannot, and 35% of solutions where it can. We measure an average runtime of ~ 3 seconds, which is an acceptable amount of time to wait for feedback.

Keywords: Computer Science, Functional Programming, Digital Programming Tutor, Program Synthesis, Haskell, Ask-Elle

Acknowledgements

First, we would like to thank our supervisor Alex Gerdes for guiding and supporting us through our work. We would also like to thank Johan Jeuring, Niek Mulleners, and Bastiaan Heeren at University of Utrecht and Open University for bringing valuable input and ideas in the early stages of our thesis. Finally, we would like to thank everyone who have volunteered their valuable time and energy to read and help us improve this thesis.

Gustav Engsmysre, Karl Wikström, Gothenburg, June 2021

Contents

1	Introduction	1
1.1	Research Question	3
1.2	Limitations	3
1.3	Contributions	4
1.4	Outline	4
2	Background	5
2.1	Haskell	5
2.1.1	Types	5
2.1.2	Eta-Transformations	6
2.1.3	The <i>Glasgow Haskell Compiler</i>	6
2.1.4	QuickCheck	7
2.2	Program Synthesis	7
2.2.1	Sketching	8
2.2.2	The Search Space	8
2.3	Intelligent Digital Programming Tutors	9
2.3.1	Model-Tracing	9
2.4	Ask-Elle	9
2.4.1	Strategies	10
3	Generating Feedback using Program Synthesis	13
3.1	System Overview	13
3.2	Working with Abstract Syntax Trees	14
3.2.1	Transforming Haskell Code to AST	16
3.2.2	Transforming AST to Haskell Code	16
3.3	Synthesising code	16
3.3.1	Eta-Expanding the Model Solutions	18
3.3.2	Combining the Student Solution with the Model Solutions	18
3.3.3	Renaming	19
3.3.4	Testing	21
3.4	Generating Feedback	24
3.4.1	Formatting the Hints	24
4	Evaluation	27
4.1	The Dataset	27
4.1.1	Compatibility	27

4.2	Accuracy	28
4.2.1	Result	28
4.2.2	Discussion	29
4.3	Performance	31
4.3.1	Result	31
4.3.2	Discussion	33
5	Discussion	35
5.1	Comparing Program Synthesis to Strategies	35
5.2	Expression-Synthesis and Pattern-Synthesis	36
5.3	Handling GHC	36
5.3.1	Type Equivalence & Variable Substitution using GHC	36
5.3.2	Coupling to GHC	37
5.4	Future Development and Improvements	37
5.5	Societal, Ethical and Ecological Aspects	37
5.5.1	Digital Tutors	38
5.5.2	Program Synthesis	38
6	Related Work	41
6.1	CodeQ	41
6.2	Hazelnut Live	42
6.3	MagicHaskeller	42
6.4	Hoogle+	43
6.5	Program Synthesis Libraries in Haskell	43
7	Conclusion	45
	Bibliography	47
A	Listing of Full AST Code	I
B	Full AST of Figure 3.1	V
C	Evaluation Script	VII
D	Some generated solutions from the evaluation	XI

1

Introduction

In a society where software is an increasingly important part of our lives, the skills of a software developer are needed more than ever. Practising programming by solving exercises is an important step in learning these skills. Exercises are usually provided by humans but preparing and supervising this can be time consuming. Additionally, the coupling between the student and the teacher can be a limiting factor, for example, students might not get access to the same tutoring due to physical or economical constraints.

By imitating a teacher's role in tutoring, digital tutors can offer solutions to these problems. One such tutor is Ask-Elle. It is an online digital tutor aimed at teaching the functional programming language Haskell at an introductory level. A key feature of Ask-Elle is letting students from anywhere in the world submit partial solutions to exercises, to which Ask-Elle can provide feedback. These partial solutions can have left out parts (called *holes*) that are each marked with a ?. The given feedback may enable the student to take the next step. Repeating this process guides the student towards a complete solution. In the following example (Figure 1.1) we can see what a partial solution to the exercise **dupli** can look like and what feedback is given when Ask-Elle is asked for help. For the **dupli** exercise, the student should write a function that duplicates each element in a list.

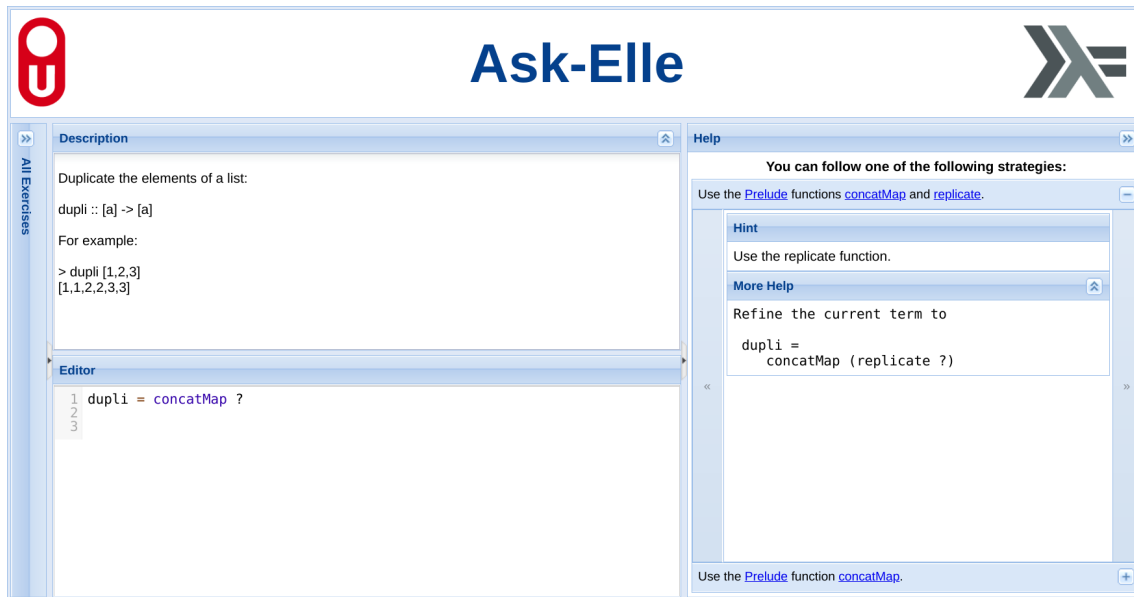


Figure 1.1: A student has entered a partial solution to the exercise called **dupli** and has received feedback on the right side.

However, Ask-Elle has its limitations. When faced with a partial solution that does not directly follow one of its predefined model solutions, Ask-Elle can no longer determine if a student is on the right track. A student's deviation from a model solution does not have to be significant as shown in the following example:

dupli = concat . map ?

The small deviation from `concatMap` to `concat . map` throws Ask-Elle off-guard even though the solution would be practically identical. Instead, Ask-Elle just replies "You have drifted from the strategy in such a way that we can not help you any more." While Ask-Elle tries to mitigate this problem by normalising the student programs, this approach does not scale well. This inherently limits Ask-Elle in what it can give feedback to.

We present an approach to complement Ask-Elle, such that Ask-Elle becomes less strict, by using program synthesis to generate code that completes the partial solution. This approach focuses more on the functionality of the code rather than the structure of the model solutions. Instead, the parts of the model solutions are used to generate a complete solution from the provided input. This approach gives students more freedom in structuring their programs whilst keeping them relatively close to a model solution. We call our proof of concept application *Tutor extension for Haskell Using Program synthesis*, or THUPY for short. The complement uses program synthesis to generate a solution from the program that the student has provided, together with the model solutions. It then guides the student towards the generated solution by giving the student a suggested next step. For instance, for the following model solution:

```
modelDupli :: [Int] -> [Int]
modelDupli list = concatMap (replicate 2) list
```

THUPY will find that replacing the hole in the previous student solution, `dupli` with `(replicate 2)` from `modelDupli` yields a valid solution. To let the student figure out the solution by themselves, THUPY only provides the student with a suggested next step, as shown below.

```
dupli = concat . map (replicate ?)
```

Although the focus of this thesis is on Ask-Elle, the general approach should be applicable to other tutors as well.

1.1 Research Question

This thesis examines if and how program synthesis can be used to complement Ask-Elle. The goal can be formalised into three questions that we seek to answer:

1. Is it possible to use program synthesis to generate feedback that suggests a next step to complete a partial program?
2. If so, how well does it perform on partial programs that Ask-Elle cannot handle?
3. Also, how well does it compare against the existing Ask-Elle feedback algorithm?

To evaluate if we have achieved the goals of this thesis, we focus on how well our program synthesis approach can generate feedback in the form of a suggested next step for a user. We apply our complement to a set of partial programs previously handled by Ask-Elle. These attempts are divided into two categories, those that Ask-Elle has succeeded to give feedback to, and those that it did not. We evaluate for how many of each our complement can suggest a next step. This allows us to assess how well THUPY compares to and complements Ask-Elle respectively.

1.2 Limitations

To limit the scope of this thesis, we will not perform any user evaluations. Furthermore, the quality of the feedback that is generated will not be assessed. The focus of the thesis will instead be the technical aspects and assumes that providing a next step is useful feedback. In addition, THUPY will only support monomorphic types, because polymorphic types are needlessly complicated for our proof of concept. Furthermore, we limit ourselves to the subset of the Haskell language that the exercises in Ask-Elle uses and limit our program synthesis to expressions. Finally, there is no focus on optimising the performance beyond running in a reasonable time. Informally, reasonable time means that the program is fast enough to not make the user quit out of frustration, which we deem to be within a few seconds.

1.3 Contributions

Our main contributions described in this thesis are:

1. THUPY, a proof of concept to complement Ask-Elle, that uses program synthesis to generate a next step from a partial solution.
2. Evaluate how well the above contribution complements and compares to Ask-Elle.

1.4 Outline

Chapter 2 presents the needed background to understand the following chapters. The next chapter explains how we implemented our solution and motivates some of the choices we made. In Chapter 4 we present and discuss our results. Next, we discuss our work, some difficulties that arose during the project and highlight possible future work. We then present some related work and how our project relates to them. Lastly, we present our conclusions drawn from working on this project.

2

Background

In this chapter, we present the concepts and tools needed to understand the following chapters. We first present the Haskell programming language and some related tools. We continue with a brief introduction to different aspects of program synthesis. After this, we introduce digital programming tutors and describe the model-tracing approach to providing feedback. Finally, we introduce the digital tutor Ask-Elle in more detail.

2.1 Haskell

Haskell is a functional, statically typed programming language [1]. As a functional language, Haskell’s functions are first-class citizens, meaning that e.g. functions can be passed to other functions as parameters [1]. Additionally, functions in Haskell are pure, which means that a function application has no side-effects and that they produce the same output for the same input [2]. Since Ask-Elle was developed to teach Haskell and most tools to work with the language are also written in Haskell, we chose to develop THUPY in it as well.

2.1.1 Types

Handling types is an important part of our synthesis, and we briefly introduce some important concepts of the Haskell type system in this section.

Haskell is a statically typed language. At compile time the type of every expression is already known [3]. Furthermore, Haskell can infer the type of an expression from the type signatures of the expressions around it [3].

The Haskell type system allows for polymorphic types. A polymorphic type in Haskell means that it can represent multiple different types [3, 4]. Monomorphism is the opposite of polymorphism, a concrete type, for instance `Int` [5]. At runtime the polymorphic type can only ever take the form of one type at a time [4]. For instance the type variable `a` can represent any type, such as `Int` or `Char`. However, it cannot represent both `Int` and `Char` in the same context. Consider the following function type signature:

```
f :: a -> a -> a
```

Here we have a function that takes two inputs and produces one output. They all have the same type, but that type can vary depending on how the function is used. For example, applying it to two `Ints` would produce an `Int` and applying it to two `Strings` would produce a `String`.

The polymorphic types can be constrained, for instance, `Num a` means that the type variable `a` can only be numerical types [4]. Common numerical types are `Int` and `Double`.

2.1.2 Eta-Transformations

Haskell is based on the formal mathematical lambda calculus system. Some of the transformations from this are directly applicable to Haskell functions [6, 7].

In lambda calculus, an eta-transformation can be described as follows:

$$\lambda x.Mx \xrightarrow{\eta} M \text{ if } x \text{ is not free in } M \text{ and } M \text{ is of function type [8].}$$

A reduction on this form can be used to normalise an expression [9]. A very simple example from the Haskell wiki of an eta-reduction is the lambda expression `\x -> abs x` $\xrightarrow{\eta}$ `abs` [10]. Eta-expanding the eta-reduced expression yields the original expression: `abs` $\xrightarrow{\eta}$ `\x -> abs x`.

In THUPY we use eta-transformations to create alternate versions of the model solutions. This gives us additional, and sometimes crucial, expressions that we can use when synthesising solutions. This process is described more in-depth in Section 3.3.1.

2.1.3 The *Glasgow Haskell Compiler*

The *Glasgow Haskell Compiler* (GHC) is according to a 2017 survey the most used compiler for Haskell [11]. It is not uncommon for compilers to offer support for extending and modifying their behaviour. GHC does this by providing its functionality as a library [12]. This gives us the ability to tap into the internals and to make use of its functionality.

