CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Improving Recognition of Student Programs in Ask-Elle

Master's thesis in Computer Science and Engineering

Matilda Blomqvist

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2023

Master's thesis 2023

Improving Recognition of Student Programs in Ask-Elle

Matilda Blomqvist

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2023

Improving Recognition of Student Programs in Ask-Elle

MATILDA BLOMQVIST

Supervisor: Alex Gerdes, Department of Computer Science and Engineering
Examiner: Mary Sheeran, Department of Computer Science and Engineering

Master's Thesis 2023
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 LaTeX
Gothenburg, Sweden 2023

Improving Recognition of Student Programs in Ask-Elle

MATILDA BLOMQVIST
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Ask-Elle is a programming tutor offering a simple interface for solving small programming exercises in Haskell. Feedback is given incrementally by allowing the students to have *holes* in their program, where the system can offer suggestions on how to fill the hole. To give constructive feedback and not only point out errors, Ask-Elle first *matches* student programs against a set of model solutions defined for the exercise. The feedback is limited for programs that cannot be recognised as matching a model solution.

Haskell offers a rich syntax that allows writing semantically but not syntactically equivalent programs, making matching programs difficult. A way of removing such syntactic differences is to normalise programs by applying normalising program transformations. Although this is already done in Ask-Elle, not all correct programs can currently be recognised. We have implemented a new approach to normalisation and feedback generation using the GHC API to utilise GHC's internal transformations and warning messages to improve the recognition rate and the generated feedback. The new approach shows a slight improvement in the number of recognised programs. It also enables an alternative way of generating hints for filling a hole in an incomplete program which is made possible by using GHC's intermediate Core language.

Keywords: Program transformations, Normalistation, GHC, GHC Core, Haskell, Programming tutor, Ask-Elle.

Acknowledgements

First and foremost, I would like to thank my supervisor Alex Gerdes for all the time, effort and support he has devoted to helping me with this thesis project. I would also like to thank Beata Burreau for many helpful discussions, late nights at the library and careful proofreading, and Maciej Goszczycki for generously allowing me to use valuable data he collected for his own master's thesis. Thanks to my examiner Mary Sheeran for pointing out important aspects of the evaluation of my results and keeping me on my toes with my grammar. Finally, to Ludvig and all my family and friends: thanks for your support and for always believing in me.

Matilda Blomqvist, Gothenburg, 2023-06-21

Contents

1

Introduction

Learning to program is challenging, even with access to suitable course materials and good teachers [1]. It requires practice, which does not always take place in a classroom where tutoring from a teacher is available. Students often write programs and test if the program produced the expected result by compiling and running it. Modern compilers point out errors at different levels in the source code, but beginners often find it difficult to decipher error messages from the compiler [2]. Hence, other tools are needed to help facilitate learning for novices. Often, some kind of specialised integrated development environments (IDEs) is proposed, like intelligent tutor systems, customised compilers with more direct error messages [3] or a combination of those [4]–[6].

In a review study of several intelligent tutor systems, VanLehn concluded that intelligent tutor systems are almost as efficient as human tutors [7], although human tutoring has long been believed to be the most efficient way of learning. Including intelligent tutors in education could have many benefits besides facilitating learning. For example, it could save course instructors time, allow instant feedback and reduce idle time for students waiting for help. It could also lead to more equal learning environments where each student has the same access to tutoring. Moreover, one review study by Kumar [8] showed that using intelligent programming tutors can help improve female computer science (CS) students' confidence. Several studies have shown that the confidence of female CS students is lower than their male counterparts [9], [10] and if digital tutors can enhance the confidence of female students, we have a strong motivation to further develop them.

1.1 Ask-Elle

Ask-Elle is an online Haskell programming tutor aiming to improve students' programming skills in Haskell by providing comprehensible feedback [11], [12]. The tutor focuses on small programming tasks, like implementing a single function, suitable for introductory Haskell courses. On a high level, Ask-Elle works in the following way: a teacher sets up programming exercises by providing *model solutions* and properties that a solution to an exercise should have. Ask-Elle tries to match the student-provided program against the model solutions, and if it succeeds, it generates feedback from *strategies*, which in turn are derived from the model solutions [13]. If a program cannot be matched with a model solution, feedback is generated from property-based testing with QuickCheck [14] or compiler messages.

We show a fictional student interaction with Ask-Elle to exemplify how this works. The editor initially contains a single question mark `?`, which denotes a *hole* in Ask-Elle and corresponds to the `undefined` keyword or typed hole (`_`) construct in Haskell. A hole is a term of any type and can thus be used as a placeholder in incomplete programs. Now, consider the Ask-Elle exercise displayed in Figure 1.1. The student is asked to define a function `dupli` that duplicates all elements in a list. The student starts by pattern-matching on the argument to the function but does not know how to proceed and simply puts `?` on the right-hand side of the function definition. When the student actively requests help, Ask-Elle displays feedback in the right panel. Here Ask-Elle hints that the student should use recursion and introduce the `(:)` constructor for lists. The student can then proceed by trying to define the function, partially or completely, before checking progress again.

Figure 1.1: Screenshot of the Ask-Elle web interface[1]. The student inputs code in the editor to the left, and feedback generated for the submitted attempt is displayed in the panel to the right.

Suppose that the student follows the guidance from Ask-Elle and proceeds by introducing the `(:)` constructor, obtaining the following program:

```
dupli (x:xs) = ? : ?
```

The next hint is similar to the first one, stating that explicit recursion can be used and that the `(:)` constructor can be introduced. If asking for more help (by clicking the "More Help" button in the interface), we get the instruction to refine the current term to `dupli (x:xs) = ? : ? : ?`. If then replacing the first two holes with x, obtaining `dupli (x:xs) = x : x : ?`, asking for help again, Ask-Elle suggests that a recursive call to `dupli` could be introduced. Refining the last hole to `dupli ?` and asking for feedback one more time gives that the variable `xs` should be introduced. If the student follows this guidance, they should have obtained the following solution:

```
dupli (x:xs) = x : x : dupli xs
```

The final hint is to introduce a new function binding. The completed solution can be obtained by introducing a function binding for the empty case:

```
dupli []     = []
dupli (x:xs) = x : x : dupli xs
```

1.1.1 Recognising solutions

To provide constructive feedback that helps a student continue developing a program, Ask-Elle tries first to recognise if the student's attempt corresponds to a model solution. If Ask-Elle cannot recognise that a student is following the strategy of any model solution, the given feedback is less helpful. Recognising that a student program matches a model solution is done syntactically, term by term, checking if the student's solution is syntactically equivalent to a model solution. Determining if two programs are equal is undecidable in general, which constitutes a challenge in deciding whether a student program is correct. To overlook syntactical differences in otherwise equivalent programs, Ask-Elle applies a set of normalising *program transformations* on both the submitted program and the model solutions. However, no normal form exists for Haskell programs, and normalisation cannot remove all syntactical differences. Nevertheless, Ask-Elle's current normalisation procedure can be improved to match a larger set of programs. A previous manual evaluation of a set of attempted student solutions concluded that refining or adding new program transformations and model solutions could increase recognised solutions from 56% to 81% for that particular set of programs [12].

1.2 Aim

The overarching goal of this thesis is to improve the usefulness of Ask-Elle in real-world learning scenarios by giving helpful feedback to students in more cases. By implementing new program transformations in Ask-Elle, we aim to recognise more

[1] https://ideas.science.uu.nl/AskElle

student programs and hence be able to give feedback to students where Ask-Elle currently fails to do so. We aim to reach the aforementioned goal by using GHC's internals to utilise the typed holes mechanism for representing holes, reuse existing program transformations, ensure type correct transformations, and use GHC's internal abstract syntax (AS) to be able to support all available Haskell constructs in future exercises. The latter is not possible in the current version, since Ask-Elle uses a customised AS that only covers function definitions.

In short, we seek to answer the following research questions:

1. Can GHC's intermediate language and internal program transformations be used for normalisation to improve recognition of student programs in Ask-Elle?
2. Can additional information be retrieved from GHC to give more detailed feedback?

1.3 Contributions

We have developed a new procedure for normalisation by using GHC's internal transformations and adding additional transformations on GHC's intermediate core language. The new approach shows slight improvements in the recognition of student programs while being simpler and faster than the current way of diagnosing programs in Ask-Elle. It also enables an alternative way of retrieving hole-fits by directly matching normalised student programs with the model solutions – without the overhead of creating intermediate model solutions as in the current version of Ask-Elle. Suggestions from HLint and other GHC warnings are given as additional feedback.

This work is meant to be integrated into Ask-Elle, offering an alternative way of generating feedback in the existing system. Using GHC's core language allows using the full syntax of Haskell 2010, including type signatures, which previously were disregarded. It also enables the possibility to incorporate other kinds of exercises in Ask-Elle, including more Haskell constructs than just functions. However, this thesis project is delimited to consider the existing exercises in Ask-Elle. Matching other types of Haskell constructs, like data type definitions, has not yet been thoroughly evaluated and is subject to future work. Haskell modules containing data type and typeclass definitions have been checked to be type correct after normalisation, but additional normalising transformations might still be needed to match such programs syntactically.

2

Background

Haskell offers a rich syntax that provides many ways to write semantically, but not syntactically, equivalent programs. Conditioning on a boolean value is a simple example highlighting how different syntactical constructs in Haskell can be used to write (semantically) equivalent code. For example, a boolean comparison can be made with a case expression:

```
case x of
    True -> something
    False -> something_else
```

Or simply with an if-then-else expression:

```
if x then something else something_else
```

It would be unfortunate if a Haskell programming tutor fails the student due to such differences. To overlook such differences when recognising student programs, normalising program transformations can be applied to the programs before attempting to match them.

2.1 Program Transformations

Most compilers apply different program transformations to remove redundant syntactic constructs used in a program. Program transformations are also a common technique for code optimisation, and refactoring [15]. A transformation is naturally defined by a rule expressing how a fragment or a particular construct in a program can be transformed in one step. Program transformations are applied to an abstract syntax tree (AST), a compiler's internal representation of a program obtained after feeding source code into a parser.

When applying different program transformations to a program, the order of application of different transformations must be considered with some care. Particular transformations might enable other transformations, so the order of application is important. It is not always possible to find an order to sequence the transformations

to normalise or optimise a program as far as possible. They might need to be applied several times, which is often the case in compilers.

In the context of Ask-Elle, the goal is not to optimise the code or to translate a program to another target language in contrast to compilers. The focus lies on program *normalisation*, using program transformations to reduce a program to a syntactically different but semantically equivalent version of the same program by some definition of semantics [15]. Ask-Elle performs a few transformations like α-conversion, partial β- and η-reduction, inlining and desugaring. All of α-conversion, β-, and η-reduction originate from the lambda calculus [16]. To begin with, β-reduction is formally defined as:

$$(\lambda x.M)E \rightarrow_\beta M[x := E]$$

which can be seen as the application of the body of the abstraction on the bound variable x substituted with the argument E. The converse of this transformation, usually referred to as β-abstraction, performs the transformation in the opposite direction. Similarly, η-reduction drops an abstraction, and is defined as

$$(\lambda x.fx) \rightarrow_\eta f.$$

The converse, η-expansion, introduces an abstraction. Commonly, without a specified direction, the above transformations are referred to as η- and β-conversion. Finally, α-conversion, or α-renaming, is the process of substituting, or renaming, bound variables. When performing α-renaming, it must be made sure that free variables are not captured by the substitution. For example, consider the expression

$$\lambda x.x\ y$$

where $[x := y]$ is not a valid substitution, since the free variable y would be captured by the abstraction in $\lambda y.y\ y$ and change the meaning of the expression.

Inlining is the process of replacing a variable with its definition. Inlining recursive functions is avoided since then the inlining process might not terminate. However, we can "inline" recursive functions by introducing a let binder for the recursive definition, creating a *letrec*. Generally, inlining should be performed carefully since inlining large expressions used in several locations in the code could slow down performance significantly [17].

Lastly, desugaring is the process of removing syntactic sugar by rewriting expressions. For example, a common desugaring transformation in Haskell is rewriting where-clauses as let-expressions and if-expressions as case expressions. Other examples of desugaring transformations implemented in Ask-Elle are rewriting function bindings and guarded expressions as case expressions, infix operators as prefix operators, and removing dead code and parentheses [11].

2.2 Ask-Elle

When a student tries to solve a programming exercise, Ask-Elle first compiles the submitted program to check that it is syntactically and type correct. Both the student program and the model solutions are translated into a customised AST (after the initial compilation) used for the internal representation of the student and model programs. The data types of this abstract syntax have special constructors to represent holes on different levels - for example in declarations, expressions, patterns etc. The full abstract syntax used in Ask-Elle is given in Appendix A.1.

If the compilation fails, error messages from the Helium Haskell compiler[1] are used to provide feedback. Instead, if the given student program parses and is type correct, Ask-Elle proceeds by matching it against a suite of model solutions. If Ask-Elle succeeds in matching a complete student program with a model solution, the exercise is finished. If a partial student solution is matched with a model solution, the student can ask for hints on the next step; for example, on possible functions to use or patterns to introduce. Otherwise, Ask-Elle runs QuickCheck tests on the student program. If all the tests pass, the student program is probably correct but written in a more or less different way than any of the model solutions. If QuickCheck instead finds an error, it reports the QuickCheck-generated counter-example to the student. Another possible outcome of the QuickCheck tests is inconclusive: when, for example, the student program contains holes or an infinite loop. Tests of such programs will be discarded by QuickCheck, and if too many tests are discarded or timed out, Ask-Elle cannot give any diagnosis. The student-provided programs can be partitioned into five categories:

Compiler-error: Syntax or type error detected.

Model: Matches a model solution.

Test passed: all QuickCheck tests passed.

Counter: QuickCheck-generated counter example.

Unknown: QuickCheck did not find any counter examples.

Programs in the Unknown and Test passed categories are of most interest in this thesis since such programs might still be a proper, yet unrecognised, solution to a given exercise.

2.2.1 Exercises and Model Solutions

Intuitively, an exercise in Ask-Elle is what the student tries to solve. The exercises currently available in Ask-Elle all ask the student to define a single function, possibly with helper functions. The main function the student is asked to implement has the same name as the exercise, but there are no restrictions on the number of functions the student may use to solve the exercise.

[1] https://hackage.haskell.org/package/helium

An exercise's functional, or input-output, behaviour is specified by the teacher-defined model solutions (source code implementations) and a set of properties. The model solutions also decide which functions are allowed in a correct solution. The student program cannot be recognised if a student uses a Prelude function that does not appear in any model solution. Model solutions can be specified further by annotations or *pragmas* attached directly to particular terms in the source code (see Figure 2.1). For example, the *feedback* (F) pragma provides additional feedback to a certain step of the program, while the *description* (DESC) pragma gives a description of the approach implemented by the model solution.

The *must use* pragma explicitly declares that a specific language construct must be used and disallows alternative definitions. Pragmas give some freedom to a teacher to customise exercises and enforce specific coding conventions. Furthermore, the *alternative* pragma allows a teacher to specify alternative definitions to accepted constructs without defining a completely new model solution. Yet, even with the alternative pragma, it would require quite some work to capture all possible program variations by specifying model solutions alone [11].

```
dupli :: [a] -> [a]
{-# DESC Use the @prelude function @concatMap. #-}

dupli = {-# F @concatMap first maps a function over a @list and
    then concatenates the result.
    #-} concatMap (replicate 2)
```

Figure 2.1: An example of a model solution for the dupli exercise. The description (DESC) and feedback (F) pragmas are used in the later feedback generation.

The properties used for QuickCheck testing are defined in a Haskell file named Config.hs, which is shared for all model solutions for a particular exercise. In addition to the specified properties, the configuration file for each exercise also contains the exercise description.

2.2.2 Deriving Strategies

Ask-Elle uses a domain-specific language by Heeren et al. [13] to diagnose a student-submitted program. The language is developed for defining strategies for analysing exercises in intelligent tutor systems (ITS). Strategies for a specific domain can be defined in the language, where a strategy and the used rules can be regarded as a context-free grammar. It can be used in different domains, such as for logic exercises or programming exercises like in Ask-Elle [11]. In the functional programming domain, the strategies consist of refinement rules. The strategy language consists of combinators that can be used to, for example, specify the order of application of the rules in the strategy, if some rules must be applied in sequence or if multiple rules can be applied to a specific stage in the development of a program. Each refinement rule has a corresponding feedback message displayed to the student when the system recognises that this rule could be applied next.

A strategy for an exercise is derived from the set of model solutions for the exercise in question. Hence, a derived strategy is connected with a specific exercise and is not a general strategy for refining programs. A strategy is used to generate intermediate versions of the model solutions, which is done by applying all possible rules (from the strategy) at that point in the program [11]. Student attempts are then syntactically matched against the intermediate model solutions. If any of the derived programs match the submitted program, Ask-Elle can determine which strategy the student is following and give feedback accordingly.

