Using GHC CORE to Normalise Student Programs

Matilda Blomqvist[1] and Alex Gerdes[2,3]

[1] Uppsala University, Uppsala, Sweden
matilda.blomqvist@it.uu.se
[2] Gothenburg University, Gothenburg, Sweden
[3] Chalmers University of Technology, Gothenburg, Sweden
alexg@chalmers.se

Abstract. Ask-Elle is an online tutor for solving small programming exercises in Haskell. Haskell offers a rich syntax that allows writing semantically but not syntactically equivalent programs, making comparing 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, which uses GHC's internal transformations and warning messages to improve the generated feedback. The new approach shows an improvement in the number of recognised programs, and is much faster. In addition, we share our experiences on using GHC as a library.

Keywords: Programming tutor · Functional programming · Haskell · GHC · Program normalisation

1 Introduction

Learning to program is challenging, even with access to suitable course materials and good teachers [17]. It requires practice and the teacher may not always be around to guide students. Students write source code, compile it, and then run usually the resulting program to test if it works as expected. Modern compilers point out errors at different levels in the source code, but these are often difficult to decipher for beginners [18]. Intelligent tutor systems aim to do a better job and support students by giving *semantically rich feedback* [13,2,25].

A review study by VanLehn concluded that intelligent tutor systems are almost as efficient as human tutors [24]. Including intelligent tutors in education can have many benefits, for example, they are always available, can give instant feedback, can handle large student numbers, and may possibly save course instructors time. In addition, a study by Kumar [14] showed that using intelligent programming tutors can help improve female computer science (CS) students' confidence. Several studies [9,21] have shown that the confidence of female CS students is lower than their male counterparts, and if digital tutors can enhance the confidence of female students, we have a strong motivation to further develop them.

Ask-Elle is an online tutor aiming to improve students' programming skills in the functional language Haskell by providing comprehensible feedback [7]. Figure 1 displays a screenshot of the learning environment. The tutor focuses on small programming tasks and allows a student to *stepwise* solve an exercise. A teacher can specify an exercise by giving a number of model solutions and properties that a solution should have. Ask-Elle uses *model tracing* based on strategies [10] to match a student program with one of the model solutions, and if it succeeds it can generate feedback on different levels. If a student program cannot be matched, Ask-Elle reverts to property-based testing with QuickCheck [4] to validate if the program has the expected behaviour.

We show a fictional interaction with Ask-Elle, where we ask a student to define a function that duplicates all elements in a list. Suppose a student has implemented the base case, but gets stuck defining the recursive case:

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

In Ask-Elle a student can ask for help even if parts of the program are yet to be defined. The undefined parts are indicated by a question mark and correspond to a (typed) *hole*. Using these holes a student can check if she is on the right track, and ask how to continue. Ask-Elle will respond that the student can use explicit recursion and refine the program by introducing the cons-operator `(:)`. It can also show the result of the proposed refinement to the student:

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

or show a complete derivation of how to construct a solution.

Although Ask-Elle can recognise many incomplete programs and guide students towards a solution, it does have its limitations. Consider the following incomplete program:

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

Ask-Elle unfortunately fails to recognise this program, even though it is essentially the same program as we have showed earlier. The problem is that the strategies, on which our model tracing is based, is too strict and does not recognise such variations.

To be more flexible Ask-Elle already uses *program transformations* to ignore non-essential differences. Ask-Elle normalises a student program while doing model tracing. It uses, for example, a program transformation for rewriting `where`-clauses to `let`-expressions, and another transformation that does α-renaming, such that a student can use different variable names.

To also recognise the above variation, we could improve the existing normalisation procedure by adding more and more program transformations. The problem is that we would need to implement very many program transformations, and by doing so we would duplicate many, if not all, of the program