Compilation of Haskell code using GHC, like most other compilers, consists of a number of phases [13]. Each phase takes as input the output from the previous phase, manipulates the input and gives an output for the next phase to process [13, 14].¹ To understand this thesis, only the parsing and type-checking phases need to be explained.

- The *parsing* phase takes a Haskell source file as input. It syntax-checks and parses the code from an unstructured string into an *abstract syntax tree* (AST) [13]. An AST is a tree structure that many compilers use to represent the code internally.

¹The entire pipeline can be seen here: <https://gitlab.haskell.org/ghc/ghc/-/wikis/commentary/compiler/hsc-pipe>

- The ***type-checking*** phase adds type information to the AST. It also provides the information needed to reconstruct the type of any expression. If the types in the AST do not fit, the type-checker returns an error to the user [13]. We use the type-checked AST to retrieve type information about the code.

Typed Holes in GHC: A *hole* in GHC is a feature where a programmer can specify a location in their code where they want help figuring out what to write [15]. These holes are marked with an `_` (underscore) in the code. The compiler recognises the holes and uses type-inference to give the hole a type. GHC then returns an error message with the type of the hole along with a list of suggested possible bindings (names of variables or functions) that would fit in the hole [15].

Although similar to how THUPY works, GHC only provides a list of suggestions based on type, whereas THUPY does it based on both type and functionality. The functionality is determined by testing the input-output behaviour of the function. THUPY uses GHC to calculate the type of a hole.

2.1.4 QuickCheck

QuickCheck is a property-based testing library for Haskell [16]. It tests functions by checking if a given property holds for it. This is done by applying arbitrary inputs and checking if the output matches the specified properties [16].

An example of a simple property is that a list reversed twice should be equal to the original list. In the original QuickCheck paper [16], this property is expressed like this:

```
prop_RevRev xs = reverse (reverse xs) == xs
```

Both Ask-Elle and THUPY use QuickCheck to check if a solution has the expected behaviour. However, where Ask-Elle only tests the partial solution from the student, THUPY tests if any synthesised solution has the expected behaviour.

2.2 Program Synthesis

The idea of program synthesis is to automatically generate programs from a specification [17, 18]. In traditional programming, the programmer implements a program that meets a particular specification. In contrast, program synthesis generates a program that adheres to a specification given by a programmer [17, 18]. Like compilers, program synthesisers translate a statement from one language to another, often from a “higher” level language (a specification) to a “lower” level language (an implementation) [17, 18].

Describing the specifications can be done in different ways, but some common methods are:

- **Input-output examples:** This approach, like the name suggests, uses a

predefined mapping between what output the function should produce given a certain input to synthesise a program [19]. If the input-output examples do not fully specify the desired behaviour of the program the synthesised program might not have the desired behaviour. A solution is to add more examples to specify the behaviour more precisely [20].

- **Logical specifications:** This approach uses a given logical specification that the output of the function should satisfy in relation to the input that was given [20]. An example of a logical specification of a function calculating the maximum of two integers is the following:

$$\begin{aligned} \max \ x \ y \rightarrow z \\ (z = x \wedge x > y) \vee (z = y \wedge y \geq x) \end{aligned}$$

Logical specifications need not be understood for this thesis, but illustrates that there are many types of program synthesis.

- **Oracle guided program synthesis:** This implementation is quite close to input-output examples. A finished black box program, called an oracle generates correct output for any input with regards to the specification [21]. Input-output examples can thus be automated [22].

Our approach uses an oracle to generate input-output examples. Furthermore, we apply a sketching approach (which is explained in the next section) for generating feedback.

2.2.1 Sketching

Sketching is a type of program synthesis where a programmer supplies the program synthesis algorithm with both a specification, and a partly finished program, or a “sketch” [23, 24]. The partly finished program contains holes [23], much like those in Ask-Elle and Haskell. The goal for the synthesiser is to generate code that fills in the holes to complete the program and make sure that it fits the specification [23].

By sketching a skeleton of the program, the programmer is given more control of the overall flow of an application [23, 24]. They can also specify the structure that the synthesised program should have. The synthesiser only operates and generates code on the parts where the programmer has left a hole [23, 24]. Sketching effectively divides the program synthesis into two sub-problems: *Program Search* and *Program Verification*. In our case, we search for all possible expressions that could fit into the hole. Then we verify that they make the solution as a whole behave correctly.

2.2.2 The Search Space

Synthesising code involves searching through a set of programs to find one that fits the specification [25]. This set of programs are usually referred to as the program search space, and quickly grows very large [25]. The search-space can be comprised of a subset of the language that the program synthesis operates on [26].

Programs in this search-space can be discovered using different search techniques [26]. The most basic technique is a brute-force search of the entire search space, which enumerates every possible program and tests if it satisfies the specified properties and constraints [26]. A pure brute-force technique is often not feasible due to resource constraints. To overcome this, the search space can be reduced by applying various optimisations to shrink it [20]. An example of reducing the search space could be to restrict it to expressions of a specific type.

2.3 Intelligent Digital Programming Tutors

An intelligent digital programming tutor is a program that is designed to mimic tutoring from a teacher but in a digital tool [27]. The main feature of most digital tutors is that they provide feedback and hints to students to help them write a program that has some desired properties [27]. A digital tutor can also help a student understand programming better [27]. This can be done by, for example asking quiz questions about the programs written by the student, and by helping the student design their solution [27].

Digital tutors have been proven to be a more effective method of teaching introductory programming to students than a traditional approach where the student is given an exercise to solve by themselves [28]. Using digital tutors as a helping tool has also been shown to decrease the amount of help needed from teachers, while not decreasing the performance on tests taken by the students [29]. Thus, once a programming tutor has been deployed, it lowers the need for a teacher or TA to aid students with trivial questions. It also allows students to learn at their own rate, while teachers can focus on more advanced questions [28].

2.3.1 Model-Tracing

Model-tracing was an early approach for constructing a digital tutor. The concept traces back to at least the 80s [30]. A tutor using a model-tracing approach solves the problem alongside the student. The tutor has divided the process of writing a specific program into a set of steps. While the student is solving the problem, the tutor is keeping track of which step the student has reached towards an ideal student solution, a so-called model solution. This step then has associated feedback which can be used when the student asks for help [30, 31]. Most digital tutors that use this approach make use of predefined model solutions to guide the student towards a well-written program. Many tutors also include a number of models that describe common incorrect steps taken by students. This enables tutors to provide more accurate and individualised feedback to a “wrong” step [32, 30, 33].

2.4 Ask-Elle

Ask-Elle is a digital tutor. It has a set of exercises that a student can attempt to solve [32]. Each exercise consists of a description of the function that the student

should implement. They can submit a *partial* solution to Ask-Elle that may contain holes marked with `?`. Such a hole marks a location in the code where the student wants feedback on how to proceed. Ask-Elle tries to give feedback on how to continue toward one of its predefined solutions, called model solutions. These model solutions are defined by teachers [34].

Ask-Elle normalises a student's attempt using a set of program transformations, which preserves the semantics of the program. The normalisations both optimise the student program and allow the system to better match an attempt to a model solution [32]. Apart from normalising the solutions, Ask-Elle internally replaces the holes with **undefined**. This allows the compiler to compile the program, given that the rest of it type-checks and is well-formed. If the program does not compile with these changes, there is a problem apart from the holes in the submitted solution.

Each submission to Ask-Elle is stored in a database along with the response and some metadata about the request and response [32]. Gerdes et al. [32] have used this to examine how well the tutor could generate feedback to user requests. They classified each student interaction as:

1. **Compiler error** - The compiler has reported a syntax or type error.
2. **Model** - The student has either solved the solution according to a model solution or is on track to do so.
3. **Counter** - Ask-Elle has run QuickCheck and found that at least one test has failed.
4. **Tests passed** - Ask-Elle has determined that the solution has passed all tests, but does not follow any model solution.
5. **Discarded** - The solution compiles, but does not follow any model and tests cannot be run against it. These entries represent where Ask-Elle has not managed to give feedback and are thus the attempts we are interested in solving when attempting to complement Ask-Elle.

2.4.1 Strategies

Ask-Elle tries to guide a student towards a predefined solution using a procedure called *strategies* [34]. The strategies are derived from the model solutions and track the student's progress throughout the exercise. Furthermore, if a teacher wishes, the model solutions can be enriched with feedback annotations. These provide more detailed feedback to certain parts of the functions and can add descriptions on how the solution works [34]. Strategies is a form of model-tracing, described in Section 2.3.1 [32].

In Listing 1 an example of a model solution to the exercise **fromBin** is given. The function should convert a list of bits (a binary number) to a decimal integer. This model solution has annotations to specify feedback and give a description of the solution.

```
{-# DESC Implement fromBin using the @foldl @prelude function. #-}

fromBin =
  {-# F Define the fromBin function using @foldl. The operator
    should multiply the intermediate result with two and add
    the value of the bit. This solution therefore multiplies
    every bit in the list n-times by two while summing the
    individual bits. #-}
  foldl op 0
  where
    n `op` b = {-# F Multiply n by two and add b. #-} 2*n + b
```

Listing 1: An example of a model solution in Ask-Elle with a description and feedback given as annotations.

3

Generating Feedback using Program Synthesis

In this chapter, we describe how we use program synthesis to suggest a next step based on input to THUPY. We start with a quick system overview, followed by a description of the details of how we use ASTs (abstract syntax tree) in our project, we continue to explain the different steps we use in our program synthesis. Finally, we describe how we use our synthesised solution to generate a next step.

3.1 System Overview

We present an overview of our approach for generating feedback using program synthesis. The program synthesis algorithm is based on input-output examples and sketching, which are both explained in Section 2.2. Our approach attempts to generate a set of complete solutions from a partial student solution. This synthesis step starts with extracting expressions from the model solutions. It then tries to put these into the holes of the partial student solution. After this, it renames any variables in the inserted expressions to fit into the partial solution. Next, it tests all the possible combined solutions to check which, if any, have the expected properties. These are called the valid solutions. Finally, it uses a valid solution as a blueprint to create the next step for the student and return this as feedback.

By synthesising the code using expressions from the model solutions, we push the system to synthesise solutions “close” to a combination of one or more model solutions. This also limits the search space and speeds up the procedure.

To illustrate this system in action, let us assume that a student gives the following partial solution for the exercise **dupli**:

```
dupli :: [Int] -> [Int]
dupli = concat . map ?
```

Here a hole is placed in the first argument to the **map** function. Now, consider that a teacher has supplied the following model solution:

```
modelDupli1 :: [Int] -> [Int]
modelDupli1 = concatMap (replicate 2)
```

By extracting the expression (`replicate 2`) from the model solution and inserting it into the hole in the partial solution, we create the following combined solution:

```
dupli :: [Int] -> [Int]
dupli = concat . map (replicate 2)
```

We can see that no names that need to be renamed are introduced in the combined solution. Thus, we move on to testing the combined solution. Testing is done by QuickChecking the following property that is defined together with the model solution:

```
propModel f = property $ \xs -> f xs == modelDupli1 xs
```

By testing this property, we conclude that the solution indeed is valid. We then obscure parts of the generated solution to only show the next step that the student should take to reach it, which looks like this:

```
dupli = concat . map (replicate ?)
```

The following sections show how the algorithm works in more detail.

3.2 Working with Abstract Syntax Trees

To represent a Haskell program in a more structured way, the Haskell code is compiled into an AST (Abstract Syntax Tree). This AST is similar to that of GHC's but with a lot of excess information removed, such as source location data and typeclass declarations. The reason for creating our own AST is to reduce the size of the search space for the program synthesis and creating a level of indirection to not be reliant on a third party AST.

One of the more important data types in our AST is `Pat`, representing patterns, and a part of it is shown below:

```
data Pat
  = PList Pats
  | PLit Literal
  | PWildcard
  ...
```

Patterns are used when pattern matching in top-level bindings, such as `case` statements and `where` clauses. An example of this could be the following:

```
case ys of
  []    -> print "empty"
  (x:_) -> print x
```

Here we match on the list `ys` and if it is empty we print `empty`, otherwise, we match the first value of the list and print that.

Finally, most of the program logic, that is the functionality that converts the input into output, is encoded in **Expr**s and this is what we have focused our synthesis on. A part of this datatype is shown below:

```
data Expr
  = Hole HoleID
  | Lit Literal
  | Paren TExpr
  | Tuple [TExpr]
  | App TExpr [TExpr]
  | Var Name
  ...
```

In our AST, every **Expr** is wrapped in a **TExpr**, which contains the type of an expression. By storing information on the type of each expression we can reduce the complexity of the code later in the process. The code for the entire AST can be seen in Appendix A

```
data TExpr = TExpr GHC.Type Expr
```

In Figure 3.1 we show an example of how the expression `concatMap (replicate n) xs` is represented in a simplified version of our AST. Note that we omit the types in this AST to give an easier overview. In reality, every node is wrapped in a **TExpr** with the type of the expression. In this figure, **App** represents a function application. The figure is simplified a bit, and the list of expressions that **App** takes is replaced with the expressions directly. In the figure, the function **Var** `concatMap` is called with the arguments from its sibling sub-trees connected to the **App** node. **Paren** represents a parenthesis around the sub-tree below it. Finally, **Var** is a variable with the name from the node below it. The **Var** is also simplified a bit, and in reality, each name is wrapped in an **Ident**. A non-simplified version of this tree can be seen in Appendix A.