2.3 The Glasgow Haskell Compiler

The Glasgow Haskell Compiler (GHC) is presumably the most used compiler for Haskell programs, as indicated by a survey from 2022 [18]. GHC contains several compilation passes; the first passes of the GHC compiler pipeline[2] are displayed in Figure 2.2. GHC first parses a Haskell program to an initial AST called HsSyn. This abstract syntax type is comparatively large since it represents Haskell in its entirety. For reference, the datatype for HsSyn expressions contains over 40 constructors. GHC's renamer and typechecker operate on the same AST but replace dummy placeholder types in the parsed AST with actual inferred types. The next pass, *desugar*, is where things become interesting. The large AST in HsSyn is transformed into a tiny intermediate core language, based on System FC, with an expression data type with only ten constructors. GHC's core language is referred to as CoreSyn or simply *Core*. The next pass, called the *simplifier*, performs many simplifying and rewriting transformations on Core programs[3]. The later passes in GHC's compiler pipeline (e.g. code generation) are not interesting for this thesis and are not discussed. GHC also offers the possibility to add compiler plugins, which allows the implementation of specialised behaviour for some compilation passes.

[2]https://gitlab.haskell.org/ghc/ghc/-/wikis/commentary/compiler/hsc-main
[3]https://gitlab.haskell.org/ghc/ghc/-/wikis/commentary/compiler/core-to-core-pipeline

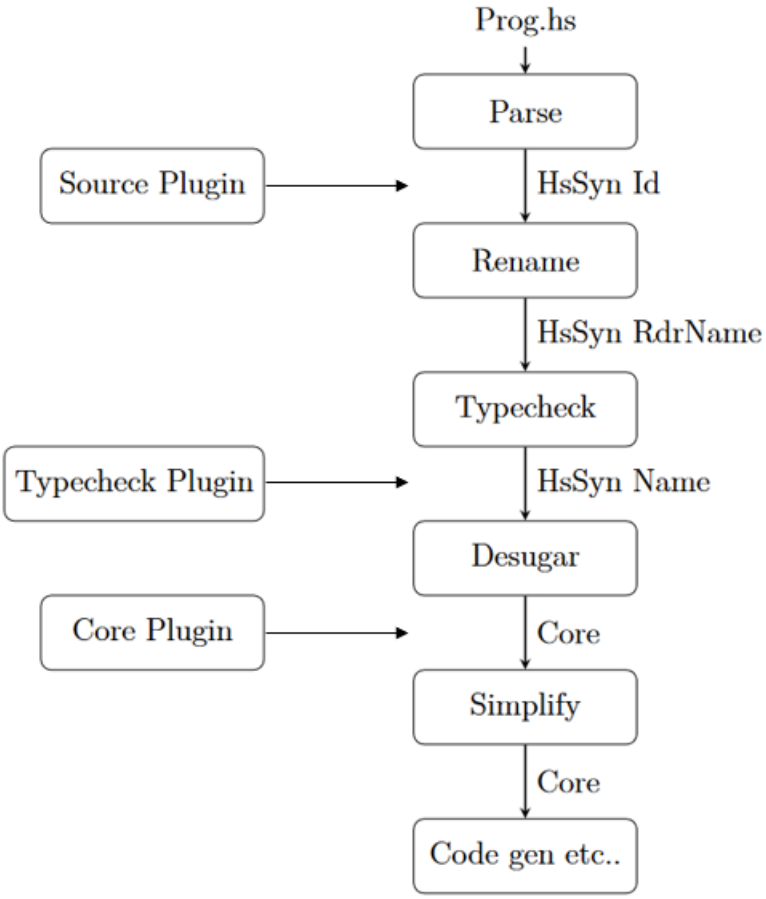

Figure 2.2: The first passes of the GHC compilation pipeline. The compiler can be extended in a modular way by allowing users to add compiler plugins.

GHC can be customised by setting different options that affect compilation in various ways. Options can specify behaviour, such as which transformation or simplifications to apply, warning messages to ignore, or errors to defer to later passes. Options can be set by passing command line flags to GHC, through the GHC API, or as option pragmas in the compiled source file.

2.3.1 GHC Core

The GHC Core language is an implementation of System FC [19], an extension to the more well-known System F. System F, in turn, is an extension of the simply-typed lambda calculus that allows existential quantification over types [19]. Existential quantification over types is a way of formalising parametric polymorphism, a common concept in many programming languages. On top of abstracting over types, the extensions made in System FC include ways of encoding constructs such as general algebraic datatypes (GADTs), functional dependencies and associated types [20]. The central idea is to explicitly pass around *evidence* of type equalities that can be abstracted over and applied to types and terms like any other term [21]. Concrete

examples of how types and type evidence are incorporated in the Core AST are given in subsection 2.3.2.

The Haskell implementation of System FC, Core, is displayed in Figure 2.3. Core shares some of the most fundamental constructs found in regular Haskell but, of course, it does not come close to the number of different available ones (then, what would be the point?). It is also relatively small compared to the current customised abstract syntax for representing student programs in Ask-Elle. A direct implication of the small number of expression constructors is the loss of specificity. Core expressions are purposely general to capture a lot of different Haskell concepts with the same constructor. The generality also entails that corresponding Haskell source programs are not immediately recognisable.

```
type CoreProgram = [CoreBind]
type CoreBind = Bind Var
type CoreExpr = Expr Var
type Var = Id

data Bind b = NonRec b (Expr b) | Rec [(b, (Expr b))]

data Expr b
   = Var   Id
   | Lit   Literal
   | App   (Expr b) (Expr b)
   | Lam   b (Expr b)
   | Let   (Bind b) (Expr b)
   | Case  (Expr b) b Type [Alt b]
   | Cast  (Expr b) Type
   | Tick  (GenTickish Id) (Expr b)
   | Type  Type
   | Coercion Type

data Alt b = Alt AltCon [b] (Expr b)
data AltCon = DataAlt DataCon | LitAlt  Literal | DEFAULT
```

Figure 2.3: The Core data types[4] and the most common type synonyms.

A Core program is a list of `CoreBind`s instantiated with the `Var` type, which is a synonym for `Id`. An `Id` is a bit simplified[5], a named thing with a `Unique` and a type. A `Unique` can be thought of as a combination of letters and digits that serves as a unique identifier of a variable, hereafter simply referred to as a unique. Both `Id` and `Var` are used extensively in GHC's codebase, but these terms are completely interchangeable and will generally be referred to as variables.

[4] https://gitlab.haskell.org/ghc/ghc/-/wikis/commentary/compiler/core-syn-type

[5] Non-type variables have a lot of additional information, like scope, multiplicity or if it is some kind of special variable (like a primitive operator, for example).

A `CoreBind` binds a variable to an expression and, for simplicity, is referred to as a *binder*. There are two types of binders: recursive and non-recursive. Yet, non-recursive binders can contain recursive structures through the `Let` expression with a recursive `Rec` binder, creating a so-called *letrec* expression. Hence, we can also talk about *top-level binders*, which refers to `CoreBind`s in the list that makes up the `CoreProgram` and *let-binders*, which refers to a `CoreBind` contained in a `Let` expression.

The `Var` constructor of the `Expr` data type is used to represent any variable, like a function name or type variable. `Lit` expressions are used for different kinds of literals, such as integer literals, deferred error messages, and names of specified data constructors in the compiled module. `App` is used for regular function applications and type applications, just like the `Lam` constructor is used for both term and type abstractions. A `Lam` expression incorporates types and type constraints in the AST or is used as a regular lambda abstraction over terms. The `Let` expression contains a `CoreBind` (the "let ..." part) and another expression (the "in ..." part).

The `Case` expression somewhat corresponds to the `case` expression in Haskell in the sense that it can be used to condition over data types or boolean values. `Case` expressions are also used to represent the different pattern-matching options in a function with multiple function bindings, guarded expressions, if-then-else expressions, etc. In other words, `Case` expressions appear frequently; when case expressions are discussed in subsequent chapters, it refers to Core `Case` expression rather than Haskell `case` expression if not stated otherwise. A `Case` expression also has a *case-binder*, a variable that binds the result of the *scrutinee*, that is, the expression we are conditioning on. Another difference from the Haskell `case` expression is that `Case` expressions are strict, meaning that the inner expression or the scrutinee of a `Case` expression always must be evaluated. Hence, any function that diverges (results in error) is represented as a `Case` expression when compiled to Core. For example, the strict Haskell function `seq`, which evaluates the first argument and returns the second one, is compiled into a `Case` expression. The `Type` is the resulting type of the whole case expression. That `Case` expression contains a type field is mostly a legacy from implementing GADTs before GHC's core language implemented System FC.

The `Case` alternatives are a list of `Alt`s. The `Alt` data type consists of an `AltCon`, a list of variables and an expression. An `AltCon` is, in turn, either a data constructor, literal, or the default case. The list of variables contains all variable patterns used in the alternative, and the expression is the resulting expression. For example, suppose that the `Alt` represents a non-empty list pattern. Then, the `AltCon` is a `(:)` constructor (`DataAlt (:)`), the variables could be the list `[x,xs]` and the expression corresponds to the expression after the arrow (`->`) in regular Haskell `case` expressions.

`Case` expressions are exhaustive in the sense that they always cover all reachable alternatives but not necessarily all available constructors. If GHC detects that a function or other structure has non-exhaustive patterns, a "pattern error" alternative is generated. In some cases where there could be multiple pattern errors, GHC

generates a new binder to a generated `fail` function (to enable sharing) that is called in every pattern error alternative.

Additional information that can be used for debugging is stored in a `GenTickish` expression, for example, breakpoints or pure annotations of the source code. A `Type` expression is simply a type, and `Coercion` is used for type coercions.

To enhance the readability of example Core programs in this report, `Tick`s, `Type`s and module-related binders are omitted when displaying Core programs if they are not relevant to the example.

2.3.2 Types in Core Expressions

Inferred and specified types from type signatures are explicitly contained in Core programs as lambda expressions. Type constraints on type variables also create evidence variables in the expression. Evidence variables are prefixed with '$d' and contain evidence that a type variable complies with its type constraint. For example, consider the very simple Haskell function below:

```
isZero :: (Eq t, Num t) => t -> Bool
isZero x = x == 0
```

It is compiled into the following Core program:

```
[NonRec isZero (Lam t (Lam $dEq (Lam $dNum
 (Lam x (App (App (App (App (Var ==) (Type (TyVarTy t)))
     (Var $dEq)) (Var x))
     (App (App (App (Var fromInteger)
         (Type (TyVarTy t))) (Var $dNum)) (Lit 0))))))))]
```

The outermost expression is a lambda abstraction over the type variable `t`, which in turn abstracts over the evidence variables imposing the constraints on `t`. The type variable `t` and the evidence variable `$dEq` is first applied to the input variable `x` in the application of the equality comparison `(==)` since the equality operator expects arguments of some type that has an `Eq` instance. The type variable `t` and the evidence variable `$dNum` is applied to the literal 0 in the application of the `fromInteger` function since `fromInteger` has a constraint on its resulting type belonging to the `Num` typeclass which also is the constraint of the top-level function `isZero`. This is a trivial example, but it highlights how types and type constraints are part of the expressions themselves in Core. Finally, GHC removes all types and type evidence before running the program, ensuring that the type passing does not incur extra run-time costs [21].

2.3.3 Transformations in GHC

GHC performs a lot of transformations; we only give a brief overview of some of the transformations made in the first passes of the GHC pipeline.

The Renamer

GHC's renaming pass performs capture-avoidant renaming by assigning uniques to every unique variable. This approach to renaming allows keeping variable names as they appeared in the source code while being able to tell them apart. Keeping the source code names provides both easier debugging and good error messages, even in later stages of the compilation process.

The Desugarer

The desugaring pass removes a lot of the syntactic sugar that Haskell includes. Function bindings, guards, if-expressions, and any function call resulting in an error are transformed into a case expression. Another example is `where` and `let` clauses that both are transformed into a `Let` expression in Core.

At the end of the desugaring pass, GHC runs a lightweight optimiser that inlines trivial functions or functions that are only used once, performs β-reduction, simplifying case expressions on known constructors and dead code elimination[6]. Whether η-reduction is performed depends on if the `Opt_DoEtaReduction` option is set. GHC's η-reduction is not deterministic and does not always reduce, even when tools like Hlint[7] manage to recognise that an expression could be reduced. GHC only applies transformations when performance would gain from it, which makes GHC's η-reduction unreliable for normalisation.

GHC's inliner is based on heuristics [17], and is hence not deterministic either. Expressions that GHC considers too large or are used in multiple places are not inlined by the simple inliner. Both top-level and local functions are defined with `where` and `let` clauses are inlined if deemed small enough. However, top-level functions are not removed from the AST when inlined since they potentially could be used somewhere else. Finally, programs only contain *reachable* cases after desugaring since all expressions with overlapping patterns will be removed in the dead code elimination.

The Simplifier

The simplifier offers transformations such as floating out lambdas or constants to top-level binders, merging nested case expressions scrutinising the same expression, η-expansion and many more. The η-expansion is based on arity information when checking whether an expression can be expanded. Holes have no arity information since they normally result in an error and are given the default arity zero. Expressions that include holes or application of holes will thus never be η-expanded. Which transformations the simplifier performs can be customised by the many options GHC offers for the simplifier. However, some transformations are *always* performed and cannot be set through some option. For example, when the simplifier encounters a program that contains a case expression that results in an error, the rest of the program might be thrown away since it will never be reached.

[6] https://gitlab.haskell.org/ghc/ghc/-/blob/master/compiler/GHC/Core/SimpleOpt.hs

[7] HLint is a tool that hints about possible rewrites, warnings, and other possible code improvements to the user

The simplifier rewrites `DEFAULT` case alternatives to the correct `DataAlt` or `LitAlt` if it can be inferred. For example, consider the partial implementation of `dupli` below:

```
dupli [] = []
dupli xs = _
```

When compiled into Core, the program will contain a case expression with one case alternative for the empty case (a `DataAlt []`) and a `DEFAULT` case for the general pattern `xs`. Since the first case alternative is a the empty list constructor, GHC can infer that the second case alternative must be a non-empty list. The simplifier will hence replace the `DEFAULT` case alternative with a `DataAlt (:)`.

2.3.4 The Core Linter

The fact that Core implements the well-defined language System FC enables a compelling concept: typechecking of Core programs. The Core typechecker implemented in GHC is called the *Core Linter*, which does some sanity checking and typechecking of Core programs [22]. The first typechecker pass has, of course, already checked that the source program was type correct, but typechecking Core programs gives a way of checking that all later transformations applied by the compiler have not resulted in an ill-typed program.

3

Problem

To understand how program transformations could *improve recognition of student solutions in Ask-Elle* more concretely, we will first look at specific scenarios considering possible student solutions to the exercise from Figure 1.1 specified below:

```
Duplicate the elements of a list:

dupli :: [a] -> [a]

For example:

> dupli [1,2,3]
[1,1,2,2,3,3]
```

First, assume that we have defined two model solutions:

```
dupli_1 [] = []
dupli_1 (x:xs) = x:x:dupli xs

dupli_2 = concatMap (replicate 2)
```

Suppose that the student begins to define a solution by writing `dupli xs = ?` and checks for progress. Ask-Elle gives feedback that the student has drifted from the strategy and that no additional feedback can be provided. This feedback might confuse a beginner Haskell programmer who might wonder why this simple partial program is considered drifted. The attempt could not be matched to a model solution simply because no existing model solution is defined using pattern matching on an input argument with a general pattern. Instead of requiring the teacher to write another model solution to be able to match this (or leave the student confused), *η-conversion* could be applied to `dupli_2` to match the student's definition. η-expanding this example goes as follows:

```
dupli_2 = concatMap (replicate 2) →η
dupli_2 xs = concatMap (replicate 2) xs.
```

Another possibility is to perform η-conversion in the other direction on the student program, that is, η-reducing the student solution. However, it is not possible on the

considered student attempt since the right-hand side of the definition is just a hole. It would have been possible to perform η-reduction if the student also would have included the pattern `xs` on the right-hand side:

```
dupli xs = ? xs →η   dupli = ?.
```

Since the first attempt failed, the student did not get any hints on how to proceed. However, since the task is to duplicate a list, it makes sense to replicate the elements in the list twice. Imagine that the student proceeds to write

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

Rechecking progress, Ask-Elle still cannot recognise the attempt and states: "You have drifted from the strategy in such a way that we can not help you anymore." This feedback is not helpful, considering that a solution can be developed from here in a single step, namely:

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

Once again, this could have been recognised with more reliable η-conversion.

Several model solutions covering different strategies can help recognise more attempts while allowing the teacher to decide which approaches should be considered correct. Specifying more model solutions to cover all syntactical variants of a particular program is not a feasible approach, and that is why normalisation and additional program analysis are needed.

3.1 Function Bindings and Case Expressions

Assume that the set of model solutions for the `dupli` exercise is defined as in the previous section and consider the following student solution:

```
dupli xs = case xs of
    [] -> []
    (x:xs) -> x:x:dupli xs
```

This program cannot be recognised as a match even though it is similar to the first model solution `dupli_1` from before. It seems like the transformation that rewrites function bindings to case expressions in Ask-Elle does not succeed in matching this definition with the `dupli_1` definition. By refining this transformation, the student attempt above could have been recognised as correct.