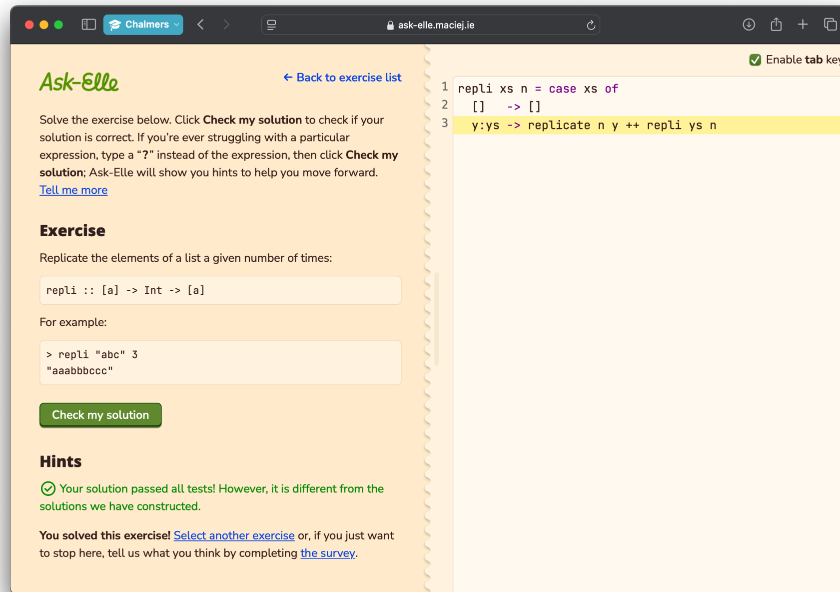

Fig. 1. Screenshot of the Ask-Elle web interface. The student writes code in the editor to the right, and the exercise description and generated feedback are displayed in the panel to the right.

transformations found in an Haskell compiler. In addition, Ask-Elle has an own abstract syntax tree (AST) for expressions, and we would probably need to restrict it to only allow a subset of expressions, such that we can implement the necessary program transformations. Such a subset would of course resemble an intermediate (core) language in a Haskell compiler.

Another problem with our current model tracing approach is that it can only handle small programs, because the usage of strategies gives rise to a search space explosion. We explain this in more detail in Section 2.

Instead of reinventing the wheel, we propose to use the program transformations and core language (GHC Core) of the 'de facto' standard Haskell compiler GHC [19]. GHC is written in Haskell and offers much of its functionality via a library. Using this library we can use the different stages available in the compiler, such as the parser, type checker, and, most important, the program transformations.

Our contributions are the following:

– We have created a new procedure for normalising student programs (Section 4) based on GHC Core (named core hereafter) that uses program transformations presented in GHC (Section 4.2), such as α-renaming, desugaring, inlining, and more. In addition to these predefined transformations we

have created a range of new, complementing CORE to CORE transformations (Section 4.3).
- We introduce a new way of model tracing, which we call: hole mapping. This improved way of model tracing (Section 3) exploits the new normalisation procedure to recognise student programs. We can still recognise the same variation of student programs as we could with strategies, but we can recognise many more, including the example we gave earlier. Moreover, we manage to avoid the search space explosion, which the strategy based approach suffers from.
- By using GHC we gain access to more information about a student program, which we can use as feedback. For example, GHC detects if there are overlapping patterns in a definition. We show how we can generate feedback (Section 4.2) when we tap into this source of information.
- We evaluate and share our experiences we gained with using GHC as a library (Section 5.3). In addition, we evaluate the improved feedback generation and compare it the existing version (Section 5.1).

Before we delve into the intricacies of our improvements, we first detail the problems with the existing way of model tracing.

2 The problem

The `dupli` example in the previous section shows that ASK-ELLE has its limitations, and that we cannot recognise particular variations of a model solution (a solution given by a teacher). On top of this, ASK-ELLE can only handle relatively small programs.

Strategies The current version of ASK-ELLE is based on strategies [10], which is a description of valid sequences of rewrite rules to solve a class of exercises in a particular domain. A strategy can be regarded as a context-free grammar and recognising a valid sequence of steps boils down to parsing a sentence. Strategies are implemented as an embedded domain-specific language in Haskell, and the library offers many combinators for creating strategies, such as sequence, choice, interleave, etc. It has been used for generating feedback in many domains, for example in propositional logic [16].

Using strategies we view solving a programming exercise as a sequence of refinement steps. A student refines a program step by step, for example, she introduces first a function binding, then a pattern match, then a function application, and so on. A model solution is automatically translated to such a strategy, a teacher does not need to do this by hand. Using the strategy we generate intermediate solutions and check if a student program matches one of these, if so, the we can conclude that the student is making progress towards one of the model solutions. Furthermore, we then know exactly where a student is in the development of a solution, and we can provide hints on how to continue.