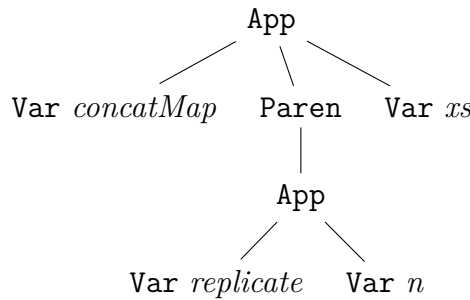


Figure 3.1: A visualisation of the AST representation of the Haskell expression `concatMap (replicate n) xs`. Note that we simplify it and omit the **TExpr**s to give an easier overview.

3.2.1 Transforming Haskell Code to AST

By using GHC as a library, (explained in Section 2.1.3) we can use the internal functionality of GHC. By limiting the compiler to only run the parsing and type-checking phases of the compiler, we can access the internal AST that GHC uses to represent the code after the type-checking phase has completed. After the type-checker has run, this AST contains type information about every expression and pattern of the compiled code.

We use the AST from GHC to create and type our own AST. To do this, we recursively go through the AST from GHC and for every relevant node in it, we create our own node based on it. For instance, the representation for an expression in GHC's AST is `HsExpr`. Each expression is transformed into our smaller representation called `Expr`. During this process, we wrap the expression in a `TExpr` datatype, that holds the type of the expression. The `Type` datatype is the exact same as GHC uses.

Using GHC's internal representation of types allows us to use some well tested and robust utility functions from the GHC library, such as `eqType` which checks if two types are equal. More importantly, the `Type` type in GHC can handle the full set of available types in Haskell, which we can use to our advantage when synthesising code.

3.2.2 Transforming AST to Haskell Code

We convert our AST to runnable Haskell code again for two reasons. First, we need to validate the behaviour of our synthesised ASTs using QuickCheck. However, direct evaluation of our AST was not implemented, since it requires a disproportionate amount of work compared to the benefits. Thus, we convert it into Haskell code again to compile and run it. Second, we show the generated feedback as code to the student. Therefore, we return to a format that can be read by both humans and machines, which in our case is pretty printed Haskell code.

Some information is lost during this transformation. Most notably, white space and the formatting of the text gets removed during compilation. This has the potential to confuse the student, but these changes should be minor. It should however be possible to extract this information from the GHC AST, but we did not implement this.

3.3 Synthesising code

From the typed AST, our program tries to synthesise code that fits into the holes left by the student. Before beginning the synthesis process, libraries, such as `Data.List`, sometimes needs to be imported to support the model solutions. To synthesise the solution, the program goes through the following phases (an illustration of the process can be seen in Figure 3.2):

1. Pre-processing the model solutions by eta-expanding them.
2. Combining the model solutions with the student solution.
3. Renaming the combinations.
4. Testing the renamed combinations.

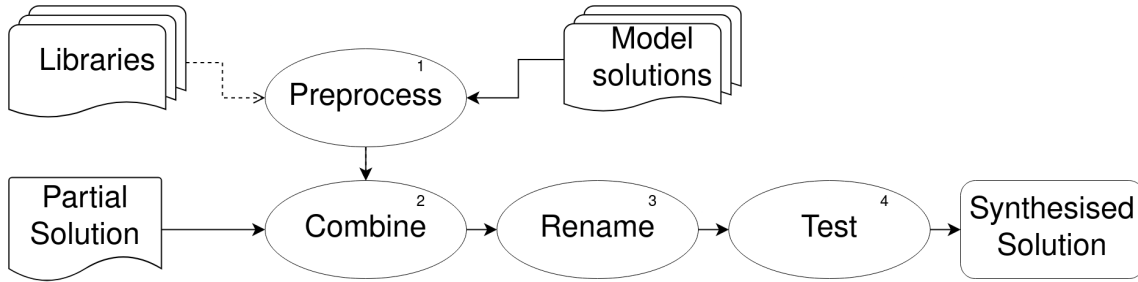


Figure 3.2: The phases of the program synthesis. First, we pre-process the model solutions, then we combine them with the partial solution. After that, we rename all undefined names and then test the potential solutions. Finally, we hopefully get a valid solution.

Now, assume that a teacher has defined the following model solutions for the exercise **dupli**:

```

modelDupli1 :: [Int] -> [Int]
modelDupli1 list = concatMap (replicate 2) list

modelDupli2 :: [Int] -> [Int]
modelDupli2 = foldr (\x xs -> x : x : xs) []
  
```

Furthermore, assume that a student submits the following partial solution:

```

dupli :: [Int] -> [Int]
dupli = foldr (\y ys -> ?) []
  
```

In this case we would expect to get a suggested next step guiding the student towards **modelDupli2** but with **x** and **xs** replaced with **y** and **ys** respectively:

```

dupli :: [Int] -> [Int]
dupli = foldr (\y ys -> y : y : ys) []
  
```

We now show how our approach comes to this result by explaining each phase of the synthesis. To simplify the explanation, we explain the process for one hole. The process for multiple holes is identical, but the combination phase is repeated for every hole. This means that the number of combinations increases exponentially with every hole.

3.3.1 Eta-Expanding the Model Solutions

The first phase of the synthesis process is to eta-expand the model solutions. We can see that only `modelDupli2` can be eta-expanded:

```
modelDupli2a as = foldr (\x xs -> x : x : xs) [] as
```

When the partial student solution and a model solution differ in form, THUPY might not succeed in finding a solution, even though with an eta-expanded model solution it might. Consider the following example:

```
-- Partial student solution
repli a xs = concatMap (replicate a) ?
-- Model solution
repli1 a = concatMap (replicate a)
-- eta-expanded model solution
repli1a a xs = concatMap (replicate a) xs
```

In this example `repli` is a partial student solution, while `repli1` is a model solution. `repli1a` is the eta-expanded version of the same model solution. In `repli` we would like the program to be able to suggest putting `xs` into the hole. However, since the model solution `repli1` does not contain `xs` or any other variables with that type we would not be able to figure this out. This is where the eta-expanded version of the solution, `repli1a`, comes in. This version has a list, `xs`, that could fit in the hole.

To ensure that we avoid this problem we always expand our model solutions. It might seem simpler to just transform the one student solution rather than all the model solutions. However, when we later need to show a next step, we want it to be a step from what the student submitted. Therefore, we think it is better to keep the student solution intact and change the model solutions instead.

3.3.2 Combining the Student Solution with the Model Solutions

The next phase in the process is combining the student solution with the model solutions. We want to substitute the hole with an expression of the same type. We, therefore, gather all the expressions from the model solutions and group them by their type. Next, we create a possible solution from each expression that has the correct type by inserting it into the hole of the student solution.

For our `dupli` example we find that the hole has type `[Int]`. We, therefore, pick all the expressions from the model solutions that have this type. We can see each of these expressions underlined below:

```
modelDupli1 list = concatMap (replicate 2) list

modelDupli2 = foldr (\x xs -> x : x : xs) []

modelDupli2a as = foldr (\x xs -> x : x : xs) [] as
```

Notice that we find duplicate expressions. For example, we find expression `xs` in `dupli2` and `dupli2a`. However, duplicate expressions do not affect THUPY’s ability to give a suggestion. It only affects the performance since it potentially has to test the same solution multiple times. However, the performance gain from removing duplicates might not be worth the overhead and we did not evaluate this further. Instead, we accept the duplicates to keep our proof of concept simpler.

Now that we have picked all the relevant expressions, we insert them into the student solution. We are not going to show the full list of resulting combinations here but limit it to two that are of interest in future steps:

```
dupli = foldr (\y ys -> as) []

dupli = foldr (\y ys -> x : x : xs) []
```

We know that the expressions we inserted into the student solution have the right type but as can be seen from the above examples many of the inserted names lack accompanying definitions. This is where the renaming step comes into play. We explain this further in the next section.

3.3.3 Renaming

In our `dupli` example from before, the renaming is based solely on variables being undefined. We can see that the only variable in scope with type `[Int]` is `ys` and the only variable with type `Int` is `y`. Therefore, `as` and `xs` are renamed to `ys` and `x` is renamed to `y`. This creates the following two `dupli` functions:

```
dupli = foldr (\y ys -> ys) []

dupli = foldr (\y ys -> y : y : ys) []
```

For this example, the need for and use of renaming is clear but probably raises some questions. We, therefore, explain the renaming phase in more detail.

When inserting an expression from one place to another, the values bound to the variables may change. Consider the following two solutions:

```
modelMyConcat first last = first ++ last
myConcat      last first = ?
```

Simply inserting the underlined expression from `modelMyConcat` into the hole in `myConcat` would not cause any type or naming problems, but the semantics have changed. The underlined expression will differ in behaviour since `first` in `modelMyConcat` is the first parameter and in `myConcat` it is the second function parameter. Therefore, the synthesised solution would concatenate the lists in the wrong order. The problem is that we do not know the original value that a variable represents in the inserted expressions. This means that we need to try all possible representations to find the correct one. We do this by first renaming and then later, testing which variable represents the correct value.

When deciding which variables to rename we look at it in two steps. First, we look at the names and check if any of them are defined in the Prelude or any imported library. Since these names are defined outside the solution, we skip renaming them.¹ Below, we show an expression from a model solution in isolation, which serves as an example of an expression that can be inserted into a hole. Here we find both `++` and `replicate` to be Prelude functions:

```
(\acc e -> acc ++ replicate a e)
```

Secondly, we check if any of the variables are bound in the inserted expression itself. These variables are also assumed to be used correctly since their definition is part of the expression itself. We, therefore, do not rename these variables either. We use the same example as above and the variables that are defined inside the expression itself are underlined together with their corresponding definition.

```
(\acc e -> acc ++ replicate a e)
```

The remaining variables need to be renamed. We try to rename them to any variable defined in the student function that fits the type. If no such variable exists and the name cannot be bound, the function cannot compile, and the synthesis cannot continue from this combination. It is possible that a name can be renamed to more than one variable. In the following listing we demonstrate this:

```
-- Model Solution
repli :: [Int] -> Int -> [Int]
repli as a = foldr (\acc e -> acc ++ replicate a e) [] as

-- Original function with a hole
repli :: [Int] -> Int -> [Int]
repli xs n = foldr (\ys y -> ys ++ ?) [] xs

-- Original combination of model solution and partial solution
repli :: [Int] -> Int -> [Int]
repli xs n = foldr (\ys y -> ys ++ replicate a e) [] xs

-- Renamed combinations
repli :: [Int] -> Int -> [Int]
repli xs n = foldr (\ys y -> ys ++ replicate y y) [] xs
-----
repli :: [Int] -> Int -> [Int]
repli xs n = foldr (\ys y -> ys ++ replicate y n) [] xs
-----
repli :: [Int] -> Int -> [Int]
```

¹There is a known bug with this approach when the student solution defines a variable in the function with the same name as a prelude function used in the inserted expression. This will likely cause a mismatch in what type the synthesis thinks the expression has and what the compiler will evaluate the type to be, which in turn will cause the synthesis to fail. A simple fix to this is to check if a prelude name is also a variable defined in the student function.

```

repli xs n = foldr (\ys y -> ys ++ replicate n y) [] xs
-----
repli :: [Int] -> Int -> [Int]
repli xs n = foldr (\ys y -> ys ++ replicate n n) [] xs

```

Here we can see that `replicate` is defined in the Prelude and is not renamed. The only variables left to rename are `a` and `e`. Since both of these variables are of type `Int`, we rename each of them to both `n` and `y`. This creates 4 renamed solutions, which are the valid ways to rename the expression. However, at this point, we do not know which (if any) of the renamed solutions are correct. Therefore we test all of them.

3.3.4 Testing

Once the program has generated potential solutions where all types match and there are no unbound variables, we check if they have the desired behaviour. For each exercise, the intended behaviour is specified by teacher-defined QuickCheck properties. The tests will either succeed or fail based on if the solution passes the QuickCheck tests. If a solution passes all the tests it should be a valid solution. (If the test cases were not comprehensive enough, there might still be counterexamples.) Once a valid solution is found, we terminate the testing and use the found solution when generating the suggested next step. In the case of our `dupli` example, only one property exists:

```

propModel f = property $ \xs -> f xs == modelDupli1 xs

```

We can see that the property checks if the synthesised function is semantically equivalent to `modelDupli1`.

To test our potential solutions that are in an AST form, we convert them back to Haskell code. To evaluate the code using GHC it is easier to first store it in a file. We store each solution separately in files along with the QuickCheck tests. This file is compiled and evaluated by GHC. An example of such a file is shown below:

```

properties :: ([Int] -> [Int]) -> Property]
properties = [propModel]

propModel f = property $ \xs -> f xs == modelDupli1 xs

dupli = foldr (\y ys -> ys) []

run = quickCheckWithResult
      stdArgs {chatty = False}
      (within 1000000 (conjoin (map ($ dupli) properties))))

```

The first combined and renamed solution that passes the test is:

```

dupli = foldr (\y ys -> y : y : ys) []

```

This is the solution that we said that we would get in the beginning of Section 3.3.