3.2 Local vs Top-level functions

First, let's introduce another example of an Ask-Elle exercise that will reoccur throughout the report. The exercise is to define the function

```
myreverse :: [a] -> [a]
```

that reverses a list. A solution to this exercise could, for example, be implemented as:

```
myreverse = reverse' []
    where reverse' acc []     = acc
          reverse' acc (x:xs) = reverse' (x:acc) xs
```

which indeed is one of the model solutions. Instead, if the student would write

```
myreverse = reverse' []

reverse' acc []     = acc
reverse' acc (x:xs) = reverse' (x:acc) xs
```

it would not be recognised as a matching program since `reverse'` were moved out to the top-level scope. This could be recognised by *inlining* the definition of reverse' as a local function. The same problem arises using let constructs, for example:

```
myreverse = let reverse' acc []     = acc
                reverse' acc (x:xs) = reverse' (x:xs) xs
            in reverse' []
```

This attempt would be categorised as TestPassed by Ask-Elle, as it is correctly implemented but cannot be recognised due to shortcomings in the normalisation. After normalisation, preferably, all the above examples would be equivalent.

3.3 Redundant and Overlapping Patterns

Programs with redundant patterns now end up in the Unknown or TestPassed category if otherwise correct. As an example, consider the following definition of our other running example `dupli`, where the second pattern is redundant since it is covered by the pattern in the last function binding:

```
dupli []     = []
dupli [x]    = [x,x]
dupli (x:xs) = [x,x] ++ dupli xs
```

The preferred outcome is to give specific feedback that the student uses a redundant pattern rather than just saying that the solution differs from any of the model solutions.

A related problem is *overlapping patterns*. A function that contains some function binding that overshadows another one or a case expression that has several cases

that match the same input is said to have overlapping patterns. For example, if we define:

```
dupli xs  = concatMap (replicate 2) xs
dupli [x] = [x,x]
dupli []  = []
```

The first pattern `xs` matches *any* list; thus, the later patterns will never be reached. This solution would also be categorised as TestPassed with no additional feedback. It would once again be better with specific feedback pointing out that the function has overlapping patterns.

3.4 Limitations of the internal AST

The current AST representing Haskell programs in Ask-Elle is restricted to representing a Haskell module only containing function definitions. Type signatures are allowed in compilation but thrown away since it is currently not possible to represent them in the abstract syntax. This means a student can declare a type signature that does not match the definition of the function as long as the function name and definition correspond to the given exercise.

Other Haskell constructs like import statements, type class definitions, data type definitions, and similar are not parsable, which limits the type of possible exercises. Exercises on defining new data types are a reasonable extension of Ask-Elle but would require significant changes to the internal representation of Haskell programs.

4

Method

The primary method to improve recognition in Ask-Elle is to utilise GHC's program transformations for normalisation and its warning messages for additional feedback. GHC is itself written in Haskell, which makes it possible to use GHC as a Haskell library, allowing some control over the internal passes in the compiler. The GHC API[1] offers functions to parse, type check, desugar and simplify a Haskell program separately. Compiling student programs with GHC and using GHC's internal representation of programs instead of Ask-Elle's customised AST allows using the full power of Haskell and reusing many features from GHC. For example, using Core as the internal representation of student and model solutions allows for checking type signatures in student submissions and importing standard libraries. It also enables future exercises on, for example, data type definitions or type classes. The Core Linter that comes with GHC also provides a way of checking that programs are still type correct after applying different transformations.

4.1 Representing Holes

Since holes are an essential concept in how Ask-Elle provides incremental feedback, changing from the customised AST of Ask-Elle that allows holes as a replacement for any term requires a suitable counterpart that is also parsable by GHC. GHC's *typed holes* is a good candidate for representing holes in Ask-Elle for two main reasons. First, they are a built-in feature of GHC, and their types can be inferred automatically. Second, GHC has support for reporting possible valid hole fits for typed holes, which allows extending Ask-Elle with simple program synthesis in the future. We can enable the "defer typed holes" option in GHC to avoid the compilation terminating with a type error and let the compilation continue past the typechecker. However, no explicit constructs exist for holes on the Core syntax level. Rather, a typed hole gets compiled into a case expression containing a variable called "typeError" and a literal with the corresponding error message. An example of how (quite unreadable) a typed hole looks like when compiled to Core is displayed in Figure 4.1.

[1] https://gitlab.haskell.org/ghc/ghc/-/wikis/commentary/compiler/api

```
Case (App (App (App (Var typeError)
(Type (TyConApp BoxedRep [TyConApp Lifted []])))
(Type (TyConApp () [])))) (Lit Prog.hs:7:9: error:
    Found hole: _ :: [a] -> [a]
    ...
    Valid refinement hole fits include
      id (_ :: p)
      where id :: forall a. a -> a
       with id @p
       (imported from 'Prelude' at Prog.hs:1:8-11
(deferred type error))) wild
(FunTy {ft_af = AnonArgFlag, ft_mult = TyConApp Many [],
ft_arg = TyConApp [] [TyVarTy a],
ft_res = TyConApp [] [TyVarTy a]}) []))
```

Figure 4.1: Part of the AST of a Core program corresponding to a typed hole expression (_).

4.2 Using GHC's Internal Transformations

Many of the proposed program transformations needed for normalising programs in Ask-Elle, like β-conversion, η-conversion and other program rewrite transformations, are already implemented in GHC. The idea is to use Core to represent the student programs internally in Ask-Elle. This also means that the internal transformations in GHC can be utilised and reduce the amount of work on transformations specifically implemented for Ask-Elle. Although GHC implements a large number of program transformations, the purpose of them is to optimise the compiled code. Fast code is different from the goal of Ask-Elle; the only concern of Ask-Elle is the ability to determine if two programs are syntactically equal. Hence, GHC's transformations are not always helpful in normalisation, and manual intervention is still needed.

It is also important to find the right configuration of GHC, such that GHC's transformations can be maximally utilised while at the same time not losing any crucial information from the original program. For example, it must be evaluated which transformations should be disabled and if the simplifier pass should be included in the compilation or not.

4.3 Using the RULES pragma

GHC's RULES pragma can be used to implement additional rewrite rules in a simple way. For example, GHC can be instructed to rewrite an expression using the function `map` twice to use a single call to `map` (map fusion) by a pragma defined as:

```
{-# RULES "map fusion"
forall f g xs. map f (map g xs) = map (f . g) xs #-}
```

The downside of using the RULES pragma to introduce new transformations is that every rule must be inserted into the source code to take effect, and even then, like other transformations in GHC, there is no guarantee that a rule specified with the RULES pragma will be applied. GHC might detect that it has no performance benefits or that some function called might be inlined, for example. Rewrite rules defined with the RULES pragma are also constrained to certain kinds of expressions. For example, it is impossible to specify a rule to remove redundant if-expressions with the rules pragma since the left-hand side must be on the form `f x ... y` where f is a specific function. Additionally, rewrite rules defined by the RULES pragma are first fired in GHC's simplifier (if also the `Opt_EnableRewriteRules` option in GHC is set) and hence using them requires running the simplifier.

4.4 Matching Holes

With the improved normalisation, an alternative approach for matching holes can be implemented. Ask-Elles's current approach is to generate several intermediate versions of the model solution and try to find an intermediate solution that matches the student program; and, from there, derive the term matching the hole, or the *hole-fit*, as explained in subsection 2.2.2. With improved normalisation, hole-fits can be retrieved directly by matching the Core AST of the student program against the matching model program.

For example, if the student has defined:

```
dupli xs = concatMap _ xs
```

we should be able to extract the hole-fit `(replicate 2)` from the model solution defined as:

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

If a student program contains holes and we can recognise that it can be refined to a model solution, the Core ASTs of the student program and the matching model solution are matched against each other, expression by expression. If a hole is encountered, the corresponding expression in the model solution is retrieved. This approach is presumably faster than generating and searching a list of intermediate solutions, especially as the function size grows.

Retrieving matching Core expressions is easy; on the other hand, it is not as straightforward to display this information to the student in a comprehensible way. Printing the Core expression as-is is not a good solution, and pretty-printing the expression is difficult due to loss of information from desugaring. For example, we cannot know if a function application originally was written in prefix or infix notation after desugaring since all applications are rewritten as prefix expressions in Core. However, `Tick` expressions, used to annotate the source code, can be used to retrieve the original parsed expression in a sequence of steps. A `Tick` contains a `GenTickish` and wraps another Core expression. The `GenTickish` of interest here is the `SourceNote` type, which contains a source span referring to the original location of the expression wrapped by the `Tick`.

A source span is, a bit simplified, a line number and a start and an end column corresponding to the location and span of an expression in the source file. The source span from a `SourceNote` can be used to map a Core expression with the source location in the original module. In the parsed AST, all expressions are *located*, meaning that we can use the source location from the `SourceNote` to extract a parsed expression at a given location. For parsed expressions, the ghc-exact-print library[2] provides functionality to print a parsed expression exactly as it was written in the source file. Hence, obtaining the original definition of a hole-fit from a Core expression is possible.

An important question remains: where do the `Tick` expressions come from? It is possible to manually wrap `Tick`s on any expression in the typechecked AST, but these are unfortunately not kept after desugaring. GHC automatically generates `SourceNote` `Tick`s if the `Opt_InfoTableMap` option is set. We can utilise those for mapping a Core expression back to its corresponding parsed expression. This means that mapping Core expressions to parsed expressions relies on GHC's placement and generation of `Tick`s, making hole-matching less robust. A substantial drawback is that we cannot control the placement nor ensure that every subexpression in a Core program is wrapped in a `Tick`. Furthermore, the source span of the `Tick` surrounding the first expression on the right-hand side of a function declaration covers the whole function binding (and not only the first expression), which imposes some additional difficulties in retrieving the correct expression from the parser AST.

4.5 Using GHC's Warnings

We can use GHC's warning messages to retrieve additional feedback on student programs. Which warnings GHC reports can be configured by setting or disabling particular warning flags. In the context of Ask-Elle, the following flags are of particular interest:

WarnTypedHoles Gives hole-fit suggestions for each typed hole in a program. It can be used for additional feedback when students use another approach than any of the model solutions.

WarnOverlappingPatterns Warns about overlapping patterns and where they occurred.

WarnIncompletePatterns Warns about incomplete patterns and where they occurred.

Another tool that can be used for generating additional feedback is HLint, which is a tool suggesting possible improvements to Haskell code[3]. To get feedback from HLint in the same format as other warnings from GHC, we can use Splint[4] - an existing HLint compiler plugin to GHC. This plugin creates regular GHC warnings from HLint suggestions.

[2] https://hackage.haskell.org/package/ghc-exactprint
[3] https://github.com/ndmitchell/hlint
[4] https://github.com/tfausak/splint

HLint suggestions are valuable when wanting to teach coding style and best practices. For example, when to avoid lambdas, if the program contains redundant brackets or when η-reduction is possible. HLint can also suggest possible rewrites, for example, if defining `dupli` as:

```
dupli  = concat . map (replicate 2)
```

HLint would give the following suggestion:

```
Use concatMap, Perhaps: concatMap (replicate 2)
```

5

Program Transformations on Core

Recall the problems related to (the lack of) different program transformations from chapter 3. The transformations applied in the process of compiling Haskell source code into Core solve the following issues directly:

Local and Top-level functions: GHC's inlining of non-recursive local definitions from where and let clauses, in many cases, allows arbitrary use of local helper functions. An exception to this is type differences that may arise if local functions are given explicit type signatures or if the inferred type of a local function is more general than the top-level function it is defined in. The latter problem is discussed further in section 8.2.

Overlapping patterns: Overlapping patterns (but not redundant patterns) are removed by GHC and a warning is generated. We need to be able to pass this information to the student as feedback; otherwise, the student could get incorrect feedback.

Capture avoiding renaming: GHC's renamer allows keeping the name as declared in the source code while still avoiding name capture by attaching uniques to variables.

β-abstraction: β-abstraction is always performed.

Pattern matching and cases: Functions with several pattern bindings and functions using a case statement on a single variable pattern are α-equivalent in Core.

When we also run the simplifier pass, we get:

Redundant equality checks: The simplifier removes redundant equality checks, e.g. an expression like `if x==y then True else False` is simplified to `x==y`.

It is not possible to rely solely on GHC's internal transformations to normalise Haskell programs in Ask-Elle. We explain in section 7.2 why simply compiling programs into Core does not give the desired normalisation. Before GHC applies a particular transformation or rewrite rule to an expression, it analyses if this will pro-

vide some performance gain. As explained in subsection 2.3.3, the decision to inline an expression is heuristical, which makes it non-deterministic. For normalisation, it is desirable to deterministically apply transformations to know that a particular expression is always normalised in a certain way. The GHC API does not offer any ways to control these heuristics, and changing this behaviour would require changes to the GHC source code. Instead, we have implemented several new Core-to-Core transformations to deal with this non-determinism.

5.1 New Core-to-Core Transformations

When a Core program is obtained, many of the transformations we want to include in normalisation are already applied. The remaining issues are mostly due to the non-determinism in GHC's application of its transformations mentioned above. For example, both inlining and η-conversion are reimplemented to obtain a deterministic version of these transformations and applied to the Core program obtained from GHC.

It is desirable to include the simplifier pass in compilation to be able to use the RULES pragma for additional rewrite rules and let GHC rewrite default case alternatives when possible, as explained in subsection 2.3.3. Including the simplifier pass in the compilation of student programs also requires some preprocessing transformation between the desugarer and the simplifier pass, which are presented in the next subsection. The remaining transformations are presented in their order of application.

A note before diving into the implementation of the new transformations: The small number of constructors in the Core expression data type also forces us to do string comparisons and other "hacks" to check certain conditions that are not directly extractable from the datatype. For example, any kind of conditional, like pattern matching, guards and if-expressions, as well as pattern errors and typed holes, are compiled into a case expression in Core. To know if a particular Core case expression corresponds to, for example, a typed hole requires checking that the expression contains a variable called "typeError" and a message reporting hole-fits.

5.1.1 Preprocessing Transformations

As a first step, some preprocessing transformations are applied to remove redundant information from the AST and to be able to include the simplifier in the compilation. For easier handling of hole expressions, the cumbersome expression displayed in section 4.1 is replaced by a single variable of the same type prefixed with the string "hole". This is done by traversing a program, and if finding a case expression that contains a deferred type error variable, the whole case expression is replaced by a new variable. Because typed holes are treated as type errors and compiled into case expressions (which are strict), we must perform this transformation to prevent the simplifier from removing code occurring after a hole. A similar transformation removes *pattern errors*, which are inserted into the AST when GHC detects that a program has incomplete patterns. Pattern errors are represented by a variable

named “patError”, either contained in a case expression with an empty list of alternatives similar to typed holes or simply as a variable in a case alternative. The first kind of pattern error expressions are removed for one more reason: they contain literal strings that refer to the source file, and hence two pattern errors will not match without making exceptions for literal strings when matching programs.

5.1.2 Inlining Bindings

A Core program is a list of binders, and since they are matched one by one, the similarity check could fail if two programs contain different numbers of top-level binders. This situation could arise if one function uses a local helper function (defined with let or where), and the function it is matched against uses an equivalent, albeit top-level helper function. This can be solved by inlining. Since Ask-Elle exercises consist of a single module, it is safe to inline any top-level function. We only inline top-level functions that are used just once. Not because of performance, but if we would extend exercises to include several functions, we would like to give feedback on any named function. We can easily change this if the need arises. Hence, all top-level binders that are used in exactly one other top-level binder are inlined.

The effect of this transformation is explained by showing what an example program looks before and after the transformation is applied (the arrow in the code below represents the application of the transformation). This is shown in Haskell source code to enhance readability for a reader unfamiliar with Core syntax, even though the transformation is actually implemented on the Core AST. In the first example, the top-level recursive function `length'` is inlined by pushing it inside a letrec in the definition of the `length` function:

```
length :: [a] -> Int
length xs = length' xs

length' :: [a] -> Int
length' []     = 0
length' (x:xs) = 1 + length' xs
```

↓

```
length :: [a] -> Int
length xs = let length' :: [a] -> Int
               length' [] = 0
               length' (x:xs) = 1 + length' xs
           in length' xs
```

A non-recursive binder is inlined directly by replacing the variable representing a function call with its expression. The result of this transformation on two non-recursive functions is shown below:

```
sum :: [Int] -> Int
sum = foldl add 0

add :: Int -> Int -> Int
add x y = x + y
```

↓

```
sum :: [Int] -> Int
sum = foldl (\x -> \y -> x + y) 0
```

Note that `add = \x -> \y -> x + y` is the β-expanded form of `add x y = x + y`. All expressions are β-expanded in Core.

After the inlining pass, if some top-level recursive binders remain, they are pushed inside a fresh non-recursive binder as a letrec. The effect of this transformation is best explained with an example and is again visualised with concrete Haskell programs before and after applying the transformation:

```
length :: [a] -> Int
length []     = 0
length (x:xs) = 1 + length xs
```

↓

```
length :: [a] -> Int
length = let fresh :: [a] -> Int
             fresh []     = 0
             fresh (x:xs) = 1 + fresh xs
         in fresh
```

Note the explicit type signature in the let binding and that `fresh` is a new variable. *Without* the explicit type signature in the letrec, the above example would not be accurate. The resulting program above would not be α-equivalent to a program defined as (with no explicit type signature):

```
length :: [a] -> Int
length = let l [] = 0
             l (x:xs) = 1 + l xs
         in l
```

The type of `l` would be inferred to `l :: Num n => [a] -> n` by GHC, and the prelude function `fromInteger :: Num n => Integer -> n` would be applied to the result of `l` to comply with the type signature of the top-level function `length`. The handling of types and type constraints is not straightforward in Core and is discussed more in subsection 5.1.7 and section 8.2.