The domain of functional programming is a bit special in the context of strategies, since there are many ways in which a student can stepwise develop a

solution to a programming exercise. This variation is captured in the strategy for a given exercise. We use the choice combinator to let a student solve an exercise with different model solutions. The interleave combinator is used to allow a student to work on different parts of a solution. Consider a variant of the `dupli` example:

```
dupli []     = ?1
dupli (x:xs) = ?2 : ?3
```

A student is not forced to refine the `dupli` in a particular order, she can choose to refine the hole `?1` to the empty list, or the second hole to the variable `x` or to continue with the third hole and introduce another cons operator.

This degree of freedom is, however, quite problematic since the number of intermediate solutions has a factorial order of growth ($O(n!)$), where n is the number of holes. The above `dupli` example can be solved in six different ways. To counter this ASK-ELLE implements a number of improvements:

- We assume that a student usually finishes a particular construct (e.g., a function binding), before continuing to the next. So, we do a depth first search when we try to match a student program to an intermediate solution.
- We try to rule out other model solutions as quickly as possible, which limits the search space drastically.
- Many of the different order of refinements (permutations) will lead to the same intermediate solution. For example, refine the first hole and then the second, or the other way around will lead to the same intermediate solution, where just the third hole is left. For recognising it does not matter in which order we have arrived at a particular intermediate solution. ASK-ELLE has therefore a different semantic interpretation of the interleave combinator that prevents the generation of duplicate intermediate programs.

These improvements make ASK-ELLE much faster, but the order of growth of the number of intermediate solutions remains factorial.

Helium Until now ASK-ELLE has used the Helium [11] compiler, known for its good (type) error messages. We reuse a large part of the front-end of the Helium compiler. After the type checking stage we convert the program from Helium's abstract syntax tree to our own AST, just for ASK-ELLE. This AST is smaller than the one in Helium, which makes defining refinement rules simpler, because there are fewer language constructs to cover. The Helium compiler has been a great aid in developing ASK-ELLE, but it has a few limitations as well. For example, it does not support user defined type classes and does not completely adhere to the Haskell98 standard. Although, these limitations are far from severe for ASK-ELLE, exchanging Helium with GHC will make ASK-ELLE more future proof. In addition, Helium is quite a large software project with many modules and dependencies, and with ever changing compiler versions it is good from a software engineering perspective to get rid of such a large dependency. Of course, we will still be dependent on GHC, but this is not new.

3 Model tracing based on normalisation

Model tracing based on strategies served us well for many years, but it has its limitations as we explained. We want to do a better job, such that we can handle larger programs. Our main idea is to use a technique that we have coined 'hole mapping'. It works as follows, suppose a teacher has defined following model solution for the `dupli` exercise:

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

and that a student submits the following incomplete program:

```
dupli []     = []
dupli (x:xs) =  x : ?1 : ?2
```

Instead of using strategies and enumerate all intermediate solutions, we just compare the student program syntactically with the teacher defined model solution. In case we encounter a hole in a student program we match it with the corresponding expression in the model solution. So, we can match the hole `?1` with the variable `x` in the model solution, and the second hole is being matched with the expression `dupli xs`. We can create a mapping from holes to expressions, hence the name 'hole mapping'.

If we succeed to create such a mapping, that is, the two programs are syntactically equivalent modulo holes, then we can conclude that the student is on their way to creating a program that we consider to be a solution to the exercise. That is in itself good to know because we can signal to the student that she is making progress. In addition, we can use the mapping to provide the student with a hint on how to continue. We can, for example, show that the first hole needs to be replaced with the variable `x`, just like we were able to do with strategies. There is a catch, because we match a hole with an entire expression, we will give it as the next refinement step, if a student ask for help. Whereas, with strategies, we can present the student with more fine-grained steps. In the above example, our new approach would suggest to refine the second hole to `dupli xs`, and using strategies we can give `dupli ?` or `dupli xs` as the next steps for the recursive call.

Now suppose a student submits the following program we showed earlier:

```
dupli = \xs -> case xs of
  []        -> []
  (:) x xs -> ?1 : ?2
```

This will clearly fail when we try to match it syntactically to the given model solution. Like we argued before, this is in essence the same program as the model solution, and we want to be able to match it. Now we need to introduce the second part of our new model tracing approach: we are going to *normalise* both the student program and model solution *before* we are going to match them syntactically.

Assume that the following program is the normalised form of the model solution:

```
dupli = \x0 -> case x0 of
  []        -> []
  (:) x1 x2 -> (:) x1 ((:) x1 (dupli x2))
```

The infix operators are written prefix, new unique names have been introduced, the function bindings have been rewritten as a `case` expression, etc. This is a simplification and in the next section we will explain the actual normalisation procedure, but it comes quite close. If we also normalise the student program and then try to match, we will succeed. However, now the resulting mapping contains normalised expressions, that is, the hole `?1` is matched with the variable `x1` (which is not even bound in the student program) and the second hole is paired with `(:) x1 (dupli xs)`. We cannot give this as feedback to the student.