Timeout: If a test runs for too long, it is set to fail. This is done to ensure that the program does not contain an infinite recursion, and to restrict the time it takes for THUPY to give feedback. The timeout duration is set to 1 second (10^6 microseconds). This timeout is somewhat excessive and in our testing, almost any (non-infinitely recursive) solution could be tested in at most ~ 100 milliseconds on modern hardware.

Figure 3.3 shows a graph over our testing of how much time it took to test 550 partial solutions timed in microseconds. In this graph, everything above 1 second timed out. All the timed-out solutions contained expressions that caused an infinite recursion in the function. We believe the reason that the timed-out solutions was measured as ~ 4 seconds was because of overhead from terminating the running function.

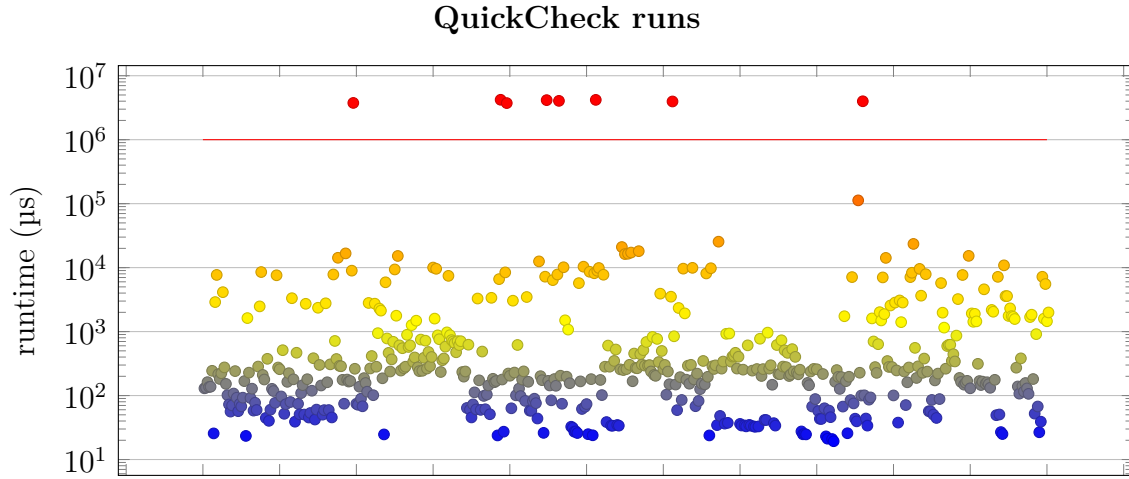


Figure 3.3: How long each run of QuickChecking a potential solution took, measured in microseconds. In total, 550 potential solutions were tested.

Ordering the Tests: We only use the first successful solution and this can cause problems. There is a chance that the first valid solution found is overly complicated and that there exists a less complicated solution. One way to mitigate this is to test the solutions in order of increasing complexity. We chose to order the inserted expressions by their depth, i.e. the height of the AST. By testing the solutions in this order, we know that the first valid solution to pass the tests should be the “least complex” one. If more than one solution has the same depth, there is no relative ordering between the solutions.

We could test all solutions and decide which is the best one after all valid solutions have been gathered. However, doing this will severely impact performance by making THUPY handle sometimes thousands of more potential solutions. Note that the ordering of the solutions, in reality, happens earlier in the pipeline, namely after the combining step. However, how the results are ordered does not have an impact

on any step in the pipeline before the testing step, thus it is not explained until now. The solutions are ordered in the combination step since there are the fewest expressions to order at that point. Furthermore, the ordering of the solutions is kept throughout every phase.

Batching the Tests: Compiling each potential solution individually has a lot of overhead. This means that the response time often becomes longer than necessary. We, therefore, opted to batch the testing of the solutions to minimise this overhead. The batching works by writing multiple solutions and QuickCheck calls into the same file. Furthermore, we add a function that tests all the solutions and returns the first one that succeeds. An example of such a file for our `dupli` example is shown below.

```

properties :: [(Int -> Int) -> Property]
properties = [propModel]

propModel f = property $ \xs -> f xs == concatMap (replicate 2) xs

dupli1 = foldr (\y ys -> ys) []

dupli2 = foldr (\y ys -> y : y : ys) []

funcs :: [(Int, IO Result)]
funcs = [(1,
  quickCheckWithResult
    stdArgs {chatty = False}
    (within 1000000 (conjoin (map ($ dupli1) properties))))),
  (2,
    quickCheckWithResult
      stdArgs {chatty = False}
      (within 1000000 (conjoin (map ($ dupli2) properties)))))]

run :: [(Int,IO Result)] -> IO (Maybe Int)
run [] = return Nothing
run ((i,f):fs) = do
  res <- f
  case res of
    Success{} -> return $ Just i
    _         -> run fs

```

The batch size was decided by testing a few different values on two partial solutions, each run five times. The values tested and their runtime can be seen in Table 3.1. The functions `repli` and `slice` from Listing 2 were used for this testing. From the table, it is easy to see that batching potential solutions decreases runtime compared to running them one by one. For our case we found a batch size of 25 or 50 to be good enough. A static batch size of 50 entries was chosen.

```

repli :: [Int] -> Int -> [Int]
repli xs a = foldl (\acc e -> ? ++ ?) [] xs

slice :: [Int] -> Int -> Int -> [Int]
slice xs i j = map snd
                $ filter (\(x,_) -> ? && ? )
                $ zip [1..] xs

```

Listing 2: The two programs **repli** and **slice** measured in Table 3.1

Batch Size	repli (s)	slice (s)
1	7.6	22.2
5	2.3	6.0
10	1.9	3.7
25	1.5	3.0
50	2.0	2.5

Table 3.1: A table over the average running-time (measured in seconds) for partial solutions of **repli** and **slice** from Listing 2 with batching size 1, 5, 10, 25, 50. Each function and batch size combination was run 5 times.

For optimal performance, the batch size should be exactly equal to the number of tries it takes to generate a correct solution. If the program batches too many solutions the synthesising algorithm will have to compile more potential solutions than necessary. If the program batches too few, we have to run the compilation many times. Compilation has a high overhead, so compiling a small program and a large program takes similar time.

3.4 Generating Feedback

From the synthesised code, we want to present a next step to the student. The aim is for the student to be able to continue to a solution using our provided next step. We do this by providing the student with a small incremental next step toward the solution. How the feedback is presented could be configurable in a future version.

3.4.1 Formatting the Hints

Once THUPY has found a solution, we insert holes into it so that it only reveals the suggested next step for the student. Consider the following example of a student input, generated solution and the formatted hint:

```

-- Student input
dupli :: [Int] -> [Int]
dupli = foldr (\y ys -> ?) []
-- Synthesised solution

```

```
dupli :: [Int] -> [Int]
dupli = foldr (\y ys -> y : y : ys) []
-- Suggested next step
dupli = foldr (\y ys -> ? : ?) []
```

This shows the next step from the continuous example in the previous section after the synthesis process has produced a result. In this case, our algorithm suggests adding a list constructor (`:`)

In general, to suggest the next step, we first look at the expressions that were inserted to create the solution. The root expression is kept while the child expressions of it are replaced with holes.

4

Evaluation

In this chapter, we describe our procedure for evaluating how well THUPY complements and compares to Ask-Elle. We use a set of previously stored attempts at solving exercises from Ask-Elle. The set of attempts can be divided into two categories, one where Ask-Elle could provide feedback and another where it could not. For each attempt, we evaluate if THUPY can generate feedback, and how long it takes to either fail or succeed. Evaluating these two categories tells us how well we complement Ask-Elle, and how well we compare directly to it. Finally, we analyse and discuss our results.

4.1 The Dataset

The dataset that we use to evaluate THUPY consists of 18139 entries from September 2015 until February 2021 and was gathered from an instance of Ask-Elle running on servers at the University of Utrecht. The dataset is not the same dataset as in any of the previous Ask-Elle papers. Out of these 18139 entries, 8370 describe student attempts submitted to Ask-Elle. The remaining entries are logs of requests, such as listing the exercises which are not relevant to us. Many of the attempts were syntactically faulty, did not type-check, or had some other problem caught by Ask-Elle. After these invalid entries were filtered out, the dataset contained 1943 compilable entries.

From the classifications described in Section 2.4, we are interested in the *discarded* and *model* attempts. The attempts classified as *model* are where Ask-Elle gave feedback or deemed the attempt finished. The attempts classified as *discarded*, on the other hand, are where Ask-Elle cannot give feedback, but finding a next step might still be possible. To better suit the context of this thesis, we call the *model* attempts *on-path* and the *discarded* attempts *drifted*. Since the finished solutions are not relevant in our case, these 715 entries are removed. In total, we use 1228 entries, out of these, 299 were categorised as drifted and 929 as on-path.

4.1.1 Compatibility

Many of the entries in the dataset are missing type-signatures for the defined functions. This means that the type of a function is determined by the type-checker using type inference which can result in the function being defined with polymor-

phic types. THUPY does not support polymorphic types, which was one of the limitations we listed in the introduction. Consider the following:

```
dupli = concatMap ?
```

In this case the type-inference in GHC (which we used to compile the entries for THUPY) would give the function the polymorphic type `[a] -> [b]` and the hole in the expression the polymorphic type `a -> [b]`. We however, have defined monomorphic types for the model solutions (for `dupli` it is `[Int] -> [Int]`) which are not equivalent to these polymorphic types. This means that these entries are not compatible with THUPY.

To allow us to run THUPY on this dataset we need to give these entries the same monomorphic type as the corresponding model solutions. If an entry has a given type signature we simply use that one since we consider it part of the student’s implementation. For each entry with no explicit type-signature, we add a monomorphic type signature that matches the model solutions. This allows us to match types between the partial student functions and the model solutions.

4.2 Accuracy

To evaluate how well THUPY complements Ask-Elle, we apply it to all the drifted attempts. Additionally, we apply it to all the on-path solutions to evaluate how the program synthesis method compares to Ask-Elle directly.

Sometimes entries take considerably longer to process than what we in the introduction described as “reasonable time”. This usually happens when the student leaves more than three holes in their solution causing the number of possible solutions to grow to an unmanageable size. If a valid solution does not appear until late in this list (or not at all) we are likely to run out of time. To handle this we set a somewhat arbitrary timeout duration of 30 seconds for testing any given entry. After this, the entry is considered failed and marked as timed-out. We believe that beyond 30 seconds would be considered unreasonable by most people.

4.2.1 Result

THUPY is able to provide feedback to **21%** of the drifted entries. For the examples where Ask-Elle had already succeeded in providing feedback, THUPY generates feedback for 35% of the attempts. Combining these numbers, we conclude that THUPY is able to provide feedback for 32% of applicable entries. Some examples of generated solutions from this evaluation can be found in Appendix D.

Evaluation Result Categories			
	Drifted	On-Path	Combined
Successful	63 (21%)	329 (35%)	392 (32%)
Not Found	231 (77%)	556 (60%)	787 (64%)
Timed Out	5 (2%)	44 (5%)	49 (4%)
Total	299	929	1228

Table 4.1: The number of entries for each category and dataset.

4.2.2 Discussion

The results are promising at first glance, given that THUPY provides feedback for 21% of entries where Ask-Elle cannot. We think there is reason to believe that this result can be even better. We limited our implementation to synthesising expressions, which means feedback will not be generated if there are holes in patterns. Furthermore, the drifted category contains entries that are impossible to solve by only substituting a hole.

In addition to the previously mentioned entries, there are a number of other entries in the on-path category that THUPY cannot provide feedback for. In this category, Ask-Elle has already shown that it is possible to provide feedback. Therefore, there is no reason why our synthesis approach should not be able to find a valid solution. We believe that this is caused by flaws in the implementation of THUPY.

Overall, the results show that a combination of Ask-Elle and THUPY performs better than any one of the two individually.

Holes in Patterns: Since we did not implement synthesis for holes in patterns, THUPY cannot provide feedback to some fairly “simple” entries in the dataset. An example of such an entry is the following:

```
dupli ? = concatMap (replicate 2) ?
```

In this example, we have a partial student solution with a hole as the function parameter as well as one at the end of the expression. Here we expect a valid solution to use the function parameter like this:

```
dupli xs = concatMap (replicate 2) xs
```

However, in the partial student solution, the function parameter has not been given a name and is therefore ignored. Creating a valid solution by only replacing the hole in the expression is therefore not possible. In this case, by implementing support for synthesis in patterns we could replace the hole in the function parameter with a variable that the expression synthesis can use.

We count the number of entries with holes in patterns using the following two regexes, written in Python style. These match holes in patterns and are applied to

4. Evaluation

every line in an entry.

```
# All function bindings with at least one hole instead of a parameter
".*\?.*=.*"
```

```
# All lambda functions with at least one hole instead of a parameter
"\( *\?.*->"
```

These entries account for 17% of the *drifted* dataset and 21% of the *on-path* dataset. If we were to disregard these entries, we would succeed to suggest a next step for $\frac{63}{299-53} \approx 26\%$ of entries in the *drifted* category and $\frac{329}{929-201} \approx 45\%$ in the *on-path* category, compared to the original 21% and 35% respectively.