5.1.3 Replacing the Case Binder

When the simplifier replaces `DEFAULT` case alternatives with concrete case alternatives, for particular programs, all occurrences of the scrutinee in the alternatives of a case expression is replaced by the case binder. Of, course this is better for performance since the scrutinee does not have to be reevaluated, but it creates a problem when trying to match incomplete student programs that has a single function binding an thus no case expressions. To avoid mismatches due to using the case binder or the scrutinee, we revert this transformation applied by the simplifier. The scrutinee is substituted back at all occurrences of the case binder if the case binder is not a "special variable", such as a primitive variable.

For example, consider the following program:

```
dropevery [] _ = []
dropevery list count = (take (count-1) list) ++
                       dropevery (drop count list) count
```

It is compiled into the following Core program if including the simplifier but no additional transformations:

```
[Rec [(dropevery, Lam a (Lam list Lam count (Case (Var list) wild
(TyConApp [] [TyVarTy a]) [Alt (DataAlt []) [] (App (Var [])
(Type (TyVarTy a))),Alt (DataAlt :) [ipv,ipv] (App (App (App (Var ++)
(Type (TyVarTy a))) (App (App (App (Var take) (Type (TyVarTy a)))
(App (App (App (App (Var -) (Type (TyConApp Int []))) (Var $fNumInt))
(Var count)) (App (Var I#) (Lit 1)))) (Var wild))) (App (App (App
(Var dropevery) (Type (TyVarTy a))) (App (App (App (Var drop)
(Type (TyVarTy a))) (Var count)) (Var wild))) (Var count)))])))))]
```

The case binder `wild`, binding the result of the scrutinised list, replaces references to the variable `list` in the inner expression. Since partial solutions to the `dropevery` exercise may lack a case expression, any referrals to the list input argument will mismatch from the model solution. The discussed transformation will replace all occurrences of `Var wild` with `Var list` in the above example.

5.1.4 Removing Redundant Equality-checks

Removing redundant equality checks is a simple transformation. For example,

```
if x == y then True else False
```

can be simplified to

```
x == y
```

which GHC's simplifier pass does automatically. Although this transformation is good for both optimising and normalising purposes and would help recognise many student programs (this is a common beginner mistake!), it might not be desirable from a pedagogical point of view. Hence, this transformation might be configured as optional and specified by the teacher setting up the exercise. Since it is not possible

to configure the simplifier to use only particular transformations, we duplicated the transformation to include it for the desugaring pass when compiling programs; to make a fair comparison when evaluating which compiler pass is most suitable for normalisation in Ask-Elle. In Core, if-statements and guards will be a case expression on booleans, so this is the only kind of Core expression to consider. The same simplification can also be made for non-equality (`/=`) checks. Finally, if an if-statement or guard is used to reverse the logic of an equality operator, the expression can be simplified by replacing the equality/non-equality operator in the expression. This is *not* done by GHC's simplifier. The implementation of this transformation is relatively straightforward:

For any case expression in a Core program, if the case *scrutinee* includes an equality or non-equality operator, it is a candidate for this transformation. Then, if the list of alternatives only contains boolean data constructors and returns the same boolean constructor as in the alternative, the case expression can be dropped and replaced with the scrutinee. If the scrutinee contains an equality or non-equality operator, and the list of alternatives only includes boolean expressions but with reversed logic (if true, then false), the `==` operator is replaced by the `/=` operator, and vice versa. That is, expressions like

```
f x y | x == y     = False
      | otherwise = True
```

are simplified to

```
f x y = x /= y.
```

5.1.5 η-conversion

As described in subsection 2.3.3, GHC's built-in η-expansion and η-reduction are unreliable for normalisation, especially for programs including holes. We implement an additional η-conversion pass, where the simplest choice is reduction since it can be performed without relying on arity information from GHC. The implementation is quite simple; Core programs are traversed to find all lambda abstractions over an application. If the body of the lambda is applied to the bound variable, e.g. if we have an expression like $\lambda x.f\ x$, the expression is reduced to f. The actual transformation takes place in the function `etaRed`, called during the traversal of the Core program (see Figure 5.1). The lambda abstraction and the application are dropped if the lambda head is the argument of the application.

```
etaRed :: CoreExpr ->  Maybe (CoreExpr)
etaRed (Lam v (App f arg))
    | isEtaReducible f arg v = return f
etaRed (Lam v (Let b (App f arg)))
    | isEtaReducible f arg v = return (Let b f)
etaRed _ = Nothing
```

Figure 5.1: The implementation of η-reduction for a Core expression with some simplifications (cases handling Tick expressions are removed since they do not affect the semantics of an expression, just the pattern matching).

Reducing Core programs requires additional checks compared to the lambda-calculus definition – that the lambda-bound variable cannot be free in the function applied to the variable. All further checks are made in the `isEtaReducible` function given in Figure 5.2.

```
isEtaReducible :: CoreExpr -> CoreExpr -> Var -> Bool
isEtaReducible f arg v = case arg of
        Var v' | v == v'
          ,not (isTyVar v)
          ,not (isEvVar v)
          ,not (ins v f) -- v appears free in f
          ,not (isLinearType (exprType f))
          -> True
        Tick _ e -> isEtaReducible f e v
        _ -> False
```

Figure 5.2: Boolean function determining if an expression is η-reducible or not.

First, type lambdas and type evidence lambdas should not be reduced since removing the types would create ill-typed programs and remove the possibility of typechecking programs after normalisation with the Core Linter. Second, `f` can itself be an application of another function to the lambda bound variable `v`; it must be ensured that `v` does not appear somewhere in the expression `f`. Third, linear functions are not η-reduced. The reason for this is that all fields in algebraic data types are linear by default [23]. This is mostly an implementation artefact from the Linear Types language extension to GHC [24], [25]. However, it means that GHC η-expands all data constructors with arity greater than zero when used as a term and introduces a linear type constraint on its arguments. For example, consider the "smart" constructor for `Just` below:

```
just :: a -> Maybe a
just = Just
```

Compiling this program to Core gives the following AST:

```
[NonRec just (Lam a (Lam ds (App (App (Var Just)
(Type (TyVarTy a))) (Var ds))))]
```

We can see that the `Just` constructor has been η-expanded by GHC. Inspecting the types of the variable `Just` in the Core program above shows that it has type multiplicity one, which in GHC means that it is a *linear* function, and its argument can be consumed at most once. Using the notation from linear type theory [26], the constructor `Just` has the type

```
forall a.  a ⊸ Maybe a,
```

while the function `just` is typed as

```
forall a.  a -> Maybe a.
```

If then manually η-reducing the inner lambda, leaving the variable `Just` on the right hand side, gives a type mismatch. Therefore, η-reducing data constructors with arity greater than zero would create ill-typed programs. This is avoided by checking for linear types when η-reducing. Checking for linear types is a simplification of the actual condition (`f` is a non-nullary data constructor). The problem is that `f` could be an application of a data constructor on, for example, a type, which would not saturate the data constructor, and we do not want to η-reduce such expressions. However, this requires looking deeper into the expression `f` and the simpler constraint is used to avoid a lot of special cases. This should not catch any unwanted cases since the linear-types extension would probably never be used in an exercise for beginners, and linear types should thus not be present anywhere else than in data constructors.

5.1.6 Renaming

Since GHC already assigned unique names to all variables, renaming Core programs to match a student program with a model solution is relatively straightforward. The name of the function asked for in the exercise is given as a string (this can be extended to a list of strings if multi-function exercises are offered in the future). Any variable matching the exercise name is *not* renamed, while all other non-special variables are renamed with fresh names in a predefined order. The renaming procedure is applied after the transformations discussed in the preceding subsections since these transformations might reorder, remove or introduce new variables. Additionally, Ask-Elle must handle matching of *incomplete* programs, so the renaming procedure must be slightly more clever than simply renaming all variables in the order they appear.

The renaming function traverses a program within a state monad containing a map from old variables to renamed variables, the exercise name and five integer counters used in variable generation. The counters are used to handle different types of variables with the sole reason of dealing with incomplete student programs. For instance, when a student only defines a single function binding with no or a general pattern for the input argument, the compiled Core program will not contain a case expression. Renaming the case-binder[1] in the model solution would offset all succeeding variable names, and there would be a name mismatch when trying to

[1] A Core case expression binds the result of the scrutinee with a variable.

match the single function binding from the student program with any of the case alternatives in the model solution. How partial and complete programs are matched against model solutions are described in detail in section 6.2.

A similar situation can arise for recursive model solutions that are rewritten to a non-recursive function with a recursive let-expression by the transformation presented in subsection 5.1.2. If a student attempts to implement an explicit recursive function but has not yet included the recursive call, the function is not (yet) bound as a recursive function. The student program would then not be affected by the aforementioned transformation nor contain any fresh let-binders, and the rest of the variables in the program would be offset by one. Hence, top-level binder variables, case-binder variables, lambda-bound variables and other variables have separate "name supplies" implemented as counters.

If a variable is encountered, it is passed as an argument along with information about which kind of variable it is (case-binder, binder, lambda variable or a general variable) to the `renameVar` function displayed in Figure 5.3. Another counter is used for "resetting" the renaming procedure for every top-level binder, meaning that for every binder in the list that makes up the Core program, this counter is increased. Every variable in the first binder will be prefixed with a 1, every variable in the second binder of the list will be prefixed with a 2, and the pattern continues. This allows future exercises on multiple functions or other constructs, such as data types or type classes. If one part of the student program is incomplete, it will not offset the renaming of other functions or binders in the program.

We demonstrate the renaming transformation on the small Core AST of the `length` function discussed previously, defined as:

```
length :: [a] -> Int
length = let fresh []     = 0
             fresh (x:xs) = 1
         in fresh
```

Before renaming, the Core AST for this function would look as follows:

```
[NonRec length (Let Rec [(fresh,Lam ds (Case
(Var ds) wild (TyConApp Int []) [Alt (DataAlt [])
[] (App (Var I#) (Lit 0)),Alt (DataAlt :) [x,xs]
(App (App (Var +) (App (Var I#) (Lit 1))) (App
(Var fresh) (Var xs)))]))] (Var fresh))]
```

The new names for the different kinds of variables are prefixed with letters: 'b' for binders, 'l' for lambdas, 'cb' for case-binders and 'v' for general variables. The first number corresponds to the top-level binder (not binders in a let-expression) in which the variables are contained. Since there is only one top-level binder in this program, all variables first have the integer 1 appended to the variable name. The second number is increased for every variable of that particular "variable kind"; the renamed version of the program above is displayed below:

```
[NonRec b10 (Let (Rec [(b11,Lam l10 (Case (Var
l10) cb10 (TyConApp Int []) [Alt (DataAlt []) []
(App (Var I#) (Lit 0)),Alt (DataAlt :) [v10,v11]
(App (App (Var +) (App (Var I#) (Lit 1))) (App
(Var b11) (Var v11)))])])]) (Var b11))]
```

The implementation of the renaming transformation is quite straightforward. The program is traversed, and if a variable is encountered, it is passed to the function `renameVar` given in Figure 5.3.

```
renameVar :: VarType -> Var -> State St Var
renameVar vty v | isSpecialVar v = return v
                | otherwise = do
        vars <- gets env
        name <- gets exerName
        if varNameUnique v == name
            then return v
            else case Map.lookup v vars of
                Just n | varType n `eqType` varType v -> return n
                       | otherwise -> renameV vty v
                Nothing -> renameV vty v
```

Figure 5.3: The main logic of the renaming procedure, the actual renaming takes place in the function `renameV`. The data type `St` in the `State` monad is used to store the counters and special names used in name generation.

If the variable name does not match the exercise name, is not a global id, a type variable, an evidence variable or any other type of "special" variable (a primitive operator, data constructor, etc.), it is not renamed, and the input variable is returned. If a variable meets none of the conditions above, it is checked if the variable can be found in the map of variables contained in the state monad. If found in the map, a variable with this name and unique has already been encountered and renamed. However, the equality comparison of variables implicitly used in the lookup does not consider the type of the variables. It is therefore also checked that the variable found in the map has the same type as the variable currently subject to renaming. This must be checked since GHC reuses generated variable names and uniques in some cases. If simply returning the variable found in the map without comparing the types of the variables, a variable with the same name and unique (but with a different type) might wrongfully be replaced and cause a type error. If the types match, the already renamed variable is returned. If not, or if the variable is encountered for the first time, it is renamed with a new name. The actual renaming takes place in the `renameV` function (see Figure 5.4).

```
data VarType = TopBind | CaseBind | GenVar | LamVar

renameV :: VarType -> Var -> State St Var
renameV vty v = do
    vname <- getVName vty
    let v' = updateVarName v vname
    modify $ \s -> s {env = Map.insert v v' (env s)}
    return v'
```

Figure 5.4: The data type used to separate the renaming of core binders, case binders, general variables and lambda variables and the renaming function. The renaming function retrieves a fresh variable name for the given variable kind and updates the variable name and the variable map.

5.1.7 Removing Types and Type Evidence

Type variables and type evidence variables are passed around in Core expressions and are needed to typecheck Core programs [21]. These variables clutter the AST with lots of irrelevant information for checking if any two programs are syntactically equal. Any ill-typed program would already have been rejected by GHC's typechecker. For example, consider the following two programs:

```
factorial :: (Eq t, Num t) => t -> t
factorial 0 = 1
factorial m = m * factorial (m - 1)
```

```
factorial :: (Eq t, Num t) => t -> t
factorial = let f 0 = 1
                f m = m * f (m - 1)
            in  f
```

They are clearly not syntactically equivalent, nor after being compiled into Core. If we look at the pretty-printed (with some irrelevant details removed) Core program without any additional transformations, the first definition of factorial looks as follows:

```
[factorial :: forall t. (Eq t, Num t) => t -> t
 factorial
   = \ (@t)
       ($dEq :: Eq t)
       ($dNum :: Num t)
       (ds :: t) ->
       case == @t $dEq ds (fromInteger @t $dNum 0)
       of {
         False ->
           * @t
             $dNum
             ds
             (factorial
```

```
                @t
                $dEq     -- evidence variables in the recursive case
                $dNum    -- evidence variables in the recursive case
                (- @t
                   $dNum
                   ds
                   (fromInteger @t $dNum 1)));
          True -> fromInteger @t $dNum 1
        };
]
```

The recursive case of the factorial function contains both the evidence variables for the argument to make the equality comparison in the recursive case. The second program, in pretty-printed Core, looks like follows:

```
[factorial :: forall t. (Eq t, Num t) => t -> t
 factorial
   = \ (@t)
       ($dEq :: Eq t)
       ($dNum :: Num t) ->
       letrec {
         f :: t -> t
         f
           = \ (ds :: t) ->
               case == @t $dEq ds (fromInteger @t $dNum 0)
               of {
                 False ->
                   * @t
                     $dNum
                     ds
                     (f
                        (- @t
                           $dNum  -- only the Num evidence variable
                           ds
                           (fromInteger @t $dNum 1)));
                 True -> fromInteger @t $dNum 1
               }; } in
        f
]
```

Where only the Num evidence variable is passed into the recursive case. Hence, the transformation needed to match these two programs could not simply be to push the top-level recursive function inside a letrec, as described in subsection 5.1.2. We would also need to correctly handle type and evidence variables to make these programs match.

Instead of manually trying to push types and evidence variables around when inlining

or rewriting recursive functions as letrecs at Core level, in similarity with GHC, all types and type evidence are removed from the Core AST as a final pass. There are some exceptions because some types can simply not be removed. Case expressions, for instance, require a type. Type removal is done by traversing the AST, removing all abstractions, applications and binders only consisting of types and type evidence. After this transformation, it is no longer possible to typecheck a program with the Core Linter. However, this should not be an issue since it is possible to test the preceding transformations with the Core Linter before removing types, which is the last transformation.

5.2 Additional Rewrite Rules

Some additional transformations are specified with GHC's RULES pragma. As discussed, rewrite rules defined by the RULES pragma are not guaranteed to be applied but can help recognition in particular cases. A rule declaration defines a transformation where the left-hand expression is rewritten to the right-hand expression. The following rewrite rules are defined and enabled by default:

- `forall f g xs. map f (map g xs) = map (f . g) xs`
- `forall f g x. f $ g x = f (g x)`
- `forall f xs. map f xs = [f x | x <- xs]`
- `forall f xs. concat (map f xs)  = concatMap f xs`
- `forall f. concat . map f = concatMap f`

Additional ones like `forall xs. null xs = xs == []` could, for example, be added, but from a pedagogical perspective, we should let such rewrites be suggested by HLint instead of accepted as a match directly.

The RULES pragma can, to some extent, replace the special pragmas for defining alternatives in Ask-Elle. This set can be extended for a particular exercise by the teacher defining the exercise by writing new ones. New rewrite rules for an exercise could, for example, be defined in the configuration file for each exercise.

6

Matching Programs and Deriving Feedback

The student program is first compiled and normalised before feedback can be generated. Additional warnings encountered during compilation must also be retrieved from GHC. We can then match the normalised program and check if it matches any of the model solutions. Matching two programs is defined by two separate similarity functions: one for checking syntactical equivalence for complete programs and one for partial programs. Finally, we can compose all the obtained information and generate feedback for the student.