To solve this problem we need to remember during normalisation to which part of the original model solution a rewritten (normalised) subexpression belongs. For example, we need to remember that the subexpression `x1` is the variable `x`. We are in a good spot here using GHC because every (sub)expression is annotated with its origin in the concrete syntax. We get this information (nearly) for free when using GHC. With this information we can make a mapping again from holes in a student program to parts of a model solution, and present these as next steps.

The above procedure does not take care of different variable names. In such a case the student would get a hint with the names from a model solution, instead of her own. These names may be unbound or capture something else. Again, GHC comes to the rescue, because it gives all variable a new unique name, and it remembers the original name (for error reporting purposes). We can use this information to create a mapping from student variable names to model variable names. For example, if we know that the student variable `y` is renamed to `x1`, and the model variable name from `x` to `x1`, then we know that need to substitute `x` with `y` in our next step hints.

We are well aware that it is in general undecidable to validate if two programs are equal, and there will always be cases where we fail to match equal programs, but that does not mean that you can come quite far in practice. Say that a student submits an intrinsically different program, for example:

```
dupli xs = concatMap (\x -> [x, x]) xs
```

We will not be able to match it with the given model solution. In such a case, we need to revert to property-based testing, to validate whether the student program at least has the expected behaviour. If you also want to recognise this student program, you can add it to the collection of model solutions.

We have explained the new 'hole mapping' procedure in a bit simplified manner, to get the general idea across. In the next section we show how we use the GHC intermediate CORE language to normalise programs.

4 Normalisation using GHC CORE

GHC contains several compilation passes that transform Haskell source code into executable code[4]. The program transformations of interest for normalisation are mainly part of the *desugarer* that translates a program to CORE, and the *simplifier* performing additional transformations.

CORE is an implementation of System FC [8], an extension of System F, and shares some of the most fundamental constructs found in regular Haskell, but is much smaller. CORE consists of a collection of data types and type synonyms. The most central data type is `Expr`:

```
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
```

which only has a handful of constructors and abstracts over the type of a binding. The `Expr` data type uses other data types, but we omit them for brevity. 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 `Bind` data type used in a `Let` supports two types of binders: recursive and non-recursive. The `Case` constructor corresponds to a `case`-expression in Haskell in the sense that it can be used to condition over data types. `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. In addition, `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. Furthermore, expressions can be annotated with extra information using the `Tick` constructor, such as breakpoints or source locations. As we will see, this is important for our feedback generation.

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

[4] https://gitlab.haskell.org/ghc/ghc/-/wikis/commentary/compiler/hsc-main

by GHC. GHC's *typed holes*, denoted _, 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 type checker. However, no explicit constructs exist for holes on the core syntax level which would have been convenient for our purposes. Holes are treated as type errors and are compiled into core case expressions.

4.2 Using GHC's core Transformations

Many of the program transformations needed for normalising programs in Ask-Elle, like β-conversion, η-conversion and other program rewrite transformations, are already implemented in GHC. Using GHC as a library, we can use these internal transformations. GHC's main purpose, though, is to optimise the compiled code, which is different from Ask-Elle's goal, namely to determine if two programs are syntactically equal. Hence, GHC's transformations are not always helpful in normalisation. It is important to find the right configuration of GHC, by means of compilation options, such that GHC's transformations can be maximally utilised while at the same time not losing information from the original program.

The transformations GHC applies while translating Haskell source code into core solve the following normalisation steps directly:

Local and top-level functions: GHC's inlining of non-recursive local definitions from `where`-clauses and `let`-expressions, in many cases, allows arbitrary use of local helper functions. An exception to this is where type differences 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.

Overlapping patterns: Overlapping patterns are removed by GHC and a warning is generated. We pass this information on to the student as feedback.

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

β-abstraction: this is always performed.

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

Including the simplifier pass in compilation also removes redundant equality checks, e.g., an expression like `if x==y then True else False` is simplified to `x==y`. However, the simplifier pass is quite aggressive and might remove code that we want to retain.