Entries Without Parameters Just like there are a number of entries with holes in patterns, there are also a number of entries with no parameters at all. In this case, the problem is that most of the model solutions use their parameters in their expressions. Using parameters from the model solutions that are never introduced in the partial solution causes the renaming phase of the synthesis to fail.

To count the entries without parameters, we use the following regex:

```
# All functions with a function name, no parameters and only a hole
# on the right hand side
"^\\w+ *= *\\?\\$"
```

There are in total 182 entries with no parameter bindings and only a hole on the right-hand side in the *on-path* category. There might be more similar problems in the dataset not caught by the regex we used to detect these entries. It should also be noted that some of these entries might be solvable by us. Since we do not know how many of these parameterless entries are solvable, we choose not to draw any conclusion from this and instead just note their existence and how many we could find.

Impossible Entries: There are also a number of entries in the dataset that are impossible to synthesise a solution from. One such example is:

```
dupli xs = map ? xs
```

Since `map` operates on every element in the list, the output of `map` must be equally long as its input. However, one key property of `dupli` is that it should return a list double the length of the original list. Therefore it is impossible to find a solution from this input.

These entries have the type signature that the exercise requires but the logic does not allow for any possible way to synthesise a solution with the expected properties. Unfortunately, we found no easy way to determine how many entries are impossible to suggest a next step for.

4.3 Performance

To evaluate if THUPY was able to generate feedback in “reasonable” time, as stated in the introduction, we applied it against the same dataset as before, using the same categories. For each entry, we measured the time it took for THUPY to provide feedback.

4.3.1 Result

The result of these tests can be seen in Table 4.2¹. This table counts every timed-out entry as a failure taking 30 seconds to complete. While the number of timed-out entries was small (less than 5% of all entries), it has a large impact on the average runtime of THUPY. However, looking at figure 4.1, we can see that more than 60% of all entries finished in less than 1 second and more than 80% finished in less than three. Figure 4.2a shows the runtime for each entry in the the drifted category and Figure 4.2b the same for entries categorised as on-path. In these figures, the blue squares are entries that THUPY succeeded in providing feedback for, and the red triangles are entries where it failed.

Metric	Drifted (s)	On-path (s)	Combined (s)
Avg. time to succeed	2.14	2.24	2.22
Max time to succeed	17.33	26.27	26.27
Avg. time to fail	2.08	4.30	3.67
Max time to fail ²	17.68	27.72	27.72
Avg. runtime	2.09	3.57	3.21

Table 4.2: The table shows a list of different metrics for running on drifted, on-path, and both categories combined. The figures in the table are running time measure in seconds.

¹These times were achieved using an Intel Core i7-7700K@4.60Ghz, with 16GB of RAM on Linux 5.10.36-1-lts.

²Not including the timed out entries

4. Evaluation

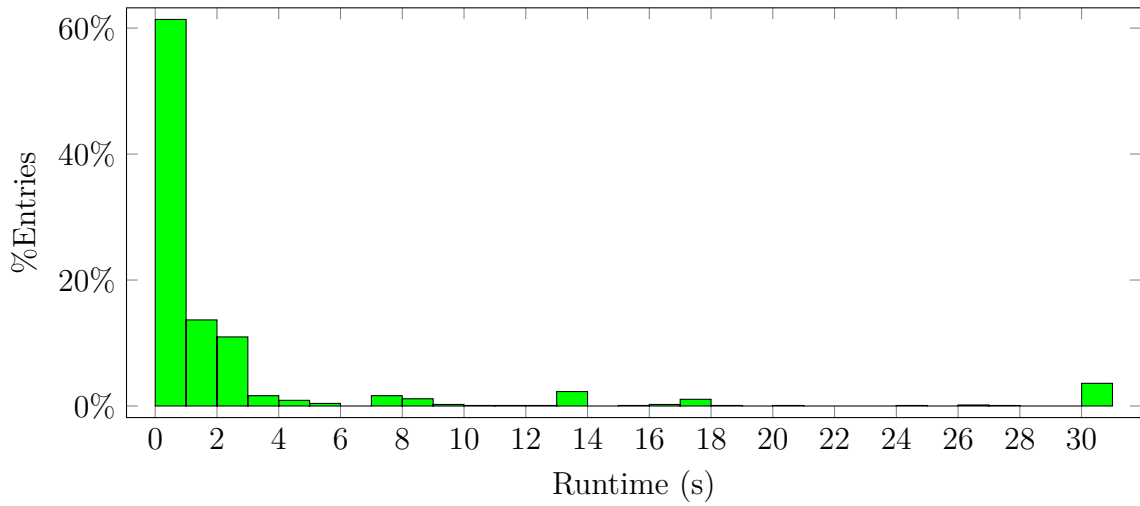
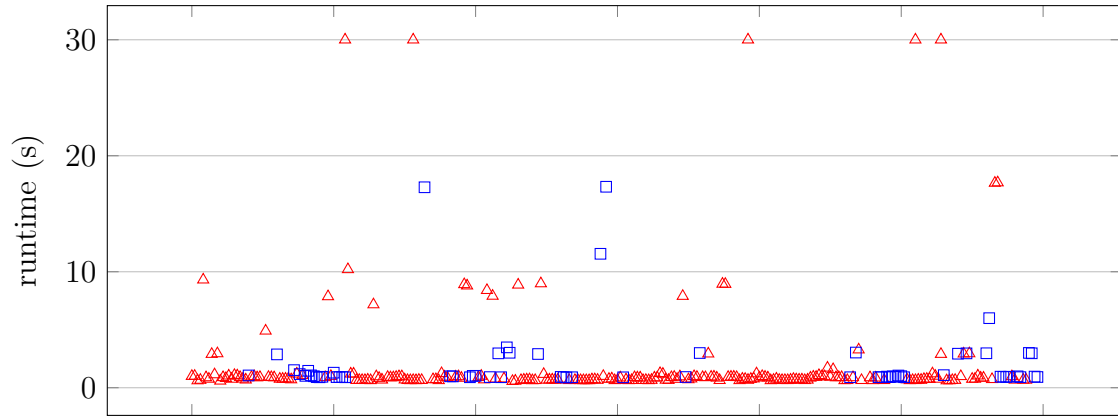
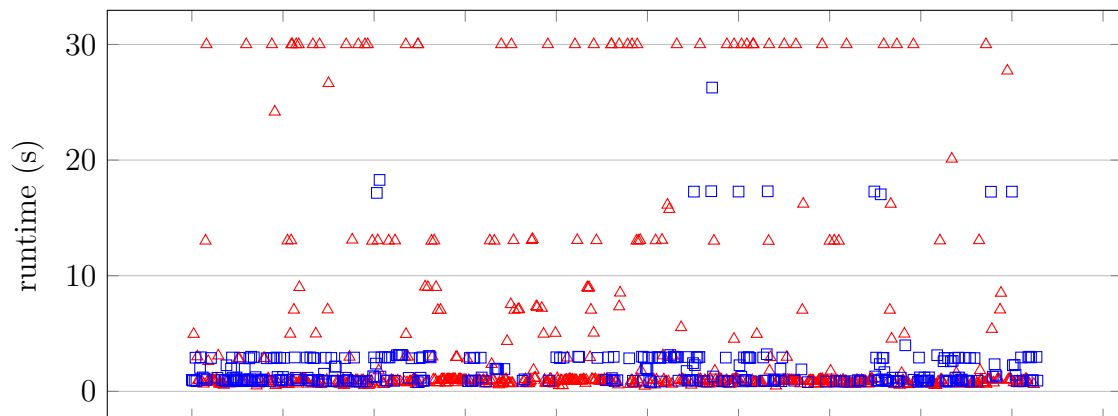


Figure 4.1: The frequency of runtimes for all entries.



(a) Runtimes for the *drifted* category.



(b) Runtimes for the *on-path* category.

Figure 4.2: The runtime for every entry. The different plots depict the different categories. The blue squares represents a success and the red triangles represent a fail.

4.3.2 Discussion

We found the runtime of THUPY to be reasonable for most entries. We do not think most users will be too frustrated by waiting a few seconds to receive feedback. Furthermore, THUPY is a proof of concept and performance can be improved in the future. There are however entries, both failed and succeeded in the dataset that took abnormally long to run and some successful entries took close to the timeout of 30 seconds to finish. Thus, there is a possibility that some entries that timed out would eventually have succeeded and provided feedback.

The timeout was already set fairly high and from Figure 4.2 we see that it is a small portion of successful runs that take more than 10 seconds ($\sim 3\%$). Later, we discuss such an entry, and why we think it took a long time to finish. From a user interaction perspective, having THUPY timeout after closer to 10 seconds might be more reasonable, even if a few more attempts would not receive feedback.

Figure 4.2 and the averages from Table 4.2 shows us that the average time to succeed is very similar between the two categories (2.24 and 2.14 seconds). However, the average time to fail was noticeably longer for the solutions categorised as on-path compared to the ones categorised as drifted (4.30 and 2.08 seconds). One reason for the difference is that the portion of entries that timed out in the on-path category is larger than in the drifted category.

Example of Entry that Took Long: The evaluation of the runtime also showed that there are examples where the runtime becomes considerably longer than desired and might cause the user to give up. When evaluating THUPY, one of the entries that took a long time to finish (~ 17 seconds), while still succeeding was the following attempt at `elementat`. The function `elementat` should find the element at a certain index in a list.

```
elementat :: [Int] -> Int -> Int
elementat (x:_) 1 = x
elementat (x:xs) n = if True then elementat ? ? else ?
```

For this input, the student solution is quite similar to a model solution that looks as follows:

```
elementat :: [Int] -> Int -> Int
elementat (x:_) 1 = x
elementat (_:xs) k = elementat xs (k - 1)
```

To synthesise a solution to this, we need to fill three holes. This in itself can take quite a lot of time since the number of possible solutions increases exponentially with the number of holes. The holes are also quite general in their types, the first hole is of type `[Int]`, while the two other are of type `Int`. This means that there is a high number of possible expressions that can be inserted into each of the holes. More specifically for this input, there are 5776 solutions for the testing phase to try (the average entry has ~ 80). Another problem is most likely our heuristic

approach which sorts the possible solutions by the depth of the inserted expressions. In this case, there are many expressions with low depth to try before reaching the expression $(k - 1)$, which is the second deepest expression in the model solution of type `Int`. There is most likely a mix between these two problems that cause the long runtime.

5

Discussion

In this chapter, we discuss some of the techniques used in this thesis. Furthermore, we discuss some possible future work. Finally, we also address some societal, ethical and ecological aspects of our project and the techniques behind it.

5.1 Comparing Program Synthesis to Strategies

Both the program synthesis and the strategies approach have their strengths and weaknesses. The strategies approach, described in Section 2.4 is strict and operates under the presumption that a student has approached the problem according to one of the supplied model solutions. The most obvious upside to this approach is that the tutor forces the student into a solution that a teacher has deemed to be appropriate. This makes it possible for a teacher to focus on a particular topic. For example, if the current topic in a course is list-comprehensions, the teacher might only enter model solutions that use a list-comprehension approach. However, the major downside of this rigidity is that it might be too strict, as we highlighted in some of the examples in this thesis.

The program synthesis approach has different characteristics. Compared to Ask-Elle, we are able to provide feedback for more diverse partial solutions and can cope with diversions from the pre-defined solutions in ways that Ask-Elle cannot. However, we are not able to give as extensive feedback compared to Ask-Elle. Ask-Elle can for instance also use annotations to explain the next step more in-depth. Thus, if both our complement and Ask-Elle can find a next step, choosing Ask-Elle would be preferable as it can provide different kinds of feedback and not just a suggested next step. We think that the best application of THUPY is as a fallback to Ask-Elle when it is unable to provide feedback to an attempt.

Furthermore, comparing our limited search space method to a more traditional program synthesis approach, with more freedom in the expressions it can synthesise, has both its pros and cons. Using the model solutions as the search space allows teachers to guide the synthesis towards solutions that they deem to use good practices. However, this requires a teacher to provide model solutions to the exercises and makes it more demanding to create new exercises. Additionally, the smaller the search space is for a program synthesis algorithm, the faster it runs. The runtime is also impacted by how efficient the algorithm is, so while our search space might

be smaller than that of other algorithms, our runtime might still be slower.

5.2 Expression-Synthesis and Pattern-Synthesis

For a student, determining the structure of an expression is, in our experience, often harder than constructing a pattern. Therefore, we believe that being able to synthesise expressions is more important than patterns. Furthermore, in the exercises that Ask-Elle and we use, patterns are often small and simple. They often consist of a simple parameter or a constructor wrapping one or more variables. Expressions, on the other hand, tend to be more complicated. Finding a suitable expression and logic is often the central part of programming. This is further supported by how few solutions have holes in patterns compared to the total number of entries in the datasets. The full dataset has a total of 254 entries with holes in patterns, which corresponds to about 21% of all entries. Contrarily, there are 1039 entries with holes in expressions. This corresponds to about 85% of entries. This seems to indicate that students have more problems determining the structure of an expression compared to a pattern. Thus, we focused on being able to synthesise code for holes in expressions and did not implement synthesis for holes in patterns.