6.1 Compiling Programs and Retrieving Warnings

Compiling a student's attempt and all the model solutions for the exercise in question is the first step towards generating feedback. A program is first parsed and typechecked before being desugared into Core. The compiling function, displayed in Figure 6.1, takes the exercise name and a file path, used for storing the student attempt, and outputs a `CompInfo` in the `IO` monad. The `CompInfo` data type contains all information later needed in feedback generation: the compiled Core program, the parsed source, the list of compiler warnings, the map containing all renamed variables and the exercise name.

```
compile :: ExerciseName -> FilePath -> IO CompInfo
compile name fp = defaultErrorHandler
                  defaultFatalMessager
                  defaultFlushOut $
                  runGhc (Just libdir) $ do
    (pcore,psrc,wref) <- toSimplify fp
    (pnorm,vars) <- liftIO $ normalise name pcore
    ws <- liftIO (readIORef wref)
    let warns = nubBy uniqWarns ws
        prog  = removeTyEvidence pnorm
    return $ CompInfo prog psrc warns vars name
```

Figure 6.1: The function `toSimplify` does the actual calls to the GHC API. The returned program is normalised, and the map of renamed variables is also obtained from normalisation. Finally, the warnings are read from the `IORef` before composing all information into the `CompInfo` constructor.

The `toSimplify` function is responsible for initialising the GHC environment, retrieving the parser and Core AST and applying the preprocessing transformations on the Core program before feeding it into GHC's simplifier. The GHC environment (`HscEnv`) is initialised by the `initEnv` function (see Figure 6.2). `initEnv` sets all flags and options in GHC and initialises a mutable `IORef` used as input to a new logging function. Instead of writing errors and warnings to the standard output (stdout), the `writeWarnings` function appends warnings to a list contained in the `IORef`. The new log function is set in the `HscEnv` by using the `pushLogHookM` function from the GHC API. All warnings encountered during compilation are stored in the `IORef`, which can be retrieved from the `IORef` at the end of the compilation. The target (the target source file) is loaded with the HLint plugin added to the compilation passes.

```
initEnv :: Bool -> [GeneralFlag] -> FilePath -> Ghc(IORef [Warning])
initEnv setdefaultFlags flags fp = do
     env <- getSession
     setFlags setdefaultFlags flags
     ref <- liftIO (newIORef [])
     pushLogHookM (writeWarnings ref) -- set new log action in GHC
     target <- guessTarget fp Nothing
     dflags <- getSessionDynFlags
     loadWithPlugins dflags target [StaticPlugin $ PluginWithArgs
               { paArguments = [],
                 paPlugin = plugin -- set plugin function
               }]
     return ref
```

Figure 6.2: The `initEnv` function is responsible for setting dynamic flags and options, adding a new log action to the compiler environment and including the HLint plugin by giving the function `plugin` from the Splint package as an argument to `loadWithPlugins` function from the GHC API.

After the environment is initialised, the program is compiled to the desugaring pass. Pre-processing transformations (replacing holes and pattern errors) are applied before the Core program is passed into GHC's simplifier. The parsed source, the simplified Core program and the `IORef` are returned to the `compile` function, which normalises the Core program and retrieves all warnings.

6.2 Defining Similarity between Two Programs

Matching programs can be done on two levels by two different relations defined in a type class `Similar`, using the same notation as in Ask-Elle:

```
class Similar a where
 (~>) :: a -> a -> Bool
 (~=) :: a -> a -> Bool
```

The `~>` relation can be read as "is a predecessor of" and if comparing two Core expressions `s ~> m`, we call `s` the student expression and `m` the model expression. The `~=` relation can sloppily be read as "is syntactically equal to". The `~=` relation is, in reality, an approximate equality relation, but for simplicity, it is referred to as equality. The `Similar` instance for binders is displayed in Figure 6.3

```
instance Similar (Bind Var) where
  (Rec es)     ~> (Rec es')       = es ~> es'
  (NonRec v e) ~> (NonRec v' e')  | isVarMod v, isVarMod v' = True
                                  | otherwise = v ~> v' && e ~> e'
  (NonRec v e) ~> Rec ((v',e'):_) = v ~> v' && e ~> e'
  _ ~> _                          = False

  (Rec es)     ~= (Rec es')      = es ~= es'
  (NonRec v e) ~= (NonRec v' e') | isVarMod v, isVarMod v' = True
                                 | v ~= v' && e ~= e'
  _ ~= _                         = False
```

Figure 6.3: The `Similar` instance for `Bind Var`.

Two programs are matched binder by binder, expression by expression. The list of tuples of variables and expressions for the recursive binders are matched pair by pair, and it is checked that the lengths of the lists are equal. For the non-recursive constructor, an additional check is added; binders containing module information (as the name of the module) are not relevant when matching functions and are thus disregarded by checking if the binder is a "module variable" with the `isVarMod` predicate.

The predecessor relation can also consider two different types of binders as similar. Why? Consider the case where a student started writing a solution using explicit recursion but inserted a hole instead of the recursive call. This function would be bound by a `NonRec`, while the most similar model solution, using explicit recursion, would be bound with the `Rec` constructor. Despite having different binders, the student program can be considered a predecessor to a model solution if the name and expression otherwise match (see the third function binding for the `~>` relation in Figure 6.3).

Expressions are also matched one to one, e.g. a `Lam name expr` expression is similar to another `Lam name' expr'` expression if `name ~= name'` and `expr ~= expr'` (the predecessor relation has the same definition). This rule has some exceptions. The first exception is if an expression from the student solution represents a hole, which is considered a predecessor to any expression. The second exception is made for case expressions to match student programs with missing cases of two kinds. First, consider the following model solution:

```
dupli []     = []
dupli (x:xs) = x:x:dupli xs
```

and that a student submits the following partial solution:

```
dupli _ = []
```

The student attempt is not complete but not yet wrong and should therefore be considered a predecessor of the model program. The wildcard pattern in the function binding of the student attempt matches *all* input, and the corresponding Core

program will not contain any case expressions since the given pattern covers all cases. Hence, we check if the student expression matches any alternative in the case expression in the model program. The predecessor relation for case expressions is defined in Figure 6.4.

```
(Case e v t as) ~> (Case e' v' t' as') = t  ~> t' &&
                                         e ~> e' &&
                                         v ~> v' &&
                                         as ~> as'
e  ~> (Case e' v' t' as') = any ((e ~>) . getAltExp) as
```

Figure 6.4: The predecessor relation for case expressions. An arbitrary expression is a predecessor of a case expression if it is a predecessor of any of the expressions in the case alternatives.

This definition might be too generous. For instance, functions defined with overlapping patterns will only contain their reachable cases when compiled to Core; all overlapping case alternatives are removed by the simplifier. A function with at least one overlapping pattern might wrongly be classified as a predecessor to a model solution (consider the `dupli` example above, any succeeding function binders would be removed). However, overlapping patterns are recognised by GHC and can be retrieved from the warnings from the compilation.

The definition of the predecessor relation for case alternatives is a bit involved (see Figure 6.5). Suppose that a student program has missing cases (e.g. consider that a student would have written `dupli [] = []`). Even though the student only defined one function binding, the corresponding Core program will have a case expression scrutinising the input list since this function binding does not cover all possible inputs. The first case alternative would be compiled into an `Alt (DataCon []) [] (Var [])`, which corresponds to the student's definition. The second case alternative is inferred by GHC to `Alt (DataCon :) [v1,v2] (Var patError)`. Its returning expression is a pattern-error variable since the student did not introduce a function binding with a `(:)` pattern. If a student case alternative contains a pattern error, it is considered a predecessor to the model case alternative if their `DataAlt`s and list of variables in the case alternatives are similar.

```
instance Similar (Alt Var) where
    a@(Alt ac vs e) ~> (Alt ac' vs' e')
        | isPatError e , not (isPatError e') = compareAlt
        | not (isCaseExpr e) , isCaseExpr e'
        , not (e ~> e') = compareAlt && matchWithAlt a e'
        | otherwise = compareAlt && e ~> e'
    where compareAlt = ac ~> ac' && vs ~> vs'
          matchWithAlt (Alt ac _ e) (Case e' _ _ as) =
                any (\(Alt ac' _ e') -> ac ~> ac' && e ~> e') as

    (Alt ac vs e) ~= (Alt ac' vs' e')  =
        ac ~= ac' && vs ~= vs' && e ~= e'
```

Figure 6.5: The `Similar` instance for case alternatives. The guards on pattern errors and nested cases of the expressions in the case alternatives are defined to deal with missing and nested cases, respectively. Syntactic equivalence for case alternatives is straightforward.

If the student expression is *not* a case expression while the model expression is, the student expression is considered a predecessor to the model expression if it matches any of the expressions in the case alternatives of the model expression (as the definition in Figure 6.4 shows). This definition is needed to match student programs with *no* case expressions but is too liberal when we have programs with nested cases. We know that we have encountered a case expression in both the student and model programs when matching case alternatives. Suppose that the expression in the model case alternative is another case expression, and the student expression is not. We cannot simply compare the expressions of the case alternatives against each other (since it will, in turn, match the student expression against any of the expressions in the nested case expression in the model solution, as defined in Figure 6.4). To understand why we need to check for nested cases in the model expression when matching case alternatives, consider the following (real-world) student attempt of the function `mylast :: [a] -> a`:

```
mylast (x:xs) = x
```

We attempt to match this attempt with a model solution defined as:

```
mylast [x]    = x
mylast (_:xs) = mylast xs
```

To understand why we need special handling when matching case alternatives for this example, we can rewrite the model solution to have the same structure as its corresponding Core program:

```
mylast = \ds -> case ds of
    [] -> error "patError"
    (x:xs) -> case xs of
        [] -> x
        (y:ys) -> mylast xs
```

When matching the cons-case alternative in the student program with the first cons-alternative in the (restructured) model solution, both programs have the same `DataAlt` (a `DataCon (:)`) and list of pattern variables. If not including the extra check for nested case expressions for the model expression (if we would match the alternatives as defined in the fall-through case in Figure 6.5), we would try to perform the following match:

```
(Var x) ~> Case (Var xs) wild (TyVarTy a)
                    [DataAlt ([]) [] (Var x),
                     DataAlt (:) [y,ys] (App (Var (mylast)) xs)]
```

The definition of the predecessor relation for case expressions (Figure 6.4) will check if the expression `Var x` matches *any* of the expressions in the list of case alternatives. `Var x` matches the expression of the case alternative for the empty list, and the student program would falsely be accepted as a predecessor to the model solution in the discussed example. When this situation arises, i.e. the student expression is not a case expression, and the model expression is a case expression, case alternatives are instead matched with the `matchWithAlt` function (see Figure 6.5). It considers a case alternative to be a predecessor to a nested case alternative if both the `DataAlt` and the expression of the case alternative match. With this additional check, we avoid falsely matching the expression of the cons-alternative with the expression of the empty list alternative in the example above. The list of variables in the case alternative is disregarded due to possible name offsetting in the nested case.

The third example where the predecessor relation differs from the equality relation is when matching letrecs. For instance, we want to match a direct recursive function with a similar function using a letrec. This is done by matching any expression against a recursive `Let` expression, and if this expression matches any of the expressions bound by the `Let` binder, it should be viewed as a predecessor of the whole expression. The predecessor relation for `Let` expressions is given below:

```
(Let b e)     ~> (Let b' e')              = b  ~> b' && e ~> e'
e             ~> (Let (Rec es) ine)       = any ((e ~>) . snd) es
```

A final exception to the rule of matching construct against construct is for applications, which might be an application of a binary function. A binary function might be commutative; thus, we must check for commutativity to not disregard student solutions like `x == y` if the model solutions are defined with `y == x`. Since functions and operators simply are variables in Core, there is no elegant way of testing that a function is commutative. Checking the commutativity of a function is a hardcoded comparison of the function to be applied and some standard commutative operators (this is done similarly in the current version of Ask-Elle). If an application of a binary function is commutative, the expressions are considered similar regardless of the order of application of its argument. The predecessor relation and the equality relation are defined in the same way for the application of binary functions; the definition for the syntactic equality relation is given below:

```
(App (App f e) a) ~= (App (App f' e') a')
                        | isCommutative f
                        , f ~= f'  = (e ~= e' && a ~= a')
                                   || e ~= a' && e' ~= a
```

Moreover, literals are checked to be equal and to have the same type. The similarity of variables is checked by comparing their *occurrence name* since uniques are not stable across compilations, and thus comparing variables with == (which considers the uniques) would fail. Two variables are also deemed similar by the predecessor relation if the student variable is a hole variable, while this is not true for the equality relation.

6.3 Generating Feedback

Feedback regarding whether an exercise is completed or the submitted attempt is on track towards a correct solution is derived from matching the normalised Core programs against each other. Additional suggestions are derived from GHC warning messages, the hole-matching procedure and the missing base-case analysis described in the following subsections. All this information is gathered from the `CompInfo` returned from the compile function and mapped into a customised feedback data type called `Feedback`, which is defined as:

```
data Feedback = Complete [Suggestion]
              | Ontrack [Suggestion]
              | Unknown [Suggestion]

data Suggestion = IncompletePat String
                | OverlappingPat String
                | HoleSuggestions String
                | HoleMatches String
                | HLint String
                | Error String
                | General String
```

`Complete` is only returned if the student program is syntactically equal (~=) to one of the model solutions. It might contain additional `Suggestion`s, such as HLint suggestions on refactoring the program. `Ontrack` is returned when: the student program is a predecessor (~>) to at least one of the model solutions, no overlapping pattern warnings were obtained from the compilation, and there are no warnings other than the ones present in a model solution. Overlapping pattern warnings must be checked explicitly since overlapping case alternatives are removed by GHC, and a program with overlapping patterns could falsely be interpreted as "on track". Model solutions may contain warnings; for example, partial functions (like `head`) would give rise to an incomplete pattern warning but still be correct according to the specification of the exercise. The warnings obtained from compiling the student program are therefore compared with the warnings from the model solutions to avoid giving unwanted incomplete pattern warnings. The `Ontrack` constructor might also contain

additional suggestions, like `HoleMatches` or other warning messages. The `Unknown` constructor is used for all programs that are neither a match nor a predecessor to a model solution. It can hold additional suggestions for all detected issues.

The `IncompletePat`, `OverlappingPat` and `HLint` constructors of the `Suggestion` type are self-explanatory and hold the warning message from GHC as a string. `HoleSuggestions` contains the GHC error message giving suggestions on terms or functions that can be used in place of the hole. Hole-fit suggestions from GHC's typed holes mechanism should be analysed and processed before being given to the student since they might be too general or not beginner-friendly. We have not focused on this and leave it as future work. `HoleMatches` contains the exact-printed term corresponding to a hole, which is explained in subsection 6.3.2. Finally, `General` is used for pointing out missing base cases or other general feedback, and `Error` is used for type errors or other compilation errors.

6.3.1 Detecting Missing Base Cases

GHC fires the incomplete pattern warning if the patterns are not exhaustive; in other words, the patterns do not cover all possible input values. However, students sometimes have "incomplete patterns" in the sense that a function would be infinitely recursive without them. Consider the following definition of `dropevery` (taken from a set of real-world student attempts), a function that drops every n:th element from a list:

```
dropevery xs n = take (n-1) xs ++ dropevery (drop n xs) n
```

It does not have incomplete patterns since the general patterns `xs` and `n` in the function binding match all possible inputs. On the other hand, this program would not terminate since there is no stop condition for the recursion. It is easy to analyse if a recursive function lacks a base case; if a recursive binder does not contain a case expression, there may be a missing base case. This check does not guarantee to find all missing base cases since the program might contain a case expression that is not a stop condition for the recursion. Nevertheless, it can catch the most basic cases.

Haskell is a lazy language, and there are special cases where infinite recursion is useful. For example, the Prelude function `repeat` is such an example. Yet, for beginner exercises, this is probably not the case. The given feedback can also be purposely vague, suggesting that there *might* be a missing base case.

6.3.2 Hole-Matching

We would like to relate holes in a student program to a corresponding term in a model solution. When a hole matches a term, we can use the term for generating feedback. The first step is to find all the Core expressions that match the holes in the student solution; all matching expressions found by the hole-matching procedure are returned in a list.

From the list of Core expressions from the model solution that match holes in the student program, if the outermost expression is a `Tick`, we can use its source location

to extract the corresponding parsed expression from the concrete AST (HsSyn) as described in section 4.4. Consider one of the solutions to the dupli exercise:

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

and that the student attempts

```
dupli xs = concatMap _ xs
```

The hole-matching function will return the following Core expression:

```
[Tick src<studentfiles/Temp.hs:4:19-31> (App (Var replicate)
(Tick src<studentfiles/Temp.hs:4:30> (App (Var I#) (Lit 2))))]
```

The outermost expression is a `Tick`, which contains a source span (the source location for this expression). The source span can first be retrieved from the `Tick`, and then the corresponding parsed expression can be extracted from the parsed source (the AST returned by the parser). The parsed expression is obtained by traversing the whole parsed source for the model program and returning the expression at the given location. If the location retrieved from the `Tick` directly matches an expression in the parsed source, it can be retrieved with the `getHsExprFromLoc` in Figure 6.6.

```
getHsExprFromLoc :: RealSrcSpan ->
                    ParsedSource ->
                    Maybe (LHsExpr GhcPs)
getHsExprFromLoc rss ps@(L l hsm)
            | nonEmpty locExps = return (head locExps)
            | otherwise        = Nothing
where exps     = universeBi ps :: [LHsExpr GhcPs]
      locExps  = catMaybes $ map (matchRealSpan rss) exps
```

Figure 6.6: A function for retrieving parsed expression on a given location in the source file. The `matchRealSpan` returns an expression if the given source span matches the location of the parsed expression.