It is not possible to rely solely on GHC's internal transformations to normalise Haskell programs in Ask-Elle. Before GHC applies a particular transformation or rewrite rule to an expression, it analyses if it will lead to a performance gain. For example, the inlining of expressions is based on heuristics, 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 control the application of some transformations.

4.3 New program transformations

When a core program is obtained, many of the transformations we want to include in normalisation are already applied. The remaining issues are due to a lack of control over which transformations are applied, especially by the simplifier, and the non-determinism in GHC's application of its transformations mentioned above. This is solved by adding new program transformations.

First, we inject new transformations between the desugarer and the simplifier to remove redundant information from the AST and avoid losing information in the simplifier. Since typed holes are treated as type errors and are, like other errors (e.g., *pattern errors*, which are inserted into the AST when GHC detects an incomplete pattern) compiled into core case expressions, which are strict, code occurring after e.g., a hole might be removed by the simplifier. Typed holes and pattern errors are distinguishable by string literals "typError" and "patError" in the case expression. We manually transform typed holes to variables prefixed with "hole" and pattern errors to a variable "patError" either in a case expression with empty alternatives or as an alternative in a case expression.

We also explicitly revert a non-configurable transformation by the simplifier that 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 an incomplete student programs with a single function binding and thus no case expressions.

Moreover, both inlining and η-conversion are reimplemented to obtain a deterministic version of these transformations at the core syntax. We also rewrite recursive top-level functions with recursive let expressions. This transformation is exemplified by applying it to the `length` function:

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

which is rewritten to:

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

CORE programs must also be renamed for student programs to match a model solution. It is relatively straightforward to implement since GHC already assigned unique names to all variables. 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 previously discussed transformations since these transformations might reorder, remove or introduce new variables. 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. For instance, when a student only defines a single function binding with no or general pattern for the input argument, the compiled CORE program will not contain a `Case` expression. Renaming the case binder[5] in the model solution would offset all succeeding variable names, and there would be a name mismatch when trying to 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 is described in subsection 4.4.

Finally, type variables and type evidence variables are passed around in CORE expressions and are needed to typecheck CORE programs [22] but clutter the AST with lots of irrelevant information for checking if any two programs are syntactically equal. They can also, depending on how the original source program was written have slightly different placement in the AST. We simply remove any type variables from the AST since any ill-typed program would already have been rejected by GHC's typechecker.

4.4 Matching Programs

Matching two programs requires some definition of 'match'. We match programs on two levels. The similarity between two CORE programs is implemented as two different relations defined in a type class `Similar`:

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

Relating two CORE programs `s ~> m` can be read as "`s` is a predecessor of `m`", meaning that a student program `s` can be *refined* into a model solution `m`. For example, (using concrete syntax) `dupli [] = _  ~> dupli [] = []`. The `~=` relation can sloppily be read as "is syntactically equal to". These two relations compare two CORE programs construct by construct, so we define an instance

[5] A CORE case expression binds the result of the scrutinee with a variable.

`Similar` for all CORE data types. The predecessor relation is more liberal and can consider different terms to be similar, for example, when dealing with holes. Comparing the different CORE constructs should be done with care and the implementation is quite intricate. We will not go into every detail, but rather give some examples to showcase this.

First, consider the following model solution:

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

and that a student submits the following partial solution:

```
dupli _ = []
```

Compiling these two programs into CORE and then syntactically compare them will fail. However, the student program is probably not complete and we want to consider it as a predecessor of the model solution. The wildcard pattern in the function binding of the student program matches *all* input, and the corresponding CORE program will not be translated to a `Case` expression, since the given pattern covers all cases. Whereas, the model solution is translated to a `Case` expression. We therefore check if the student expression matches any alternative in the `Case` expression in the model solution.

Second, assume that a student starts writing a solution using explicit recursion and inserts a hole, postponing the definition of the recursive call. Such a function would be bound by a non-recursive binder, while the corresponding model solution, which also uses explicit recursion, will be bound with a recursive binder. Despite having different types of binders, the student program should be considered a predecessor to the model solution, if the other parts of the programs do match.

4.5 Hole-Matching

The new normalisation procedure allows matching student programs with holes directly against model solutions to relate holes with terms from the 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.