5.3 Handling GHC

We decided to use GHC as a library to handle much of our types and compilation. The library essentially exposes the inner functionality of GHC and is a massive library. Unfortunately, its documentation is a bit lacking and most functions are not documented in any detail. The GHC and Haskell wiki explains, among other things, how to run each phase of the compiler. But for things such as how the typing system works and how to interact with it, we found little documentation except for comments in the code and very basic explanations for a few functions. Thus, we found it to be user-unfriendly, especially to developers that are not familiar with the library.

5.3.1 Type Equivalence & Variable Substitution using GHC

We rely heavily on GHC for checking equivalence between two types since it handles most of that logic for us. Haskell uses a very complex type system and checking if two variables of polymorphic types **a** and **b** are substitutable is non-trivial. To be able to substitute a variable of type **a** with a variable of type **b**, both types must share the same constraints in their context.

In our case, we have two different contexts: one from the partial solution and another from a model solution. This makes it even harder to decide if two polymorphic types are substitutable. A very simple example of how type equivalence and type substitutions are complex is the following: A hole **?** has type **Num a**, and an expression *e* that we try to insert when synthesising has type **Int**. Since **Int** (in the default context) has an instance for **Num**, it is possible to insert *e* into the hole. However,

we need to look up what types `Int` has instances for in the current environment. As the context of the current program changes, even a simple type like `Int` can derive different types and be subject to different constraints. The easiest way to calculate all constraints of a variable and its type is to let a library or compiler do it for us. GHC should have support for this, but we found extracting this information to be hard and would require rewriting a large portion of THUPY.

5.3.2 Coupling to GHC

Many of the features and functionality of THUPY are closely connected to GHC. Both when parsing and testing the code we use both GHC's ability to compile Haskell code (and its AST, which we convert to our custom one). The fact that the coupling is so strong makes it difficult to replace GHC for another Haskell compiler, but it should be doable as long as it is a compiler with the required functionality.

5.4 Future Development and Improvements

We believe that our results show that using program synthesis to complement Ask-Elle is a valid idea. We also think that this concept can work for other digital tutors, and not just Ask-Elle. It would be interesting to evaluate program synthesis as a complement to strict tutors in general. Furthermore, it would also be interesting to evaluate the quality of the feedback, by performing field studies with students.

In Section 4.2 we discuss how potential future features could improve the results of the evaluation. We think it would be interesting to continue this research by evaluating a more sophisticated complement to Ask-Elle that uses either polymorphic types, pattern synthesis, or both.

Our program synthesis uses only the expressions in model solutions as its search space. It would be interesting to see how well feedback could be given using program synthesis if the algorithm could use the entire Haskell language and its base libraries as search space.

In our evaluation, we found impossible entries that have no way to fulfil the properties defined for the function. Being able to tell the student that their solution has this flaw, could reduce the time a student spends attempting an approach that will never work. It would be interesting to see if it is possible to detect these impossibilities.

5.5 Societal, Ethical and Ecological Aspects

During the project, some thoughts regarding societal, ethical and ecological aspects came up. These mainly concerned digital tutors and program synthesis, and how they can affect the world around us. These thoughts are mainly theoretical and we do not think that the work in this thesis will directly affect any part of society.

5.5.1 Digital Tutors

This thesis was written during a pandemic. This pandemic has shown the importance of good tools when learning from home. It has also shown that digitalised solutions can offload human tutors when teaching from a distance. One example of this is digital and recorded lectures. We think that programming tutors can both help students and teachers. Students can get dynamic feedback on what they are doing even when no human is present. A teacher on the other hand can let a digital tutor do most of the heavy lifting, allowing the teacher to focus on the students who need the most help.

By being able to teach and learn remotely to a greater extent, the need for travelling to school and back could be reduced. This, in turn, can permit more free time for both students and teachers, but also reduce the climate impact caused by travelling.

We think it will be hard for a digital tutor to replace a real teacher. For feedback to be as effective as possible, it should fill in the gaps between what the student understands and what the goal is for the student to understand [35]. What a student already understands and does not understand differs between students. Thus, the process of providing feedback should be customised to fit each student [35]. A teacher can tailor their feedback based on how much knowledge the student has in the area. A digital tutor can probe and try to estimate how much knowledge a student has based on the code they have written, and on how they reason about their code. However, we have not found a tutor that tailors its feedback in this way, and think that most students prefer answering questions verbally to a human over interacting with a machine. We do not think that digital tutors will replace a real teacher anytime soon, but at most aid them in their work.

THUPY (and Ask-Elle) could be manipulated by a student to give the entire solution away by repeatably just replacing the hole with the feedback given in the previous step. By iteratively doing this, a student is led through the solution step by step until it is complete. If they did this with intending just to skip through the exercise, the student would most likely not learn anything, or at least not as much as it would have if they had tried to find a solution themselves. However, this can also be used as a way to see the entire derivation of a solution and to learn from it.

5.5.2 Program Synthesis

Program synthesis algorithms can produce whole programs from a specification and could theoretically replace writing code. However, at its current state, program synthesis requires a very precise specification. In our limited experience, providing a precise specification can be hard. We think program synthesis will at most become a part in a software developers toolbox, similar to a good IDE. Many IDEs offers help in suggesting which functions and variables are in scope, and in the future, these suggestions might be extended with suggestions generated from program synthesis.

In a way, writing code in high-level languages can be seen as the program synthesis specification and the produced machine code as the generated program. Nonetheless,

we think that even if program synthesis would become powerful enough to generate whole programs, the work of a software engineer would shift to producing precise specifications instead.

6

Related Work

The fields of both program synthesis and digital tutors are large. In this chapter, we describe some projects and papers that are similar to ours.

6.1 CodeQ

CodeQ is a programming tutor, which like Ask-Elle provides a set of exercises. These exercises are for the programming languages Prolog and Python. Feedback is provided by showing implementation plans, hints and by running tests on the program. Hints can be provided explicitly by a teacher for common errors or automatically using rewrite rules or code patterns. These rewrite rules are essentially transformations on a student's program, that should bring the student's solution closer to a correct solution. CodeQ tracks every student's progress and their attempts at solving an exercise. The system then tracks how the student rewrote a fragment of their code to reach a program closer to a solution. These tracked changes can then be used to transform another student's program. If the transformed program is closer to a solution than the previous program, feedback is provided to the student based on the rewrite transformation. CodeQ keeps doing these transformations until it finds a solution or it times out. Code patterns also use machine learning on previous student attempts, this time to find patterns in submissions that corresponds to correct or incorrect solutions [36].

CodeQ is quite similar to our project in that it can automatically generate feedback based on a student's input. One major difference is that CodeQ uses student attempts to generate feedback, while we only base our feedback on teacher supplied model solutions. This can mean that as long as a teacher provides a good model solution, THUPY might guide the student toward a better solution than CodeQ can. However, it also means that CodeQ can improve with time and that the suggestions can become more sophisticated as time passes. Another major difference is that we only provide feedback to locations that the student has explicitly marked in their code, while CodeQ tries to improve and work on the entire program that the student has written [37].

6.2 Hazelnut Live

Hazelnut Live, by Omar et al. [38] is an online programming editor built to be able to work with incomplete programs in the programming environment Hazel. The environment was developed by the same researchers as Hazelnut Live [39]. Like our complement, it uses holes to mark a part of the code that is not yet implemented. These holes are treated as arbitrary expressions with appropriate types so Hazelnut can continue evaluating the program around the hole. Hazelnut can continuously provide feedback to the user by dynamically evaluating the code as it is being written. An example from the paper is $(30.0 * hw + _)$, where `hw` is a variable equal to 88.0, 76.0 or 93.0 in three different cases. For these examples Hazelnut will evaluate the expressions to $(2640.0 + _)$, $(2280.0 + _)$ and $(2790.0 + _)$ and show these results live to the user. Hazelnut can also ignore ill typed expressions and do the same evaluation around them as it can around holes [38]. Hazelnut will evaluate the type needed for each hole, and tell the user. Furthermore, it can use the type to suggest which variables can be inserted into the hole [38].

Like our complement, Hazelnut Live uses typed holes as a key element in how they generate feedback to a user. It also tries to aid a user in formalising and constructing their intended program. However, where we (and Ask-Elle) try to guide a user to a solution, Hazelnut instead supplies live supplementary information about the code. Hazelnut can in many ways be regarded as an improvement to code editors and IDEs, but their approach to giving supplemental information about holes in a users code is in many regards close to our method of supplying feedback to exercises. This feedback is given directly on the student's code, without the need to utilise model solutions. Furthermore, Hazel operates on a language built around typed holes, while in our case we applied it to Haskell which is a more general language.

6.3 MagicHaskeller

MagicHaskeller is a program synthesis library written in and built for generating Haskell code, it was developed by S. Katayama [40]. To generate an expression the user needs to provide a boolean function that specifies the properties of an expression, this function resembles a QuickCheck property. The library generates a stream of expressions that satisfies the specified input-output examples. The expressions are streamed in order of small to large, and every expression in the stream is tested until one is found that fulfils the specified properties.

The approach of generating type correct expressions and testing them one by one is pretty close to what we are doing in THUPY. Where we only test a relatively small number of expressions for each hole and each expression is generated before testing, MagicHaskeller generates a very large number of possible expression as a stream. Both THUPY and MagicHaskeller use the same method of ordering how the expressions are tested, from small to large, in the case of MagicHaskeller, this is also the order that the expressions are generated. While THUPY only uses the

expressions from the model solutions, MagicHaskeller uses the entire Prelude (and `Data.List`) to generate expressions [41].

6.4 Hoogle+

Hoogle+ by James et al. [42] is an online synthesis engine for Haskell. Unlike what the name suggests, it is not related to the Haskell API search engine *Hoogle*. The program is a web-based tool for generating Haskell programs by specifying a set of input-output examples and optionally a type. Hoogle+ generates programs that match the specified type and the behaviour defined by the input-output examples. Like our complement, Hoogle+ uses a property-based testing tool, in their case SmallCheck, to test if a synthesised program has the behaviour specified by its input-output examples [42].

Underneath, Hoogle+ uses the TyGAR type-search algorithm [42]. TyGAR combines functions from different libraries to construct an expression with a type-signature specified by the user [43]. Furthermore, TyGAR handles polymorphic types [43].

Hoogle+ uses a much larger search space than THUPY and can use any function from the standard Haskell libraries for its synthesis [42]. Our synthesis is more targeted, and THUPY uses only expressions from the model solutions for synthesising code. Furthermore, while Hoogle+ is an online tool to demo program synthesis, our proof of concept was built to show if and how program synthesis can complement Ask-Elle.

6.5 Program Synthesis Libraries in Haskell

Program synthesis has not been an active topic in the Haskell community, and there are only a few program synthesis libraries and projects for Haskell. We investigated if we could use two of the more prominent projects, which all potentially synthesise code for us. We ended up not using any of them, for the reasons listed below.

- The library we looked into the most was [MagicHaskeller](#). This library showed a lot of promise when initially researching it since it promised ease of use and seemed to fit our need. However, MagicHaskeller is rather strict in what it needs to synthesise code, and we do not know what properties holes should have to fulfil the finished program, only their types. Furthermore, MagicHaskeller has not been maintained for the last couple of years¹, and we could not get it to compile and build properly.
- [TyGAR](#)(TYpe Guided Abstract Refinement) is a tool for synthesising Haskell programs [43]. It is used in the backend of Hoogle+ described in Section 6.4. TyGAR takes as input a type signature and generates an expression that

¹The last release that reported successfully building was: <https://hackage.haskell.org/package/MagicHaskeller-0.9.6.7> in 2017

matches the type. This tool also showed to be quite interesting, and could in theory be used to generate expressions to insert into holes. It can use polymorphic types, which according to us is its greatest advantage over our implementation. However, the results from TyGAR could as far as we found not be tailored to use only a set of predefined expressions, but only to use functions from libraries.

We found neither of these libraries to be suitable for our use-case of synthesising code from partial solutions (sketching).

7

Conclusion

The main research questions that we address in this thesis is if program synthesis can be used to complement Ask-Elle, examine how well it works, and if it can be done in a reasonable time. We have built a proof of concept, which demonstrates that this is indeed possible. It was however not integrated into Ask-Elle.

The rudimentary program synthesis we developed provide feedback for several partial programs (21% of entries) where Ask-Elle failed. This result shows that program synthesis can indeed be a valid method of generating feedback to *complement* Ask-Elle. Furthermore, we found that on attempts where Ask-Elle could provide feedback, we are not as successful (THUPY only succeeded for 35% of entries). After testing THUPY against the Ask-Elle dataset we can also conclude that it generally provides a suggestion in a reasonable time (avg. $\sim 3s$).

Our implementation only works for partial solutions with holes in expressions and with monomorphic types. Furthermore, many implementation details are not finished, and we think that implementing these could improve how well THUPY performs in the evaluation.