Using the function in Figure 6.6 on the example above, the following parsed expression is returned:

```
L EpAnnNotUsed (HsPar EpAnn AnnParen {ap_adornment = AnnParens,
ap_open = EpaSpan SrcSpanOneLine "./studentfiles/Temp.hs" 4 19 20,
ap_close = EpaSpan SrcSpanOneLine "./studentfiles/Temp.hs" 4 31 32}
(L EpAnnNotUsed (HsApp EpAnn NoEpAnns (L EpAnnNotUsed (HsVar
NoExtField (L  replicate))) (L EpAnnNotUsed
(HsOverLit EpAnn NoEpAnns 2)))))
```

The obtained parsed expressions can then be printed as originally defined with the function `exactPrint`. Passing the parsed expression above to `exactPrint` returns the string `(replicate 2)`. In this case, the exact-printed string from the model solution could be returned directly. If, instead, the hole-match expression includes a

user (or teacher) defined variable, it must be translated to the corresponding variable in the student solution for the feedback to make sense. For example, consider the model solution and following student solution to an exercise on a function that, given a list and an integer index, returns the element at the given index (starting from 1):

```
elementat :: [a] -> Int -> a
elementat list n = list !! (n-1)
```

```
elementat :: [a] -> Int -> a
elementat xs i = _ !! _
```

The hole-matching procedure will return a list of two matching expressions, one for each hole in the example above. After converting the hole-matching expressions to strings with the exact-printer, the list `["list","(n-1)"]` is obtained. Displaying these hole-fits to the student would not make sense since the student used other variable names in their solution. To give feedback with the same names the student used, the maps containing the renamed and original variables of the student and model solution, respectively, are combined into a single map of variable names (as defined before renaming) with the model variable names as keys and the student's variable names as values. For each item in the list of hole-fits, the corresponding student variable can be looked up in the map, and `xs` and `(i-1)` can be used in the feedback to the student instead.

If the hole matches a locally defined helper function in a model solution, the definition of the local function must also be retrieved from the parsed source. The function definition is then added to the feedback along with the hole-fit and can be displayed to the student.

The first expression at the right-hand side of a function binding is often wrapped in a `Tick` with a source span spanning the whole function definition. Hence, it is not possible to search for a corresponding parsed expression directly since this source span corresponds to a declaration. When matching a hole corresponding to the whole model side of a function binding, the parsed expression is retrieved by first extracting the complete function binding and then extracting the model side expression of the declaration. Hence, we use the `showFromLoc` function displayed in Figure 6.7. If it finds a matching parsed expression directly, its string representation is returned along with potential additional definitions (if it is a locally defined function). If no expression can be found at the given source location, we check if we can retrieve the whole right-hand side for the function binding and once again check for additional local declarations of the retrieved expression.

```
showFromLoc :: ParsedSource -> RealSrcSpan -> (String,[String])
showFromLoc ps@(L l hsm) rss = case getHsExprFromLoc rss ps of
   Just exp -> (showParsedExprexp,
            map showPsBinds (getLocalDeclarations rss ps [exp]))
   Nothing   -> case getHsRhsFromLoc rss ps of
       Just d -> (showParsedExpr d,
             map showPsBinds (getLocalDeclarations rss ps [d]))
       Nothing  -> ("",[])
```

Figure 6.7: The main entry point for retrieving the string representation of a hole-fit. If the hole-fit corresponds to a local function, the definition of the function is also returned as a separate list.

7

Evaluation

The primary goal of this thesis is to improve the number of recognised student attempts, and therefore the evaluation focuses on recognition. First, the configuration of the new normalisation procedure is evaluated, i.e. how matching programs work when normalising programs using Core compared to Ask-Elle's current approach. We will also evaluate if adding GHC's simplifier and the new transformations gives any improvements. Second, the recognition of programs and correctness of the additional feedback is considered. For example, it is checked if programs previously classified as having missing cases now receive feedback on incomplete patterns. We also discuss and analyse which programs that cannot be recognised.

Furthermore, the execution time of the new normalisation and the diagnosing procedure is measured and compared to the execution time of the current diagnosis in Ask-Elle. Finally, the correctness of the added Core-to-Core transformations is evaluated with the Core Linter. The Core Linter is also run on programs that include other constructs than functions to assess that the new transformations are not overly customised for Ask-Elle's current exercises.

7.1 Evaluation data

The new normalisation procedure and the Core-to-Core transformations are benchmarked by compiling student programs and model solutions into Core, applying the transformations and matching the programs against each other. The set of student programs includes both real-world student attempts and artificial ones designed to capture particular Haskell constructs that are expected to be normalised in a certain way. Testing on both contrived examples and real-world student programs is done to avoid bias both in the chosen test cases and from which exercises the students were trying to solve in the particular experiment. In addition, we want to get real-world examples of mistakes students commonly make.

The student data is retrieved from two databases of logged student attempts from two experiments conducted in 2015 and 2023, respectively. Gerdes et al. [12] has collected the data from 2015, and the 2023 data was gathered in a master thesis

by Maciej Goszczycki. In the experiments, students were asked to solve exercises in Ask-Elle while their interaction was logged. The first database from 2015 consists of 889 student attempts logged from Ask-Elle, and the second one of 448 attempts.

An attempt is a program or fragment of a program that was logged when a student actively requested help from the tutor. Both data sets, but especially the one from 2015, included many duplicate attempts, attempts that did not compile, or had other errors. There is no obvious explanation for the large discrepancy in the share of erroneous programs in the two data sets. One explanation could be the experience level of the participants or differences in the instructions they were given. Before testing, all erroneous programs (including compiler errors, type errors and implementation errors) and duplicate programs[1] were removed. After all such attempts were removed, 197 and 113, respectively, student attempts remained.

All attempts were re-categorised as *Complete, OnTrack, TestPassed, Missing, Unknown* based on the logged feedback from the experiments. Complete programs are programs without holes that Ask-Elle successfully matched with a model solution. Programs categorised as OnTrack are given feedback that the attempt is Correct but not yet complete or that the attempt is similar but might have a missing parenthesis. Programs labelled as TestPassed is all attempts that could not be matched with a model solution but passed all QuickCheck test. The Missing category includes programs with missing cases, such as functions not defined on all possible inputs. Finally, the Unknown programs did not fit into any of the above categories but had no errors.

7.2 Compilation Configuration and Added Transformations

Since the simplifier is not completely configurable, we must evaluate whether including the simplifier as part of the normalisation would improve recognition. We also want to assess whether the new Core-to-Core transformations contribute to the recognition rate. The different results obtained from setting specific options in GHC are not displayed here, but the flag configuration of GHC can be found in Appendix A.2. The new transformations cancel out some transformations from the simplifier. For example, the simplifier sometimes floats out constants as top-level binders; this transformation would probably cause student programs to have a different number of binders than any of the model solutions, but this is reversed by the manual inlining pass.

Matching student attempts with the model solutions was repeated for different configurations of the compilation, where the only success criterion was if the programs could be matched, disregarding additional feedback. Before compiling the student attempts and the model solutions, all declared RULES pragmas from section 5.2 and the type signature were inserted into the source files, if not already declared.

[1] Duplicates were only checked by comparing the input strings, and hence some duplicates may remain depending on variable names and if parentheses were used.

The student attempts were then matched against the set of model solutions for that exercise, and a match was considered successful if the student program either was a predecessor to or syntactically equivalent to a model solution. The tests were conducted for both the 57 contrived student examples and the 310 real-world student attempts. The programs were compiled to the desugaring (Desugar) and simplifier (Simpl) pass, with the additional renaming transformation (marked by *) or the complete set of manual transformations (marked by **), respectively. Complete programs could, to a very high degree, be recognised already by compiling the programs to Core and applying the renaming transformation. The new normalisation procedure improves recognition of incomplete (OnTrack) programs when applying the simplifier and the additional transformations, albeit more significantly so for the contrived examples. The results for the artificial student examples are shown in Table 7.1, and the results for the real-world student data are given in Table 7.2 and Table 7.3, respectively, where the original feedback-based categorisation from Ask-Elle is given as well.

Table 7.1: The number of successful matchings of the artificial student attempts with a particular model solution, compiled to Core with different compilation and normalisation configuration.

Expectation	Desugar*	Simpl*	Desugar**	Simpl**	Total
Match/Ontrack	22 (46%)	25 (52%)	30 (63%)	43 (90%)	48
No match	9	9	9	9	9

Table 7.2: Results of matching logged student attempts from the 2015 data set with the model solutions.

Category	Desugar*	Simpl*	Desugar**	Simpl**	Total
Complete	36 (99%)	36 (99%)	37 (100%)	37 (100%)	37
OnTrack	54 (79%)	58[1](85%)	62 (91%)	68 (100%)	68
TestPassed	0 (0%)	1 (0%)	2 (1%)	1 (0%)	33
Missing	6 (14%)	6 (14%)	6 (14%)	7 (16%)	43
Unknown	4 (25%)	4 (25%)	7 (44%)	7 (44%)	16

Table 7.3: Results of matching logged student attempts from the 2023 data set with the model solutions. Two programs originally classified as Complete in was disregarded in this evaluation since they were given wrong type signatures. The categories are derived from the feedback given from Ask-Elle in the logged student attempts.

Category	Desugar*	Simpl*	Desugar**	Simpl**	Total
Complete	19 (100%)	19 (100%)	19 (100%)	19 (100%)	19
OnTrack	18 (72%)	21[1] (84%)	19 (76%)	25 (100%)	25
TestPassed	0 (0%)	0 (0%)	1 (0%)	4 (0%)	52
Missing	0 (0%)	0 (0%)	0 (0%)	0 (0%)	8
Unknown	2 (14%)	2 (29%)	2 (29%)	2 (29%)	7

Since the artificial student programs in Table 7.1 were designed to capture specific problems, the results showed more significant differences between applying the minimal set of transformations and all manual transformations. These results highlight that manual transformations play an important role in normalising Haskell programs even though almost all the Complete programs from the 2015 data set and all Complete programs from the 2023 data set could be matched by comparing programs compiled to Core without further normalisation. It is not surprising that programs already classified as Complete could be recognised by compiling to Core without additional transformations. When an attempt is a finished solution, the student has likely followed a strategy proposed by Ask-Elle, and there is a high chance that a program categorised as Complete is very similar to a model solution.

In the following section, the results of matching feedback, and not only the similarity, is tested with the compilation configuration that performed best, namely including GHC's simplifier and applying all additional transformations.

7.3 Evaluating Feedback

In contrast to the previous section, the evaluation of recognition and correctness of the given feedback is now based on more criteria:

Complete All student programs categorised as Complete by Ask-Elle, should be syntactically equal to a model solution (matching with `~=`).

OnTrack All student programs categorised as OnTrack by Ask-Elle should be a predecessor to at least one of the model solutions (matching with `~>`).

Missing Cases Student programs that were categorised as Unknown but given feedback that they had missing cases should receive feedback that they have incomplete patterns (retrieved from GHC's warnings).

TestPassed QuickCheck tests were not run in this evaluation since the QuickCheck tests are run on the unmodified source code and are not affected by normalisation. Hence, these programs are assumed to still be in this category if not found to match or be on track. For this category, the only interesting result is whether more programs can be given more interesting feedback than whether the tests passed.

HLint Programs that received feedback from HLint that includes a suggestion of a correct model solution but where the current attempt is not recognised.

Type Signature Error Attempts where the type signature does not match the specified type signature, and feedback is given accordingly.

Note that for the new classification, programs that were previously categorised as, for example, Missing, will be in the Complete category if it could be matched with the new normalisation procedure. The results are shown in Table 7.4.

[1] The result for OnTrack programs in the Simpl* column is misleading since GHC's simplifier might throw away code that appears after a hole when holes are not replaced.

Table 7.4: Comparison of the categorisation of data with the old and the new approach. The categories HLint and TypeSig are specific to the new approach.

	Experiment classification			New classification		
Category	2015	2023	Tot	2015	2023	Tot
Complete	37	21*	58	40	23	63
OnTrack	68	25	92	81	27	107
Missing	43	8	51	32	8	40
TestPassed	33	52	85	30	47	77
Unknown	16	7	23	12	5	17
HLint	-	-	-	2	1	3
TypeSig	-	-	-	-	2	2
All	197	113	310	197	113	310

*Two programs had wrong type signatures.

The number of programs recognised as Complete increased from 58 to 63. Though, two of the programs that were classified as Complete in the experiments had wrong type signatures. Since Ask-Elle previously removed type signatures, these two programs were accepted as Complete, although they should not have been. For example, an attempt on the length function was typed `mylength :: [a] -> int` instead of `mylength :: [a] -> Int` where non-capitalised `int` is a type variable. Hence we can consider the number of recognised Complete programs to have increased from 56 to 63, or 12,5%. Without the faulty type signatures, these programs would have matched a model solution also with the new approach but are counted in the TypeSig category. The number of programs categorised as OnTrack increased from 92 to 107, an increase of about 16%. In total, the number of recognised programs increased from these two data sets increased from 48,7% to 55.2%, which corresponds to about 13%.

Of the 51 programs for which Ask-Elle detected that the student programs were not defined on all input, eleven, or about 20% programs, could now be recognised. The remaining 40 programs correctly got feedback that the solution had incomplete patterns. The attempts that now *could* be recognised are due to exercises on partial functions, which are purposely not defined on all possible inputs and were previously inaccurately categorised as having missing cases. Six programs, or about 38%, of the Unknown programs, could now be matched. Finally, only five programs in the TestPassed category could be recognised as similar to any of the model solutions. Three additional programs got a suggestion for a correct model solution from HLint.

At first glance, the results for the programs in the TestPassed and Unknown categories are a bit disappointing. The majority of these programs could still not be matched with the new normalisation procedure, and these results are far from those mentioned in subsection 1.1.1, that the number of recognised programs could be increased from 56% to 81% as suggested in [12]. However, these numbers are not directly comparable to the results displayed in Table 7.4: the set of student attempts is different, the calculations are done in different ways, and the suggested increase in recognition in [12] also includes *adding more model solutions*. These model solutions

– that would contribute to the suggested increase in recognition – are not explicitly stated, and it is unknown whether they are added to Ask-Elle since [12] was published in 2017. In the case that these model solutions have been added, they would already account for a higher recognition rate in the current version of Ask-Elle.

We analysed the 94 unrecognised Testpassed and Unknown programs to discover why they could not be recognised; these can be grouped as follows:

Using prelude corresponding function: Six attempts used the corresponding prelude function of the exercise they were trying to implement (e.g. `myreverse = reverse, mylength = length`). This fails since no model solution uses this approach, and the definition of the prelude functions is not inlined.

Using other prelude functions: Eleven programs failed due to using other prelude functions not used in any of the model solutions.

Point-free: Three attempts failed where the model solution used point-free style and the student used normal application and parenthesis or the application operator `$`.

Redundant patterns: 20 attempts could not be matched due to redundant patterns in the function bindings. Another four attempts could not be matched due to redundant "patterns" on the expression level, e.g. returning `[[]]` instead of `[]` in the base case for a function with type `[a] -> [[a]]`.

Type mismatch: A handful of attempts failed to minor differences originating from the types of expressions in the program inferred by GHC. This could, in several cases, be resolved by a slightly changed model solution (this is discussed further in section 8.2).

Subtle differences: A few attempts failed to match due to subtle differences. For example, one attempt failed by putting an infix operator in parenthesis, which makes it a prefix operator that does not match the intended structure of the attempt: `palindrome xs = _ (&&) _`.

Error in tests: A few attempts, that at first looked like they had very subtle differences from the model solution, rather revealed an error in the test properties. For instance, consider the model solution

```
compress []         =  []
compress [x]        =  [x]
compress (x:y:ys)   =  if x==y
                       then compress (x:ys)
                       else x:compress (y:ys)
```

and compare it with the student's attempt:

```
compress []         =  []
compress [x]        =  [x]
compress (x:y:xs)   | x==y       = compress (x:xs)
                    | otherwise  = x:y:compress xs
```

Guards with equality checks and if expressions are equal in Core; hence the only difference is that the student prepends both x and y before the recursive call to compress. However, the semantics of these two programs are different and rather suggest a mistake in the test property. This is another problem, and the above attempt, and similar ones, are correctly not recognised as complete.

Complicated or other solutions: The remaining programs could not be recognised due to over-complicated solutions or different approaches to the model solutions.

As the above evaluation shows, we do not want to recognise all of these programs as correct solutions. For example, programs classified as TestPassed due to errors in the test property should, of course, not be recognised as a correct solution. Programs with redundant patterns should similarly not be accepted as a correct solution but not fall into the Unknown category either, which only gives general feedback. It would be much better to detect redundant patterns and give feedback accordingly, as discussed in section 3.3.

We could also see several attempts that implemented an explicit recursive solution to the `fromBin` exercise, which failed since no explicit recursive model solution currently exists. Such a model solution probably *should* be added, but it is up to the teacher to decide what should be considered a correct solution to an exercise. Another recurring issue was for the `foldl` implementation of `fromBin` exercise where the model solution folds with a locally defined function. Due to how GHC infers types, (correct) student attempts folding with a directly inlined lambda cannot be recognised. For a more extensive discussion on this issue, see section 8.2.