From the obtained list of CORE expressions from the model solution that match holes in the student program, we can use its source location to extract the corresponding parsed expression if the outermost expression is a `Tick`. 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/Tmp.hs:4:19-31> (App (Var replicate)
(Tick src<studentfiles/Tmp.hs:4:30> (App (Var I#) (Lit 2))))]
```

The outermost expression is a `Tick`, which contains the source location. The corresponding parsed expression can be extracted from the parsed source which we have stored separately (the AST returned by the parser) using the source location. GHC's API provides functionality to print a parsed expression as originally defined, `(replicate 2)` in the above example. In this case, the original 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 example, consider the model solution and the 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. The list `["list","(n-1)"]` is obtained after applying the hole-matching procedure. To give feedback with the same names the student used, we have stored maps containing the renamed and original variables of the student and model solution, respectively. The student variables corresponding to the variables in the list of hole-fit expressions can then be looked up in the substitution map. In the above example, we can retrieve the translated expressions `xs` and `(i-1)` and use it in the feedback to the student.

Retrieving hole-fits might require additional work, for example, if the hole matches a local helper function in a model solution or the whole right-hand side of a function definition. How to retrieve hole-fits in these cases is described in [3]. Matters are also complicated by the sparse control over `Tick` placements.

5 Evaluation

Our overall goal is to improve the feedback that we can give to a student during a programming exercise. Recognising a student program is important in generating this feedback, and we evaluate if the new approach improves the recognition student programs. Our new approach makes it also possible to use GHC warnings and we evaluate if this improves our generated feedback. Furthermore, we compare the execution time of the new normalisation procedure against the current procedure in Ask-Elle.

Evaluation data We use a data set containing student programs from both real-world student interactions with Ask-Elle and artificial ones. The latter are

designed to capture particular Haskell constructs that are expected to be normalised in a certain way. The student programs are retrieved from two experiments conducted in 2015 and 2023. During these experiments, the students were asked to solve exercises in Ask-Elle while their interactions were logged. The experiment from 2015 consists of 889 student programs, and the second from 2023 resulted in 448 programs.

Both data sets include duplicate programs, programs that do not compile, or have other errors. Programs that have syntax or type errors, or don't pass the tests are discarded. Of course, when a student submits a such a program, Ask-Elle gives a proper feedback message, but these programs are not relevant for evaluating the recognition rate. Furthermore, we ignore duplicate programs[6] as well. After excluding these programs there remain 197 and 113 student programs, respectively.

5.1 Evaluating Feedback

The student programs are categorised as follows:

Complete Programs without holes that are successfully matched with a model solution.

OnTrack Incomplete programs, containing at least one hole, that match one of the model solutions.

HLint Programs that cannot be matched, but where we can give a HLint feedback message that leads to a model solution.

Missing Cases Programs with missing cases, such as functions that are not defined on all possible inputs.

TestPassed We don't run the tests in this evaluation since these use the unmodified student programs. Hence, these programs are assumed to still be in this category if they are not recognised.

Unknown Programs that cannot be matched and are too incomplete to test. In other words, we cannot generate meaningful feedback.

The results of our comparison are summarised in Table 1. The aim is to have many programs in the Complete, OnTrack, and HLint categories, because in those situations we can give better support to a student. We can see that the number of student programs in those categories went up from 150 to 175, an increase of about 17%. Of the in total 310 programs, the new approach can handle 56%. This might seem like a low figure, but in the grand scheme where we count all diagnosed student programs, Ask-Elle can give proper feedback in 1027 of the in total 1337 cases, which is about 77%.

We analysed the 94 programs that fall in the TestPassed and Unknown categories. We noticed the following patterns:

Using disallowed functions: Six programs used a function from the Prelude (Haskell's standard library) corresponding to the exercise they were supposed to implement (e.g., `myreverse = reverse`).

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

Table 1. Comparison of the old and new approach. The HLint category is specific to the new approach.

	Existing procedure			New procedure		
Category	2015	2023	Tot	2015	2023	Tot
Complete	37	21	58	40	25	65
OnTrack	68	25	92	81	27	107
HLint	-	-	-	2	1	3
Missing	43	8	51	32	8	40
TestPassed	33	52	85	30	47	77
Unknown	16	7	23	12	5	17