Bibliography

- [1] HaskellWiki. *Introduction* — *HaskellWiki*, [Online; accessed 2021-05-27]. 2020. URL: <https://wiki.haskell.org/index.php?title=Introduction&oldid=63206>.
- [2] HaskellWiki. *Pure* — *HaskellWiki*, [Online; accessed 2021-05-17]. 2013. URL: <https://wiki.haskell.org/index.php?title=Pure&oldid=56834>.
- [3] Paul Hudak, John Peterson, and Joseph Fasel. *A Gentle Introduction to Haskell, Version 98: Values, Types, and Other Goodies*. [Online; accessed 2021-05-17]. 2000. URL: <https://www.haskell.org/tutorial/goodies.html>.
- [4] HaskellWiki. *Polymorphism* — *HaskellWiki*, [Online; accessed 2021-05-31]. 2015. URL: <https://wiki.haskell.org/index.php?title=Polymorphism&oldid=59216>.
- [5] HaskellWiki. *Monomorphism* — *HaskellWiki*, [Online; accessed 2021-05-31]. 2017. URL: <https://wiki.haskell.org/index.php?title=Monomorphism&oldid=61815>.
- [6] HaskellWiki. *Lambda calculus* — *HaskellWiki*, [Online; accessed 2021-04-20]. 2021. URL: https://wiki.haskell.org/index.php?title=Lambda_calculus&oldid=63887.
- [7] Simon Peyton Jones. “Compiling Haskell by program transformation: A report from the trenches”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 1058. Springer Verlag, 1996, pp. 18–44. ISBN: 3540610553. DOI: [10.1007/3-540-61055-3_27](https://doi.org/10.1007/3-540-61055-3_27).
- [8] GP Loczewski. *A++ The Smallest Programming Language in the World: An Educational Programming Language*. 2018. URL: <https://books.google.com/books?hl=en&lr=&id=AvhcDwAAQBAJ&oi=fnd&pg=PT19&dq=A%2B%2B+The+smallest&ots=eZSqRRs1Vm&sig=sODy2kP90zjXs0aHIELKR21e23w>.
- [9] Greg Michaelson. *An Introduction to Functional Programming Through Lambda Calculus*. Tech. rep. 2011. URL: <https://books.google.com/books?hl=en&lr=&id=gKvwPtvSjsC&oi=fnd&pg=PR3&dq=the+lambda+calculus+a+brief+introduction&ots=YAfYibQu9Y&sig=b4jySHXXPYXKYTTlsXIz49racaQ>.
- [10] HaskellWiki. *Eta conversion* — *HaskellWiki*, [Online; accessed 2021-04-20]. 2021. URL: https://wiki.haskell.org/index.php?title=Eta_conversion&oldid=63978.

- [11] Taylor Fausak. *2017 state of Haskell survey results*. [Online; accessed 2021-06-14]. 2017. URL: <https://taylor.fausak.me/2017/11/15/2017-state-of-haskell-survey-results/#question-14>.
- [12] HaskellWiki. *GHC/As a library — HaskellWiki*, [Online; accessed 2021-03-26]. 2017. URL: https://wiki.haskell.org/index.php?title=GHC/As_a_library&oldid=62257.
- [13] Simon Marlow and Simon Peyton Jones. “The Glasgow Haskell Compiler”. 2012.
- [14] Stephen Diehl. *Dive into GHC: Intermediate Forms*. [Online; accessed 2021-02-19]. 2016. URL: https://www.stephendiehl.com/posts/ghc_02.html.
- [15] Haskell Wiki Contributors. *Typed Holes in GHC*. [Online; accessed 2020-12-03]. 2014. URL: https://wiki.haskell.org/index.php?title=GHC/Typed_holes&oldid=58717.
- [16] Koen Claessen and John Hughes. “QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs”. In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*. ICFP ’00. New York, NY, USA: Association for Computing Machinery, 2000, 268–279. ISBN: 1581132026. DOI: [10.1145/351240.351266](https://doi.org/10.1145/351240.351266). URL: <https://doi.org/10.1145/351240.351266>.
- [17] Rastislav Bodik and Barbara Jobstmann. “Algorithmic program synthesis: Introduction”. In: *International Journal on Software Tools for Technology Transfer* 15.5-6 (Oct. 2013), pp. 397–411. ISSN: 14332779. DOI: [10.1007/s10009-013-0287-9](https://doi.org/10.1007/s10009-013-0287-9). URL: <https://link.springer.com/article/10.1007/s10009-013-0287-9>.
- [18] Pierre Flener. “Achievements and prospects of program synthesis”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 2407 (2002), pp. 310–346. ISSN: 16113349. DOI: [10.1007/3-540-45628-7_13](https://doi.org/10.1007/3-540-45628-7_13). URL: <https://link.springer.com/chapter/10.1007>.
- [19] Peter Michael Osera and Steve Zdancewic. “Type-and-example-directed program synthesis”. In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2015. ISBN: 9781450334686. DOI: [10.1145/2737924.2738007](https://doi.org/10.1145/2737924.2738007).
- [20] Sumit Gulwani. “Dimensions in Program Synthesis”. In: *PPDP’10 - Proceedings of the 2010 Symposium on Principles and Practice of Declarative Programming*. New York, New York, USA: ACM Press, 2010, pp. 13–24. ISBN: 9781450301329. DOI: [10.1145/1836089.1836091](https://doi.org/10.1145/1836089.1836091). URL: <http://portal.acm.org/citation.cfm?doid=1836089.1836091>.
- [21] James Bornholt. *Program Synthesis Explained*. [Online; accessed 2021-04-16]. 2015. URL: <https://www.cs.utexas.edu/~bornholt/post/synthesis-explained.html>.
- [22] Susmit Jha et al. “Oracle-guided component-based program synthesis”. In: *Proceedings - International Conference on Software Engineering*. 2010. ISBN: 9781605587196. DOI: [10.1145/1806799.1806833](https://doi.org/10.1145/1806799.1806833).
- [23] Armando Solar-Lezama et al. “Combinatorial sketching for finite programs”. In: *ACM SIGARCH Computer Architecture News* 34.5 (Oct. 2006), pp. 404–

415. ISSN: 0163-5964. DOI: [10.1145/1168919.1168907](https://doi.org/10.1145/1168919.1168907). URL: <https://dl.acm.org/doi/10.1145/1168919.1168907>.
- [24] Armando Solar-Lezama. “Program Synthesis by Sketching”. PhD thesis. 2009. ISBN: 3642106714.
 - [25] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. “Program synthesis”. In: *Foundations and Trends in Programming Languages* 4.1-2 (2017), pp. 1–119. ISSN: 23251131. DOI: [10.1561/25000000010](https://doi.org/10.1561/25000000010). URL: <http://dx.doi.org/10.1561/25000000010>.
 - [26] Jonathan Aldrich. *Lecture Notes: Program Synthesis*. [Online; accessed 2021-04-16]. 2019. URL: <http://www.cs.cmu.edu/~aldrich/courses/17-355/notes/notes13-synthesis.pdf>.
 - [27] Tyne Crow, Andrew Luxton-Reilly, and Burkhard Wuensche. “Intelligent Tutoring Systems for Programming Education: A Systematic Review”. In: *ACM International Conference Proceeding Series*. Association for Computing Machinery, Jan. 2018, pp. 53–62. ISBN: 9781450363402. DOI: [10.1145/3160489.3160492](https://doi.org/10.1145/3160489.3160492).
 - [28] Albert T Corbett, Kenneth R Koedinger, and WH Hadley. “Cognitive Tutors: From the research classroom to all classrooms”. In: *Technology enhanced learning: Opportunities for change* (2001), pp. 235–263.
 - [29] Elizabeth Odekirk-Hash and Joseph L. Zachary. “Automated feedback on programs means students need less help from teachers”. In: *Proceedings of the thirty-second SIGCSE technical symposium on Computer Science Education - SIGCSE '01*. New York, New York, USA: Association for Computing Machinery (ACM), 2001, pp. 55–59. DOI: [10.1145/364447.364537](https://doi.org/10.1145/364447.364537). URL: <http://portal.acm.org/citation.cfm?doid=364447.364537>.
 - [30] Brian J Reiser, John R Anderson, and Robert G Farrell. “Dynamic Student Modelling in an Intelligent Tutor for LISP Programming.” In: *IJCAI*. Vol. 85. 1985, pp. 8–14.
 - [31] J. R. Anderson and E. Skwarecki. “The automated tutoring of introductory computer programming”. In: *Communications of the ACM* 29.9 (Sept. 1986), pp. 842–849. ISSN: 15577317. DOI: [10.1145/6592.6593](https://doi.org/10.1145/6592.6593). URL: <https://dl.acm.org/doi/10.1145/6592.6593>.
 - [32] Alex Gerdes et al. “Ask-Elle: an Adaptable Programming Tutor for Haskell Giving Automated Feedback”. In: *International Journal of Artificial Intelligence in Education* (2017), pp. 65–100. ISSN: 15604306. DOI: [10.1007/s40593-015-0080-x](https://doi.org/10.1007/s40593-015-0080-x).
 - [33] Neil T. Heffernan, Kenneth R. Koedinger, and Leena Razzaq. “Expanding the Model-Tracing Architecture: A 3rd Generation Intelligent Tutor for Algebra Symbolization”. In: *International Journal of Artificial Intelligence in Education* 18.2 (Jan. 2008), pp. 153–178. ISSN: 1560-4292.
 - [34] Alex Gerdes, Johan Jeuring, and Bastiaan Heeren. “An interactive functional programming tutor”. In: *Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE*. 2012, pp. 250–255. ISBN: 9781450312462. DOI: [10.1145/2325296.2325356](https://doi.org/10.1145/2325296.2325356).
 - [35] John Hattie and Helen Timperley. “The Power of Feedback”. In: *Review of Educational Research* 77.1 (Mar. 2007), pp. 81–112. ISSN: 0034-6543. DOI: [10.1145/1168919.1168907](https://doi.org/10.1145/1168919.1168907).

- 3102/003465430298487. URL: <http://journals.sagepub.com/doi/10.3102/003465430298487>.
- [36] Timotej Lazar. *Hint generation in programming tutors*. Tech. rep. 2018. URL: <https://core.ac.uk/download/pdf/154919696.pdf>.
- [37] Timotej Lazar, Martin Možina, and Ivan Bratko. “Automatic extraction of AST patterns for debugging student programs”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 2017, pp. 163–174. ISBN: 9783319614243. DOI: [10.1007/978-3-319-61425-0_14](https://doi.org/10.1007/978-3-319-61425-0_14).
- [38] Cyrus Omar et al. “Live functional programming with typed holes”. In: *Proceedings of the ACM on Programming Languages* (2019). ISSN: 2475-1421. DOI: [10.1145/3290327](https://doi.org/10.1145/3290327).
- [39] Cyrus Omar et al. “Hazelnut: A Bidirectionally Typed Structure Editor Calculus”. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL 2017. Paris, France: Association for Computing Machinery, 2017, 86–99. ISBN: 9781450346603. DOI: [10.1145/3009837.3009900](https://doi.org/10.1145/3009837.3009900). URL: <https://doi.org/10.1145/3009837.3009900>.
- [40] Susumu Katayama. “Recent improvements of MagicHaskeller”. In: *Lecture Notes in Computer Science*. Vol. 5812 LNCS. Springer Verlag, 2010, pp. 174–193. ISBN: 3642119301. DOI: [10.1007/978-3-642-11931-6_9](https://doi.org/10.1007/978-3-642-11931-6_9). URL: https://link.springer.com/chapter/10.1007/978-3-642-11931-6_9.
- [41] Susumu Katayama. *MagicHaskeller: An Inductive Functional Programming System for Casual/Beginner Haskell Programmers*. [Online; accessed 2021-05-27]. 2021. URL: <http://nautilus.cs.miyazaki-u.ac.jp/~skata/MagicHaskeller.html>.
- [42] Michael B James. “Digging for Fold: Synthesis-Aided API Discovery for Haskell; Digging for Fold: Synthesis-Aided API Discovery for Haskell”. In: (2020). DOI: [10.1145/3428273](https://doi.org/10.1145/3428273). URL: <https://doi.org/10.1145/3428273>.
- [43] Zheng Guo et al. “Program synthesis by type-guided abstraction refinement”. In: *Proceedings of the ACM on Programming Languages*. Vol. 4. POPL. Association for Computing Machinery, Jan. 2020. DOI: [10.1145/3371080](https://doi.org/10.1145/3371080).