7.4 Execution Time

Not only is it important that a tutor system gives understandable and correct feedback it must also be provided within a reasonable amount of time. Enumerating all possible intermediate model solutions and searching for a matching one, as currently done in Ask-Elle (as explained in subsection 2.2.2), takes time. Our initial hypothesis was that hole-matching, as described in subsection 6.3.2, would be much faster.

Simple benchmark tests were defined with the Criterion[2] library to measure the execution time for compiling programs and generating feedback, both for the strategy-based approach and the hole-matching approach. We run the complete diagnosis, including other types of diagnosis, and any differences can not entirely be attributed to the hole-matching process. Criterion runs each benchmark a number of times and creates statistics for the measured running time. One benchmark corresponds to diagnosing *all* attempts read from a particular JSON file. The benchmark tests were also run a few times to ensure reliable measurements. The exact measured execution time is not the main focus since it could be influenced by the environment in which tests are run. The tests were run on the same machine inside the Windows

[2]https://hackage.haskell.org/package/criterion

Subsystem for Linux (WSL 2) environment. What is interesting is rather the comparison between the old approach and the new approach. The results show that the new diagnosing approach is faster than the current approach in Ask-Elle and are displayed in Table 7.5 and Table 7.6, respectively.

Table 7.5: Mean execution time and standard deviation for compiling and generating feedback for all recognised attempts (Complete and OnTrack) from a JSON file in batch, with Ask-Elles deep-diagnosing without QuickCheck testing.

Data	Number of attempts	Average time	Std dev
Complete (2015)	37	41.80s	76.21ms
Complete (2023)	21	24.19s	40.29ms
OnTrack (2015)	68	73.29s	517.7ms
OnTrack (2023)	25	27.88s	74.77ms

Table 7.6: Mean execution time and standard deviation for compiling and generating feedback for all recognised attempts (Complete and OnTrack) from a JSON file in batch, with the new approach for recognition and creating feedback.

Data	Number of attempts	Average time	Std dev
Complete (2015)	37	17.07s	299.3ms
Complete (2023)	21	10.79s	76.21ms
OnTrack (2015)	68	55.57s	790ms
OnTrack (2023)	25	20.03s	342.2ms

With the current diagnosis procedure in Ask-Elle, diagnosing one student attempt is measured to about 1.1 seconds. In contrast, with the GHC approach, diagnosing a student program is done in less than half a second for finished programs and about 0.8 seconds for programs on track. The time difference between diagnosing OnTrack and Complete programs by the new approach is likely because additional suggestions for Complete programs only are retrieved from HLint warnings, and no hole-matching or further analysis is performed.

It is notable that the new approach is faster, even though it has not been optimised, whereas Ask-Elle has been. Supporting larger exercises is likely to increase this difference.

7.5 Correctness of New Transformations

The Core Linter has been a valuable tool for catching mistakes early when implementing new transformations. Around 300 Haskell programs have been run through the Core linter after applying all transformations (before removing the types from the AST) without giving any errors. These are all small programs containing one to two functions and aim to implement a solution to one of a small set of exercises which gives even less diversity to the tested programs. Preferably, other types of Haskell programs should be tested as well.

Testing the new approach with Haskell source files from the wild is difficult since most will probably use external libraries. Including external libraries when compiling with the GHC API requires including the external packages in the compilation configuration (in the source code), which is hard to automate. However, larger files containing other Haskell constructs, like data type definitions, were tested on a small scale. The test suite included a few smaller test files with data type definitions, like a data type for natural numbers and a small expression data type for arithmetic with a complementing eval function. It also included larger files, for example, the first two assignments in the course *Functional Programming* at Chalmers. The second and larger of the two latter programs is a blackjack game with about 200 lines of code. All tests in the added test suite could be checked with the Core Linter without errors.

The results would have been much stronger if the correctness of the transformations could have been proved. Proving the correctness of the transformations would have required a substantial amount of additional work compared to settling for testing. The transformations in GHC are not proven correct either; instead, they are tested with a large number of test cases.

8

Discussion

Many transformations we want to use for normalisation are already implemented in GHC, but GHC has different reasons for applying them and does not always apply them when we would like to. Another problem is that we cannot disable single transformations. For example, the transformation replacing case binders with its scrutinee *reverses* the same transformation just applied by GHC instead of disabling the transformation in the first place. Another (in our case) problematic transformation is GHC's dead code elimination. It might leave Ask-Elle unable to provide feedback for incomplete parts of a solution, for example, if a student starts defining a local helper function that is not yet called anywhere.

Using GHC's abstract syntax for the internal representation of programs introduces a strong dependency on GHC. This dependency might require massive refactorings if we want to upgrade GHC to a newer version or use another compiler. We picked the latest recommended version (GHC 9.2.5) and hope that upgrading GHC or using another compiler won't be necessary for the foreseeable future.

8.1 Working with the Core AST

Using GHC's Core AST, as opposed to earlier intermediate representations in the GHC pipeline, has the benefit of abstracting away from many high-level constructs and, of course, has fewer options to consider due to the small number of data constructors in the expression type. The downside is that the available Core data types represent many different Haskell constructs, and we must resort to string comparisons and other less type-safe options to check what kind of terms we are working with. For example, typed holes are compiled into case expressions where the scrutinee contains a "typeError" variable. To recognise that a case expression originates from a typed hole, we must check that the scrutinee contains a variable named "typeError". Working on a higher level of abstraction is likely more type-safe but would require more work on implementing the transformations used for normalisation.

8.2 Type Signatures in Exercises

GHC's type inference rules can cause subtle differences in the AST, which makes syntactical matching fail. For instance, the following two programs are not syntactically equal in Core:

```
isZero :: Int -> Bool
isZero 0 = True
isZero _ = False
```

```
isZero :: Int -> Bool
isZero x = x == 0
```

The function binding in the first definition will be compiled into a case expression scrutinising the primitive data constructor `I#`, and the second one compiles to a Core program where directly applying the equality operator on the input argument and the literal 0. Using the most general type,

```
isZero :: (Eq t, Num t) => t -> Bool
```

or using `Integer` instead of `Int` would give syntactically equivalent programs. Even though both `Integer` and `Int` are in the `Eq`, and `Num` type class, using `Integer` (or `Num t`) compiles to a program directly applying the equality check on the input argument and the literal 0 using == for both definitions of the `isZero` function above.

More generally, pattern matching on data constructors will not be normalised to the same program as one using guards and equality checks. However, it would only be possible to perform "pattern matching" by guards and equality checks in special cases; for nullary data constructors or data constructors with other nullary data types or primitive types as arguments. This is probably not a big issue in practice.

Some exercises are not defined with the most general type inferred by GHC; hence, all programs (model solutions and student attempts) for such exercises should have an explicit type signature to match the one specified in the exercise. To not force the student to write the specified type signature explicitly, we first check if the input program has the specified type signature. If a student program lacks a type signature, we insert the type signature, which is extracted from a model solution. When specifying new exercises, it is good to be aware that the choice of type signature for the main function in an exercise can affect which programs can be matched and which cannot, as exemplified above.

Specifying type signatures is good coding practice and can, in some cases, help GHC infer the type of a hole in a partial program or the type of a `Foldable` when using higher-order functions. Another solution to resolve such type-ambiguities is to extend GHC's type defaulting rules. The standard type defaulting is used to resolve ambiguities that may arise when using type classes for numeric types. For example, the default type of the `Num` type class is an `Integer`. We have extended the default rules with the `ExtendDefaultRules` language pragma to also apply for non-numeric typeclasses. Without including a type signature and without extending the default

rules, the following program would not compile:

```
fromBin = foldl op 0
    where n `op` b = 2*n + b
```

GHC cannot infer what type the input to this function should have. Extending the default rules, GHC can infer the type to be `fromBin :: [Integer] -> Integer`. Using extended default rules will probably allow omitting type signatures for the existing exercises, which in turn will improve recognition of student programs in some cases (like the `isZero` example above). Another possibility is to disable the `MonomorphismRestriction`[1]. The definition of `fromBin` is then compilable by GHC and the type is inferred to `fromBin :: (Foldable t, Num a) => t a -> a`.

Whether to explicitly type Ask-Elle exercises is a design choice, but this choice gives different implications. As we have seen, exercises defined with less general types did affect which student programs could be recognised. On the other hand, defining model solutions without type signatures and still allowing students to give a type signature explicitly might cause the need to rename type variables. Typing exercises should be investigated further to find a solution that possibly both enhances recognition and is intuitive from a pedagogical point of view.

8.3 Testing

Testing correctness should, in retrospect, have been done much more thoroughly and at an earlier stage. The current testing approach mainly consists of manual testing and relying on the Core Linter for sanity-checking and type checking. However, more effort should have been devoted to continuing compilation after applying the new transformations - up to removing types - to ensure that semantics is preserved by evaluating the normalised programs.

8.4 Using the GHC API

The main difficulties of using the GHC API are finding the correct functions in the vast code base and understanding the, sometimes sparse, documentation. A piece of advice to anyone wanting to work with the GHC API, but does not know where to start: search GHC's code repository for things you expect to find! The names of functions, modules, data types etc, are most often reasonable and what you would expect. I found the easiest way of getting acquainted with the code base was to browse the GHC source code from the package documentation, and follow links in the source code to inspect data types and functions. It is especially useful due to the extensive use of type synonyms and type families. These might require further investigation to find the actual type of a function or composite data type you are looking for. The notes and comments added directly in the source code can often provide more understanding of certain concepts than the wiki pages do. So

[1] https://wiki.haskell.org/Monomorphism$_r estriction$

navigating through the source code and reading the comments might be a good way of improving understanding of how certain things work.

Another good starting point for working with Core is the informal description found in [20] and a series of blogposts [27], even if both are slightly outdated. Another resource that gives some insight into how the simplifier operates in general and how GHC's inliner works in particular is [17] by Peyton-Jones. A more general description of the Core-to-Core transformations in GHC can be found in [28]. Altough this paper is a bit outdated, the main ideas are still valid.

An additional problem with using the GHC API is that not all functions and data types are exported, which can sometimes be inconvenient. A simple thing like inspecting the Core AST requires writing `Show` instances for all Core data types (and their underlying data types).

8.5 Future Work

There is room for improvement and future work on the new normalisation and diagnosing approach. Some have already been mentioned, like using hole-fit suggestions from typed hole error messages to do program synthesis, or proving the correctness of new transformations. The recognition of student programs could also be improved if η-conversion could be made reliable for programs with holes.

8.5.1 η-conversion and Holes

Representing holes in Ask-Elle with GHC's typed holes has many benefits but also substantial drawbacks. Holes are a fundamental concept in Ask-Elle that allows students to be guided step by step towards a solution, and it is therefore important that normalisation works for programs with holes. Unfortunately, η-conversion works poorly in both directions if a program contains holes. Since we perform η-reduction instead of η-expansion, Ask-Elle can run into trouble when trying to match a partial student solution that never uses the patterns in the function binding on the right-hand side. For example, consider a partial solution to the familiar `dupli` exercise:

```
dupli :: [a] -> [a]
dupli xs = concatMap (replicate 2) _
```

Since `xs` is never used on the right-hand side in the definition, this cannot be reduced to `dupli = concatMap (replicate 2)` (the model solution). However, `xs` is detected as a valid hole-fit by the typed holes mechanism. Hence, if the hole-fit suggestions for typed holes are used in the future, the hint that `xs` could replace the hole could come from program synthesis instead. Alternatively, η-expansion could always be performed on model solutions (which do not have holes and thus should have correct arity information) by borrowing some functionality from GHC. Then, the student attempt could be matched with the model solution in both η-reduced and non-reduced forms.

8.5.2 Detecting Redundant Patterns

As discussed in section 3.3, and as seen by the evaluation in section 7.3, it would be valuable with a similar analysis for redundant patterns as done for missing base cases. Unfortunately, GHC does not detect redundant patterns and finding redundant patterns turned out to be much more complex than finding missing ones. Since it would be complicated and time-consuming, we wanted to avoid evaluating each case expression and *test* if different case alternatives give the same result for the same input. Instead, the approach was to compare all case expressions in a student program against the case expressions of a model solution.

The set of model solutions often implements different approaches, and it must first be decided which model solution the student solution should be compared with. This could be done by calculating the difference between the two programs and giving penalties for every mismatching expression. However, longer programs could be given a larger difference than short programs, even if the short program implements an entirely different approach. Additionally, a student program might have more case expressions than a corresponding model solution, without, for that matter having redundant patterns. A difference function for Core programs and an analysis of redundant patterns were implemented, but due to its experimental nature, this is left as future work.

8.5.3 Matching Other Types of Exercises

Using Core as the intermediate representation of exercises and solutions in Ask-Elle opens up the opportunity to define exercises on constructs other than functions, for example, data types and typeclasses. In section 7.5, we saw that running the new normalisation on programs that includes other Haskell constructs than functions successfully passed the Core Linter. This is a crucial first step to include other types of Haskell constructs in the exercises of Ask-Elle. However, the current similarity relations and normalising transformations are mainly implemented with Ask-Elle's current small exercises implementing one or two functions in mind. How this scales to any type of Haskell exercise must be further investigated. For instance, consider an exercise on defining a data type for the natural numbers, which could be defined as:

```
data Nat = Zero | Succ Nat
```

Compiling this definition in an otherwise empty module with the current GHC settings and normalisation, results in the following Core program:

```
[NonRec $tc'Succ (App (App (App (App (App (App (Var TyCon)
(Lit 5780159517751058955)) (Lit 2759501021308746894))
(Var $trModule)) (Var $tc'Succ)) (Lit 0)) (Var $krep)),
NonRec $tc'Succ (App (Var TrNameS) (Var $tc'Succ)),
NonRec $tc'Succ (Lit 'Succ),
NonRec $krep (App (App (Var KindRepFun) (Var $krep)) (Var $krep)),
NonRec $tc'Zero (App (App (App (App (App (App (Var TyCon)
(Lit 5578991467370359990)) (Lit 8450640879969411908))
```

```
(Var $trModule)) (Var $tc'Zero)) (Lit 0)) (Var $krep)),
NonRec $tc'Zero (App (Var TrNameS) (Var $tc'Zero)),
NonRec $tc'Zero (Lit 'Zero),
NonRec $krep (App (App (Var KindRepTyConApp) (Var $tcNat)) (Var [])),
NonRec $tcNat (App (App (App (App (App (App (Var TyCon)
(Lit 14728780495088517121)) (Lit 9861982106080716404))
(Var $trModule)) (Var $tcNat)) (Lit 0)) (Var krep$*)),
NonRec $tcNat (App (Var TrNameS) (Var $tcNat)),
NonRec $tcNat (Lit Nat)]
```

The long integer literals seem to contain some module-specific information and are stable across compilation if the file is unchanged. An equivalent definition of the `Nat` data type will have different literal numbers if specified and compiled in another module. If trying to match a student solution with a model solution, it will fail since these numbers are unequal. Another issue is that the names of the data constructors are contained is "special variables" prefixed with `$tc` *and* as string literals, neither of which are renamed by the renaming transformation. Depending on how the exercise is defined, it might be desirable to allow the student to name the constructors freely – data constructors must then be considered in the renaming transformation accordingly.

Matching modules containing data type definitions, as in the above example, requires a more liberal similarity relation or a similarity relation that considers more special cases. The first option is not viable since we generally want to check that literals are equal (recall the `isZero` function from 8.2, if not checking literal equality, a student program defined as `isZero x = x == 100` would be considered correct). The second option, subject to future work, could be to see if it scales and how many special cases would be needed for other types of exercises. A third option, also subject to future work, could be to define an additional similarity relation to match data type definitions.

8.5.4 Allowing Teacher-Defined Feedback

Ask-Elle's current implementation allows teachers to specifically provide model solutions with descriptive annotations that are part of the feedback displayed to the student. The annotation pragmas used in Ask-Elle are not standard GHC pragmas; they are not parseable by GHC and thus not part of the Core AST. This means that teacher annotations cannot be used as feedback when displaying hole-matching suggestions. However, the annotation pragmas are parsed by a separate parser and are neither in Ask-Elle part of the AST of a model program. Hence, when integrating the new normalisation approach into Ask-Elle, the annotation parser can be refined to include source locations and be used when creating feedback for hole-matches.

8.5.5 Refine Feedback and Hole-Matching

The feedback generated by the new approach should be refined to better serve the purpose of being understandable for new programmers. The warning messages

should be rewritten at an adequate level instead of displaying GHC's warning message directly. For instance, consider that the student attempts:

```
dupli [] = []
```

The generated feedback would look as follows:

```
Solution is on track.

You have an incomplete pattern:
Pattern match(es) are non-exhaustive
In an equation for `dupli':
    Patterns of type `[a]' not matched: (_:_)
```

The string "Pattern match(es) are non-exhaustive" could simply be dropped, and "In an equation" could, for example, be simplified to "in the definition of".