All	197	113	310	197	113	310

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 programs couldn't be matched due to redundant 'patterns' on the expression level.

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

We will not be able to recognise all of these (and, in some cases, we don't want to), but it would be an improvement if we could detect some of the patterns. For example, detecting redundant patterns and signalling this to the student would be a great improvement.

5.2 Execution Time

Enumerating all possible intermediate model solutions and searching for a matching one, as is the modus operandi currently in ASK-ELLE, takes time. We started out with the hypothesis that the 'hole matching' matching approach, as described in Section 3, will be faster. To assert this, we have created simple benchmark tests using the Criterion[7] library to measure the execution times for generating feedback for both the strategy-based and our new approach. The exact measured execution times are not relevant, but rather the comparison between two approaches.

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

The results, highlighted in Table 2, confirm our hypothesis and show that the new approach is faster than the current approach in Ask-Elle. For complete student programs the new approach is more than twice as fast and in case of incomplete there is a performance gain of about 27%. Note, that we have not optimised the new approach yet, whereas the strategy-based approach is. Supporting larger exercises is likely to increase this difference.

Table 2. Benchmark results

		Strategy-based		New approach	
	Number	Avg. time	σ	Avg. time	σ
Complete 2015	37	41.80s	76.21ms	17.07s	299.3ms
Complete 2023	21	24.19s	40.29ms	10.79s	76.21ms
OnTrack 2015	68	73.29s	517.7ms	55.57s	790ms
OnTrack 2023	25	27.88s	74.77ms	20.03s	342.2ms

5.3 Using GHC as a library

The small number of constructors in the core expression data type makes some transformations on the core syntax harder to implement and require string comparisons and other "hacks" to check certain conditions that are not directly available in a constructor. For example, any kind of conditional, like pattern matching, guards, `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".

We found that 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. We 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 documentation pages do.

Another good starting point for working with core is the informal description found in [23] and a series of blogposts [5], despite being slightly outdated.

Another resource that gives some insight into how the simplifier operates in general and how GHC's inliner works in particular is [20] by Peyton Jones. A more general description of the CORE to CORE transformations in GHC can be found in [12]. Although 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).

6 Related work

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. Related work has been done by the Hermit tool [6] with GHC's CORE language, implementing optimising transformations as a CORE plugin. The tool is used for interactive exploration of optimisations of Haskell programs in a command line tool. The presented program transformations could have also been implemented as a CORE plugin, allowing reuse for normalisation purposes.

There also exists work on refactoring Haskell programs like HaRe [15], 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 refactorisations is limited. It is available as a Haskell library[8] 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 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 on the parsed AST, and it is possible to retrieve the source code representation directly.

In this paper we have only discussed normalisation achieved by applying program transformations, or rewrite rules, on a program. Another approach to normalisation is normalisation by evaluation (NBE). The idea of NBE is to translate a source language into some underlying host language, evaluate it in the host language and *reify*, or translate it back to the source language. Aehlig et. al [1] presents an NBE approach for functional languages, which translate a source language into some underlying functional language, evaluating and translating back to the source language.

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

6.1 Future work

There is ample room for improvement and future work on the new diagnosing approach; we highlight two possibilities.

η-conversion and holes Representing holes in Ask-Elle with GHC's typed holes has many benefits but also drawbacks. 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 an incomplete student solution. Consider the following student program:

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

Since `xs` is not used on the right-hand side in the definition, this cannot be reduced to `dupli = concatMap (replicate 2)`, which is a 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.

Allowing Teacher-Defined Feedback Ask-Elle's current implementation allows teachers to annotate model solutions with descriptive feedback using pragmas. The annotation pragmas used in Ask-Elle are not standard GHC pragmas; they cannot be parsed by GHC and will therefore not end up in the core AST. When we integrate the new normalisation approach in Ask-Elle, the annotation parser should be refined to include source locations, such that it can be used when creating feedback for hole matches.