A

Listing of Full AST Code

The code for the full AST, since this AST was derived from Ask-Elle's, some unused datatypes are left in the code, for instance `RefactorChoice` and `Alt`

```
1  type HoleID = String
2
3  -- Alt -----
4  data Alt
5    = AHole HoleID
6    | Alt (Maybe String) Pat Rhs
7    | AltEmpty
8    deriving (Data, Eq, Ord, Show, Typeable)
9
10 -- Alts -----
11 type Alts = [Alt]
12
13 -- Body -----
14 data Body
15    = BHole
16    | Body Decls
17    deriving (Data, Eq, Show, Typeable)
18
19 -- Decl -----
20 data Decl
21    = DHole HoleID
22    | DEmpty
23    | DFunBinds FunBinds
24    | DPatBind Pat Rhs
25    deriving (Data, Eq, Ord, Show, Typeable)
26
27 -- Decls -----
28 type Decls = [Decl]
29
30 -- TExpr -----
31 ↪ -----
```

```
31 data Expr
32   = Hole HoleID -- HsUNboundVar?
33   | Feedback String TExpr
34   | MustUse TExpr
35   | Eta Int TExpr
36   | Refactor TExpr RefactorChoices
37   | Case TExpr Alts
38   | Con Name
39   | If TExpr TExpr TExpr
40   | InfixApp MaybeExpr TExpr MaybeExpr
41   | Lambda Pats TExpr
42   | Let Decls TExpr
43   | Lit Literal
44   | Paren TExpr
45   | Tuple TExprs
46   | App TExpr TExprs
47   | Var Name
48   | Enum TExpr MaybeExpr MaybeExpr
49   | List TExprs
50   | Neg TExpr
51   deriving (Data, Eq, Ord, Show, Typeable)
52
53 -- TExpr -----
54 data TExpr = TExpr GHC.Type Expr
55   deriving (Data, Show, Typeable)
56
57 -- TExprs -----
58 type TExprs = [TExpr]
59
60 -- TExprs -----
61 type Exprs = [Expr]
62
63 -- FunBind -----
64 data FunBind
65   = FBHole HoleID
66   | FunBind (Maybe String) Name Pats Rhs
67   deriving (Data, Eq, Ord, Show, Typeable)
68
69 -- FunBinds -----
70 type FunBinds = [FunBind]
71
72 -- GuardedExpr -----
73 data GuardedExpr = GExpr TExpr TExpr
74   deriving (Data, Eq, Ord, Show, Typeable)
75
```

```

76  -- GuardedExprs -----
77  type GuardedExprs = [GuardedExpr]
78
79  -- Literal -----
80  data Literal
81    = LChar Char
82    | LFloat Float
83    | LInt Int
84    | LString String
85    deriving (Data, Eq, Ord, Show, Typeable)
86
87  -- MaybeExpr -----
88  data MaybeExpr
89    = NoExpr
90    | JustExpr TExpr
91    deriving (Data, Eq, Ord, Show, Typeable)
92
93  -- MaybeName -----
94  data MaybeName
95    = NoName
96    | JustName Name
97    deriving (Data, Eq, Show, Typeable)
98
99  -- Module -----
100 data Module = Module MaybeName Body
101    deriving (Data, Eq, Show, Typeable)
102
103  -- Name -----
104 data Name
105    = Ident String
106    | Operator String
107    | Special String
108    deriving (Data, Eq, Ord, Read, Show, Typeable)
109
110  -- Names -----
111 type Names = [Name]
112
113  -- Pat -----
114 data Pat
115    = PHole HoleID
116    | PMultipleHole HoleID
117    | PCon Name Pats
118    | PInfixCon Pat Name Pat
119    | PList Pats
120    | PLit Literal

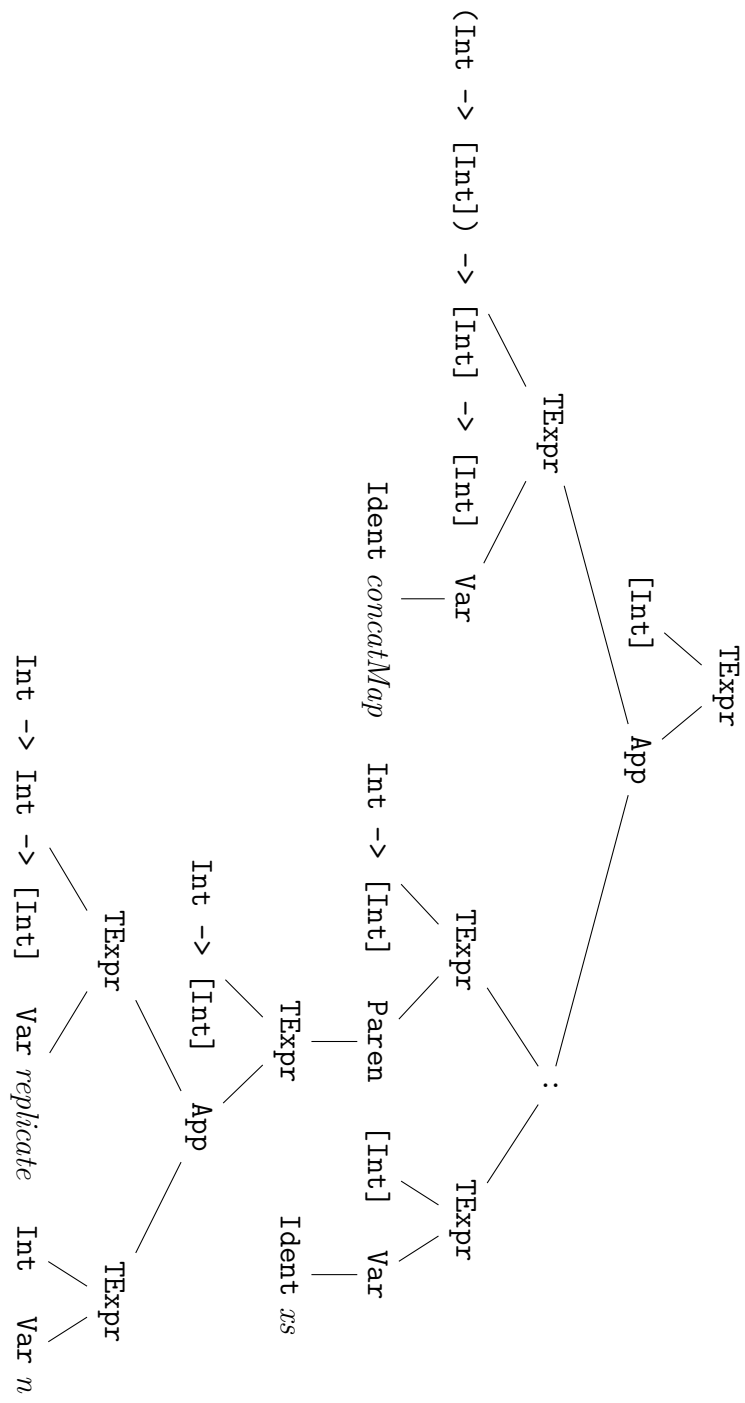
```

```
121 | PParen Pat
122 | PTuple Pats
123 | PVar (GHC.Type, Name)
124 | PAs Name Pat
125 | PWildcard
126 deriving (Data, Eq, Ord, Show, Typeable)
127
128 -- Pats -----
129 type Pats = [Pat]
130
131 -- RefactorChoice -----
132 data RefactorChoice = RefactorChoice String TExpr
133   deriving (Data, Eq, Ord, Show, Typeable)
134
135 -- RefactorChoices -----
136 type RefactorChoices = [RefactorChoice]
137
138 -- Rhs -----
139 data Rhs
140   = Rhs TExpr Decls
141   | GRhs GuardedExprs Decls
142   deriving (Data, Eq, Ord, Show, Typeable)
```

B

Full AST of Figure 3.1

The AST from Figure 3.1 in full, with **TExprs** and types appended to the original.
The expression represented is `concatMap (replicate 2) xs`



C

Evaluation Script

```
1  import sqlite3 as sql
2  import subprocess
3  import json
4  import os
5  import signal
6  import sys
7  import math
8  import time
9  import re
10
11 def signal_handler(sig,frame):
12     sys.exit(1)
13
14 signal.signal(signal.SIGINT, signal_handler)
15
16 def run_test(logs):
17     ps_bin = compile_program()
18     success = 0
19     fail = 0
20     timed_out = 0
21     subprocess_error = 0
22     done = 0
23     success_times = []
24     fail_times = []
25     total = len(logs)
26
27     for log in logs:
28         params = json.loads(log[0])["params"]
29         ex_id = params[0][0].replace(".",
30             ↪ "/").replace("haskell/", "")
31         ex_fun_name = ex_id.split("/")[1]
32
33         inp = params[1]
34         inp = str.strip(inp)
```

```
34     inp = re.sub(ex_fun_name, ex_fun_name, inp,
35     ↪ flags=re.IGNORECASE)
36     inp = getFuncString(ex_id) + "\n" + inp
37     print(f"--- Trying {ex_id} ---")
38     try:
39         s = time.time()
40         result = subprocess.run(["stack", "run", "--", ex_id,
41         ↪ inp], capture_output=True, text=True, timeout=30,
42         ↪ check=False)
43         e = time.time()
44
45         if os.path.exists("/tmp/askelle-test.hs"):
46             os.remove("/tmp/askelle-test.hs")
47         if os.path.exists("/tmp/askelle.hs"):
48             os.remove("/tmp/askelle.hs")
49
50         stderr = result.stderr
51         stdout = result.stdout
52         print(stdout)
53
54         if "Suggestion" in stderr:
55             print("successful: ", inp)
56             print(stderr)
57             success += 1
58             success_times.append(e-s)
59         elif "No holes, nothing to do" in stderr:
60             print("nothing to do ", inp)
61             success += 1
62             success_times.append(e-s)
63         elif "scavenge_one" in stdout:
64             print(stderr)
65             subprocess_error += 1
66         else:
67             print("# FAILED #")
68             # print(stderr)
69             print(inp)
70             fail += 1
71             fail_times.append(e-s)
72     except subprocess.TimeoutExpired:
73         print("timed out")
74         timed_out += 1
75     except subprocess.SubprocessError:
76         print("subprocessError")
77         subprocess_error += 1
```

```

76     done += 1
77     print("#####")
78     print(f"## Finished {done:4} out of: {total:4} ##")
79     print(f"## Timed out: {timed_out:3} ##")
80     print(f"## Subprocess Error: {subprocess_error:3}
      ↪ ##")
81     print(f"## Successful: {success:3} ({round(success/done *
      ↪ 100):3}%) ##")
82     print(f"## Avg Success time:
      ↪ {avg_time(success_times,2):.2f}s ##")
83     print(f"## Max Success time:
      ↪ {max_time(success_times,2):.2f}s ##")
84     print(f"## Avg Fail time:
      ↪ {avg_time(fail_times,2):.2f}s ##")
85     print(f"## Max Fail time:
      ↪ {max_time(fail_times,2):.2f}s ##")
86     print(f"## Avg Run time: {avg_time(success_times +
      ↪ fail_times,2):.2f}s ##")
87     print("#####")
88
89 def avg_time(times,rnd):
90     avg = 0
91     if len(times) > 0:
92         avg = round(sum(times) / len(times),rnd)
93     return avg
94
95 def max_time(times,rnd):
96
97     if len(times) > 0:
98         return round(max(times), rnd)
99     return 0
100
101 def compile_program():
102     bin_name = "./askelle-ps-pythontest"
103     try:
104         subprocess.run(["stack", "--local-bin-path", ".",
      ↪ "--copy-bins", "build"])
105         subprocess.run(["mv", "askelle-ps-exe", bin_name])
106         return bin_name
107     except:
108         print ("couldn't build askelle-ps binary, aborting...")
109         sys.exit(1)
110
111 def getFuncString(exName):
112     fName = exName.split("/")[1]

```

```
113     f = f"exercises/haskell/{exName}/Config.hs"
114     with open(f) as file:
115         for line in file:
116             if "properties" in line:
117                 return fName + " :: " + line[16:-15]
118
119 print(getFuncString("list/repli"))
120
121 request_file = "requests.db"
122 con = sql.connect(request_file)
123 cursor = con.cursor()
124 drifted_logs = cursor.execute("SELECT input FROM requests WHERE
    ↳ service LIKE 'feedbacktextdeep' AND output LIKE
    ↳ '%%drifted%%').fetchall()
125 cursor2 = con.cursor()
126 askelle_success_logs = cursor2.execute("SELECT input FROM
    ↳ requests WHERE service LIKE 'feedbacktextdeep' AND output
    ↳ LIKE '%%Correct%%').fetchall()
127
128
129 print("where ask-elle drifted, we did:")
130 run_test(drifted_logs)
131 print("where ask-elle succeeded, we did:")
132 run_test(askelle_success_logs)
```

D

Some generated solutions from the evaluation

```
-- Student Input
dupli list = concat (map ? list)
-- Generated Solution
dupli list = concat (map (replicate 2) list)
-- Suggestion
dupli list = concat (map (replicate ?) list)
=====
-- Student Input
pack (x : xs) = ?
pack [] = ?
-- Generated Solution
pack (x : xs) =
  (x : takeWhile (== x) xs) : pack (dropWhile (== x) xs)
pack [] = []
-- Suggestion
pack (x : xs) = ? : ?
pack [] = []
=====
-- Student Input
dupli :: [Int] -> [Int]
dupli = \l -> ?
-- Generated Solution
dupli = \l -> (concatMap (replicate 2) l)
-- Suggestion
dupli = \l -> (concatMap ? ?)
=====
-- Student Input
compress (x : (y : ys)) = ?
compress x = x
-- Generated Solution
compress (x : (y : ys)) =
  (if x == y then [] else [x]) ++ compress (y : ys)
```

```

compress x = x
-- Suggestion
compress (x : (y : ys)) = ? ++ ?
compress x = x
=====

-- Input
myreverse :: [Int] -> [Int]
myreverse [] = []
myreverse (x:xs) = ?
-- Generated Solution
myreverse [] = []
myreverse (x : xs) = myreverse xs ++ [x]
-- Suggestion
myreverse [] = []
myreverse (x : xs) = ? ++ ?
=====

-- Input
dupli :: [Int] -> [Int]
dupli [] = []
dupli (x:xs) = ? : ? : ?
-- Generated solution
dupli [] = []
dupli (x : xs) = x : x : dupli xs
-- Suggestion
dupli [] = []
dupli (x : xs) = x : x : dupli ?

```