Moreover, the hole-fits should not be displayed all at once. It is better to let the student take the next step by providing *refinement hole-fits*, i.e., just hinting at how to refine the hole in one step and replacing additional terms with new holes. The strategies from Ask-Elle could be reused for this purpose. Currently, the hole-fits for each hole are displayed. For example, consider the partial solution to the `dupli` exercise:

```
dupli =  _
```

With the current implementation, the entire hole-fit is displayed directly:

```
Solution is on track:
Perhaps use foldl (\acc x -> acc ++ [x,x]) []
```

8.6 Related Work

The main focus of this thesis project was not to change the Ask-Elle itself but to extend and improve normalisation. We will only discuss work related to the additions to Ask-Elle. A related work of particular concern is the Master's thesis by Engsmyre and Wikström from 2021, where they developed a program synthesis tool based on sketching for Ask-Elle. Their tool, called THUPY, served as a proof of concept that program synthesis could be incorporated in Ask-Elle [29]. THUPY is limited to synthesising terms by matching terms from the model solution after applying an $\eta-$conversion step. Similar to our approach, they utilised GHC and the typed holes mechanism to retrieve the hole type but did not use the typed hole suggestions available from GHC. Instead, they only used the type information to find terms of matching types in the model solution by enumerating all terms in the model solution before inserting each term into the hole and QuickChecking the resulting program to test whether this term was a valid hole-fit. The hole-matching approach described in subsection 6.3.2 achieves similar results (finding terms in the model solution matching a hole) without the need for enumerating, searching and testing (which can be a time-consuming process).

Using GHC for normalisation in a programming tutor is a quite specific usage area, and there is not (expectedly so) a lot of related work on this subject. Other work has been done on GHC's Core language, like the Hermit tool [30] that implements optimising transformations as a Core plugin. The tool is used for interactive exploration of optimisations of Haskell programs in a command line tool. The program transformation implemented in this thesis could also have been implemented as a Core plugin, allowing reuse for normalisation purposes. There also exists work on refactoring Haskell programs like HaRe [31], which can be seen as applying program transformations directly to the source code. HaRe operates on the parsed AST to be able to retain source locations and keep as much information as possible after refactoring. It implements transformations such as rewriting if-expressions to case-expressions and lifting definitions to the top level, but the number of available refactorings is limited. It is available as a Haskell library[2] and allows implementation of new transformations, although it is only compatible with earlier versions of GHC. It would be interesting to explore if refactoring tools like [31] could be used to rewrite student programs as a first step to match student programs with model solutions. However, a problem with such an approach is that even though it might improve *recognition*, it might be difficult to perform hole-matching and provide hole-fit suggestions. An advantage is that the transformations are performed at the parsed AST, and it is possible to retrieve the source code representation directly.

[2]https://hackage.haskell.org/package/HaRe

9

Conclusion

We can conclude with positive answers to both questions defined in section 1.2: GHC's core language and internal program transformations *can* be used to improve normalisation and recognition of student programs in Ask-Elle, and GHC can also be used to give additional feedback, retrieved from warnings and the HLint tool. However, Core has some drawbacks when implementing new transformations since the number of available data types is small, and some information is contained in strings and literals, making modifications on the Core AS less type-safe and more error-prone.

Even though it, at times, has been a battle against GHC to utilise it for purposes it is not intended for, the new approach of normalising, matching and providing feedback to student programs has several benefits:

- Compiling programs to Core already gives complete programs a decent level of normalisation.
- Hole-matching, with some limitations, can be performed instead of searching through a (possibly) large set of intermediate model solutions, which is significantly faster.
- It enables analysis of other coding problems like overlapping patterns by utilising HLint and GHC's warning messages for feedback.
- It allows students defining type signatures in their solutions.
- It gives a starting point for future work on including exercises on more Haskell constructs than functions.
- GHC's modular interface allows adding own and other compiler plugins for future add-ons.

Some work remains to improve recognition further and refine the additional feedback; for example, by finding a suitable way to do η-conversion for programs with holes and detecting redundant patterns. Using Core as the internal representation of programs in Ask-Elle facilitates including other constructs than functions in Ask-Elle exercises, but it requires future work on normalisation and matching.

Bibliography

[1] M. McCracken, V. Almstrum, D. Diaz, *et al.*, "A multi-national, multi-institutional study of assessment of programming skills of first-year cs students," in *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education*, ser. ITiCSE-WGR '01, Canterbury, UK: Association for Computing Machinery, 2001, pp. 125–180, ISBN: 9781450373593. DOI: 10.1145/572133.572137. [Online]. Available: https://doi.org/10.1145/572133.572137.

[2] M.-H. Nienaltowski, M. Pedroni, and B. Meyer, "Compiler error messages: What can help novices?" In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, ser. SIGCSE '08, Portland, OR, USA: Association for Computing Machinery, 2008, pp. 168–172, ISBN: 9781595937995. DOI: 10.1145/1352135.1352192. [Online]. Available: https://doi.org/10.1145/1352135.1352192.

[3] T. Flowers, C. Carver, and J. Jackson, "Empowering students and building confidence in novice programmers through gauntlet," Nov. 2004, T3H/10–T3H/13 Vol. 1, ISBN: 0-7803-8552-7. DOI: 10.1109/FIE.2004.1408551.

[4] M. Kölling, B. Quig, A. Patterson, and J. Rosenberg, "The bluej system and its pedagogy," *Computer Science Education*, vol. 13, Dec. 2003. DOI: 10.1076/csed.13.4.249.17496.

[5] S. M. Algaraibeh, T. A. Dousay, and C. L. Jeffery, "Integrated learning development environment for learning and teaching c/c++ language to novice programmers," in *2020 IEEE Frontiers in Education Conference (FIE)*, 2020, pp. 1–5. DOI: 10.1109/FIE44824.2020.9273887.

[6] S. J. Whittall, W. A. C. Prashandi, G. L. S. Himasha, D. I. De Silva, and T. K. Suriyawansa, "Codemage: Educational programming environment for beginners," in *2017 9th International Conference on Knowledge and Smart Technology (KST)*, 2017, pp. 311–316. DOI: 10.1109/KST.2017.7886101.

[7] K. VanLehn, "The relative effectiveness of human tutoring, intelligent tutoring systems, and other tutoring systems," *Educational Psychologist*, vol. 46, no. 4, pp. 197–221, 2011. DOI: 10.1080/00461520.2011.611369. eprint: https://doi.org/10.1080/00461520.2011.611369. [Online]. Available: https://doi.org/10.1080/00461520.2011.611369.

[8] A. N. Kumar, "The effect of using problem-solving software tutors on the self-confidence of female students," in *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, ser. SIGCSE '08, Portland, OR, USA: Association for Computing Machinery, 2008, pp. 523–527, ISBN:

9781595937995. DOI: 10.1145/1352135.1352309. [Online]. Available: https://doi.org/10.1145/1352135.1352309.

[9] B. Harrington, S. Peng, X. Jin, and M. Khan, "Gender, confidence, and mark prediction in cs examinations," in *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE 2018, Larnaca, Cyprus: Association for Computing Machinery, 2018, pp. 230–235, ISBN: 9781450357074. DOI: 10.1145/3197091.3197116. [Online]. Available: https://doi.org/10.1145/3197091.3197116.

[10] B. Spieler, L. Oates-Indruchova, and W. Slany, "Female students in computer science education: Understanding stereotypes, negative impacts, and positive motivation," *Journal of Women and Minorities in Science and Engineering*, vol. 26, pp. 473–510, Oct. 2020. DOI: 10.1615/JWomenMinorScienEng.2020028567.

[11] A. Gerdes, "Ask-elle : A haskell tutor," Ph.D. dissertation, Heerlen, 2012.

[12] A. Gerdes, B. Heeren, J. Jeuring, and L. T. van Binsbergen, "Ask-elle: An adaptable programming tutor for haskell giving automated feedback," *International Journal of Artificial Intelligence in Education*, vol. 27, Feb. 2016. DOI: 10.1007/s40593-015-0080-x.

[13] B. Heeren, J. Jeuring, and A. Gerdes, "Specifying rewrite strategies for interactive exercises," *Mathematics in Computer Science*, vol. 3, pp. 349–370, May 2010. DOI: 10.1007/s11786-010-0027-4.

[14] K. Claessen and J. Hughes, "Quickcheck: A lightweight tool for random testing of haskell programs," *SIGPLAN Not.*, vol. 35, no. 9, pp. 268–279, Sep. 2000, ISSN: 0362-1340. DOI: 10.1145/357766.351266. [Online]. Available: https://doi.org/10.1145/357766.351266.

[15] E. Visser, "A survey of rewriting strategies in program transformation systems," *Electronic Notes in Theoretical Computer Science*, vol. 57, pp. 109–143, 2001, WRS 2001, 1st International Workshop on Reduction Strategies in Rewriting and Programming, ISSN: 1571-0661. DOI: https://doi.org/10.1016/S1571-0661(04)00270-1. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1571066104002701.

[16] H. P. Barendregt, *The lambda calculus: its syntax and semantics* (Studies in logic and the foundations of mathematics v. 103), Rev. ed. Amsterdam ; New York : New York, N.Y: North-Holland ; Sole distributors for the U.S.A. and Canada, Elsevier Science Pub. Co, 1984, ISBN: 9780444867483 9780444875082.

[17] S. Peyton Jones and S. Marlow, "Secrets of the glasgow haskell compiler inliner," *J. Funct. Program.*, vol. 12, pp. 393–433, Jul. 2002. DOI: 10.1017/S0956796802004331.

[18] T. Fausak, *Haskell survey*, 2022. [Online]. Available: https://taylor.fausak.me/2022/11/18/haskell-survey-results/#s2q0.

[19] J.-Y. Girard, "The system f of variable types, fifteen years later," *Theoretical Computer Science*, vol. 45, pp. 159–192, 1986, ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(86)90044-7. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0304397586900447.

[20] Andrew Tolmach, Tim Chevalier, GHC Team, *An external representation for the ghc core language (for ghc 6.10)*, Jul. 2009. [Online]. Available: https:

//downloads.haskell.org/~ghc/7.8.3/docs/html/users_guide/an-external-representation-for-the-ghc-core-language-for-ghc-6.10.html.

[21] M. Sulzmann, M. M. T. Chakravarty, S. P. Jones, and K. Donnelly, "System f with type equality coercions," in *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, ser. TLDI '07, Nice, Nice, France: Association for Computing Machinery, 2007, pp. 53–66, ISBN: 159593393X. DOI: 10.1145/1190315.1190324. [Online]. Available: https://doi.org/10.1145/1190315.1190324.

[22] "Corelint," GHC Team. (2023), [Online]. Available: https://ghc-compiler-notes.readthedocs.io/en/latest/notes/compiler/coreSyn/CoreLint.hs.html#note-checking-for-global-ids.

[23] "Linear types," GHC Team. (2023), [Online]. Available: https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/linear_types.html.

[24] (2019), [Online]. Available: https://ghc-proposals.readthedocs.io/en/latest/proposals/0111-linear-types.html.

[25] J.-P. Bernardy, M. Boespflug, R. R. Newton, S. Peyton Jones, and A. Spiwack, "Linear haskell: Practical linearity in a higher-order polymorphic language," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, Dec. 2017. DOI: 10.1145/3158093. [Online]. Available: https://www.microsoft.com/en-us/research/uploads/prod/2017/12/linear-haskell-popl18-with-appendices.pdf.

[26] P. Wadler, "Linear types can change the world!" In *PROGRAMMING CONCEPTS AND METHODS*, North, 1990. [Online]. Available: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.3s1.5002&rep=rep1&type=pdf.

[27] S. Diehl. [Online]. Available: https://www.stephendiehl.com/posts/ghc_01.html.

[28] S. L. P. Jones, "Compiling haskell by program transformation: A report from the trenches," in *Programming Languages and Systems — ESOP '96*, H. R. Nielson, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 18–44, ISBN: 978-3-540-49942-8.

[29] Gustav Engsmyre and Karl Wikström, "Complementing the digital programming tutor ask-elle with program synthesis," M.S. thesis, Chalmers University of Technology, University of Gothenburg, 2021.

[30] A. Farmer, A. Gill, E. Komp, and N. Sculthorpe, "The hermit in the machine: A plugin for the interactive transformation of ghc core language programs," *SIGPLAN Not.*, vol. 47, no. 12, pp. 1–12, Sep. 2012, ISSN: 0362-1340. DOI: 10.1145/2430532.2364508. [Online]. Available: https://doi.org/10.1145/2430532.2364508.

[31] H. Li, S. Thompson, and C. Reinke, "The haskell refactorer, hare, and its api," *Electronic Notes in Theoretical Computer Science*, vol. 141, no. 4, pp. 29–34, 2005, Proceedings of the Fifth Workshop on Language Descriptions, Tools, and Applications (LDTA 2005), ISSN: 1571-0661. DOI: https://doi.org/10.1016/j.entcs.2005.02.053. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S157106610505173X.

A

Appendix 1

A.1 The Ask-Elle AS

```
-- Alt ---------------------------------------------------
data Alt = AHole (HoleID)
         | Alt ((Maybe String)) (Pat) (Rhs)
         | AltEmpty
         deriving ( Data,Eq,Ord,Show,Typeable)
-- Alts --------------------------------------------------
type Alts = [Alt]
-- Body --------------------------------------------------
data Body = BHole
          | Body (Decls)
          deriving ( Data,Eq,Ord,Show,Typeable)
-- Decl --------------------------------------------------
data Decl = DHole (HoleID)
          | DEmpty
          | DFunBinds (FunBinds)
          | DPatBind (Pat) (Rhs)
          deriving ( Data,Eq,Ord,Show,Typeable)
-- Decls -------------------------------------------------
type Decls = [Decl]
-- Expr --------------------------------------------------
data Expr = Hole (HoleID)
          | Feedback (String) (Expr)
          | MustUse (Expr)
          | Eta (Int) (Expr)
          | Refactor (Expr) (RefactorChoices)
          | Case (Expr) (Alts)
          | Con (Name)
          | If (Expr) (Expr) (Expr)
          | InfixApp (MaybeExpr) (Expr) (MaybeExpr)
```

```
          | Lambda (Pats) (Expr)
          | Let (Decls) (Expr)
          | Lit (Literal)
          | App (Expr) (Exprs)
          | Paren (Expr)
          | Tuple (Exprs)
          | Var (Name)
          | Enum (Expr) (MaybeExpr) (MaybeExpr)
          | List (Exprs)
          | Neg (Expr)
          deriving ( Data,Eq,Ord,Show,Typeable)
-- Exprs ---------------------------------------------------
type Exprs = [Expr]
-- FunBind -------------------------------------------------
data FunBind = FBHole (HoleID)
             | FunBind ((Maybe String)) (Name) (Pats) (Rhs)
             deriving ( Data,Eq,Ord,Show,Typeable)
-- FunBinds ------------------------------------------------
type FunBinds = [FunBind]
-- GuardedExpr ---------------------------------------------
data GuardedExpr = GExpr (Expr) (Expr)
                 deriving ( Data,Eq,Ord,Show,Typeable)
-- GuardedExprs --------------------------------------------
type GuardedExprs = [GuardedExpr]
-- Literal -------------------------------------------------
data Literal = LChar (Char)
             | LFloat (Float)
             | LInt (Int)
             | LString (String)
             deriving ( Data,Eq,Ord,Show,Typeable)
-- MaybeExpr -----------------------------------------------
data MaybeExpr = NoExpr
               | JustExpr (Expr)
               deriving ( Data,Eq,Ord,Show,Typeable)
-- MaybeName -----------------------------------------------
data MaybeName = NoName
               | JustName (Name)
               deriving ( Data,Eq,Ord,Show,Typeable)
-- Module --------------------------------------------------
data Module = Module (MaybeName) (Body)
            deriving ( Data,Eq,Ord,Show,Typeable)
-- Name ----------------------------------------------------
data Name = Ident (String)
          | Operator (String)
          | Special (String)
          deriving ( Data,Eq,Ord,Read,Show,Typeable)
```

```
-- Names -----------------------------------------------------
type Names = [Name]
-- Pat -------------------------------------------------------
data Pat = PHole (HoleID)
         | PMultipleHole (HoleID)
         | PCon (Name) (Pats)
         | PInfixCon (Pat) (Name) (Pat)
         | PList (Pats)
         | PLit (Literal)
         | PParen (Pat)
         | PTuple (Pats)
         | PVar (Name)
         | PAs (Name) (Pat)
         | PWildcard
        deriving ( Data,Eq,Ord,Show,Typeable)
-- Pats ------------------------------------------------------
type Pats = [Pat]
-- RefactorChoice --------------------------------------------
data RefactorChoice = RefactorChoice (String) (Expr)
                    deriving ( Data,Eq,Ord,Show,Typeable)
-- RefactorChoices -------------------------------------------
type RefactorChoices = [RefactorChoice]
-- Rhs -------------------------------------------------------
data Rhs = Rhs (Expr) (Decls)
         | GRhs (GuardedExprs) (Decls)
        deriving ( Data,Eq,Ord,Show,Typeable)
```

A.2 GHC Flag Configuration

Flag configuration for GHC. Some flags are set explicitely, some flags are set by default and some default flags have been unset.

Enabled non-default general flags:

```
Opt_DoCoreLinting -- enables the Core Linter
Opt_InfoTableMap  -- generates source note Ticks
Opt_EnableRewriteRules -- enables rewrite rules by the RULES pragma
```

Enabled non-default warning flag:

```
Opt_WarnIncompletePatterns
```

Disabled default warning flags:

```
Opt_WarnUnrecognisedPragmas
Opt_WarnInlineRuleShadowing
```

Enabled non-default extension flag:

```
ExtendedDefaultRules -- extend type defaulting for non-numeric types
```