7 Conclusion

We have shown that we can successfully use GHC's core language and program transformations to normalise student programs, such that we can generate semantically rich feedback. We have introduced a new way of model tracing to generate this feedback. This new method circumvents the search space explosion inherent to the strategy-based approach. The new approach can recognise even more student programs, and can thus deliver good feedback. Furthermore, the new 'hole mapping' is a lot faster then it predecessor.

Using GHC core has been a success, but also a struggle at times. We have shared our experiences with using GHC as a library, such that others can get a head start and know what to look out for.

Acknowledgments. We would like to thank Maciej Goszczycki for generously allowing us to use valuable experiment data he collected for his master's thesis using Ask-Elle.

References

1. Aehlig, K., Haftmann, F., Nipkow, T.: A Compiled Implementation of Normalization by Evaluation. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) Theorem Proving in Higher Order Logics. pp. 39–54. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
2. Algaraibeh, S.M., Dousay, T.A., Jeffery, C.L.: Integrated Learning Development Environment for Learning and Teaching C/C++ Language to Novice Programmers. In: IEEE Frontiers in Education Conference (FIE). pp. 1–5 (2020)
3. Blomqvist, M.: Improving Recognition of Student Programs in Ask-Elle. Master's thesis, Chalmers University of Technology (2023)
4. Claessen, K., Hughes, J.: QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. SIGPLAN Not. **35**(9), 268–279 (sep 2000)
5. Diehl, S.: https://www.stephendiehl.com/posts/ghc_01.html
6. Farmer, A., Gill, A., Komp, E., Sculthorpe, N.: The HERMIT in the Machine: A Plugin for the Interactive Transformation of GHC Core Language Programs. SIGPLAN Not. **47**(12), 1–12 (sep 2012)
7. Gerdes, A., Heeren, B., Jeuring, J., van Binsbergen, L.T.: Ask-Elle: an Adaptable Programming Tutor for Haskell Giving Automated Feedback. International Journal of Artificial Intelligence in Education **27** (02 2016)
8. Girard, J.Y.: The system F of variable types, fifteen years later. Theoretical Computer Science **45**, 159–192 (1986)
9. Harrington, B., Peng, S., Jin, X., Khan, M.: Gender, Confidence, and Mark Prediction in CS Examinations. In: Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education. pp. 230–235. ITiCSE 2018, ACM (2018)
10. Heeren, B., Jeuring, J., Gerdes, A.: Specifying Rewrite Strategies for Interactive Exercises. Mathematics in Computer Science **3**, 349–370 (05 2010)
11. Heeren, B., Leijen, D., van IJzendoorn, A.: Helium, for learning Haskell. In: Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell. pp. 62–71. Haskell '03, ACM (2003)
12. Jones, S.L.P.: Compiling Haskell by program transformation: A report from the trenches. In: Nielson, H.R. (ed.) Programming Languages and Systems — ESOP '96. pp. 18–44. Springer (1996)
13. Kölling, M., Quig, B., Patterson, A., Rosenberg, J.: The BlueJ System and its Pedagogy. Computer Science Education **13** (12 2003)
14. Kumar, A.N.: 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. pp. 523–527. ACM (2008)
15. Li, H., Thompson, S., Reinke, C.: The Haskell Refactorer, HaRe, and its API. Electronic Notes in Theoretical Computer Science **141**(4), 29–34 (2005)
16. Lodder, J., Heeren, B., Jeuring, J., Neijenhuis, W.: Generation and Use of Hints and Feedback in a Hilbert-Style Axiomatic Proof Tutor. International Journal of Artificial Intelligence in Education **31**(1), 99–133 (2021)
17. McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y.B.D., Laxer, C., Thomas, L., Utting, I., Wilusz, T.: 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. pp. 125–180. ACM (2001)

18. Nienaltowski, M.H., Pedroni, M., Meyer, B.: Compiler Error Messages: What Can Help Novices? In: Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education. pp. 168–172. SIGCSE '08, ACM (2008)
19. Peyton Jones, S.: Haskell 98 language and libraries: the revised report. Cambridge University Press (2003)
20. Peyton Jones, S., Marlow, S.: Secrets of the Glasgow Haskell Compiler inliner. J. Funct. Program. **12**, 393–433 (07 2002)
21. Spieler, B., Oates-Indruchova, L., Slany, W.: Female Students in Computer Science Education: Understanding Stereotypes, Negative Impacts, and Positive Motivation. Journal of Women and Minorities in Science and Engineering **26**, 473–510 (2020)
22. Sulzmann, M., Chakravarty, M.M.T., Jones, S.P., Donnelly, K.: System f with type equality coercions. In: Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation. pp. 53–66. TLDI '07, Association for Computing Machinery, New York, NY, USA (2007). https://doi.org/10.1145/1190315.1190324, https://doi.org/10.1145/1190315.1190324
23. Tolmach, A., Chevalier, T., GHC Team: An External Representation for the GHC Core Language (For GHC 6.10) (July 2009), 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
24. VanLehn, K.: The Relative Effectiveness of Human Tutoring, Intelligent Tutoring Systems, and Other Tutoring Systems. Educational Psychologist **46**(4), 197–221 (2011)
25. Whittall, S.J., Prashandi, W.A.C., Himasha, G.L.S., De Silva, D.I., Suriyawansa, T.K.: CodeMage: Educational programming environment for beginners. In: 9th International Conference on Knowledge and Smart Technology (KST). pp. 311–316 (2017)