IMPROVED NORMALIZATION FOR ASK-ELLE THROUGH SEMANTICS-PRESERVING TRANSFORMATIONS

ADOLFO OCHAGAVÍA

Supervisors

prof. dr. J.T. Jeuring
dr. ir. A. Serrano Mena

June 30, 2018
Utrecht, The Netherlands

ABSTRACT

Ask-Elle is a tutor for learning the Haskell programming language. It supports development of Haskell programs by providing stepwise hints and verifying the correctness of the resulting programs. Ask-Elle can do this based only on a model solution, which minimizes the work to set up exercises. The main contribution of our work is improving Ask-Elle's normalization mechanism, used to compare two programs for equivalence. Normalization is at the core of Ask-Elle's ability to provide hints and verify the correctness of programs. Based on a dataset of student programs, we (1) identify cases in which Ask-Elle's normalization does not work as expected, (2) implement semantics-preserving transformations that alleviate the problems and (3) measure the improvements. The results are very positive, as we achieve substantial improvements without resorting to advanced analysis techniques.

CONTENTS

1 INTRODUCTION

Ask-Elle [5] is a programming tutor designed to help students learn the Haskell programming language. With Ask-Elle, teachers can specify exercises along with *model solutions* and students can solve them interactively. During the process, the system is able to check whether a student is on the right track and, in case they get stuck, it can provide relevant *hints*. Afterwards, it can check whether the provided solution is correct.

The exercises supported by Ask-Elle ask to implement functions. For instance, an exercise to teach basic concepts around lists and recursion could be to implement the `length` function.

One of the strengths of Ask-Elle is its ability to provide feedback and hints based only on a model solution. The teacher writes one or more solutions for the exercise and Ask-Elle does the rest. This is very convenient, because it minimizes the work to set up the exercises.

1.1 ASK-ELLE IN ACTION

To understand how Ask-Elle works, consider a hypothetical situation where a student follows Ask-Elle's hints until reaching a solution to the assignment. In our example, the task is to implement the function `double :: [Int] -> [Int]`, which multiplies each number in a list by two. We assume the exercise to have a single model solution, defined as `double = map (* 2)`.

Figure 1.1 shows a student's path to a solution. It starts with an empty program, which is refined step by step. *Refinement* here refers to filling a hole in such a way that the resulting program is closer to a model solution. Ask-Elle hints are nothing more than refinement steps that have been generated from the model solution.

1.2 DEVIATIONS FROM THE MODEL SOLUTION

A fundamental limitation of Ask-Elle is its inability to produce hints for programs that *deviate* from the model solution. In such cases, Ask-Elle returns an error message: *You have drifted from the strategy in such a way that we can not help you any more*. With such a message, the only way forward is to start over from the very beginning (an empty program, represented as `?`) or to restore a previous version of the program known to be accepted by Ask-Elle.

While the session from Figure 1.1 starts with the empty program, a student has the freedom to choose a different starting point. Consider,

```
-- Starting point (? represents a hole)
?

-- Apply hint: Introduce the function double
double = ?

-- Apply hint: Use the higher-order map function
double = map ?

-- Apply hint: Use the times operator (*)
double = map (* ?)

-- Apply hint: Introduce the integer 2
double = map (* 2)
```

Figure 1.1: Interactive Ask-Elle session

```
double [] = []
double (x:xs) = x * 2 : double xs
```

Figure 1.2: Recursive implementation of `double`

for instance, the case of a student who wants to implement a recursive version of `double`. Figure 1.2 shows a recursive implementation, which returns the same result as the model solution for all possible inputs (we call this semantic equivalence, see Section 2.2 for more details). Let us imagine that the student does not start with an empty program, but with a partial implementation of the function. What would an Ask-Elle session look like in this case? Figure 1.3 shows an expected yet unfortunate outcome: there are no hints available.

The main way to deal with this limitation is by defining multiple model solutions. For instance, adding a recursive model solution would fix the problem in this concrete example. Still, this is not a scalable approach, since it requires a teacher to foresee all paths a student might follow. Furthermore, there are just too many ways to drift from the model solution in terms of syntax.

```
-- Starting point
double [] = ?
double (x:xs) = ?

-- Ask-Elle's reply: You have drifted from the strategy in
-- such a way that we can not help you any more
```

Figure 1.3: Unsuccessful Ask-Elle session

1.3 RESEARCH QUESTIONS

A typical use case for Ask-Elle is to aid teaching in an introductory course on functional programming. In that context, we have observed that Ask-Elle is often unable to handle student programs, regardless of whether the program is semantically equivalent to the model solution (see Chapter 5). In practice, this means that Ask-Elle is in many cases failing to provide hints or to assess the correctness of a finished program.

Previous research by Gerdes et al [6, 5] shows that part of the problem lies in limitations of Ask-Elle's normalization procedure (see Chapter 2 for details on the inner workings of Ask-Elle). The authors suggest implementing additional semantics-preserving transformations as a possible solution to the problem. This gives rise to the following questions:

1. In which cases does normalization not work as expected?
2. How can we improve Ask-Elle's normalization procedure?

This thesis is structured as follows. Chapter 2 presents the necessary background knowledge about Ask-Elle's architecture. Chapter 3 refers to related work relevant to our research. Chapter 4 explains our research methodology. Chapter 5 identifies the main normalization issues and contributes solutions in most cases. Chapter 6 shows the measured results achieved through normalization improvements. Finally, Chapter 7 summarizes our research and offers future work perspectives.

2

BACKGROUND KNOWLEDGE

Before attempting to answer our research questions, it is necessary to explain the relevant parts of Ask-Elle's architecture. This chapter presents the concepts of *programming strategy*, *program equivalence* and *program unification*.

2.1 PROGRAMMING STRATEGIES

As mentioned in Chapter 1, an exercise in Ask-Elle is a request to implement a function that matches one of the model solutions. Under the hood, Ask-Elle generates a *programming strategy* that specifies how an exercise can be solved, as a series of steps that go from an empty program to one of the model solutions.

Alternatively, we can think of a strategy as a finite-state machine. The initial state is the empty program, the transitions are the refinement steps and the accepting states are the model solutions.

Consider, for example, the model solution to the problem presented in Section 1.1 (defined as `double = map (* 2)`). Figure 2.1 shows the generated strategy.

According to the finite-state machine perspective on strategies, the algorithm used by Ask-Elle to provide hints works in two steps:

1. Translate the student's program to a state in the machine;
2. Retrieve a list of all transitions available from that state and present them in a user-friendly fashion.

From these two steps, the second is a common operation on finite-state machines. The first one, however, is much more involved, as it requires unifying programs.

2.2 PROGRAM EQUIVALENCE

Before diving into the concept of program unification we need to refer to program equivalence, which comes in three flavors.

A basic form of program equivalence is *syntactic equivalence*, according to which two programs are considered equivalent if their abstract syntax trees are equal. In its simplicity, this kind of equivalence has clear limitations. For instance, it does not account for differences in variable names or in the structure of the code.

More advanced is *semantic equivalence*, according to which two programs are considered equivalent if their output is similar for all possible inputs. In the context of Ask-Elle, programs are functions, inputs

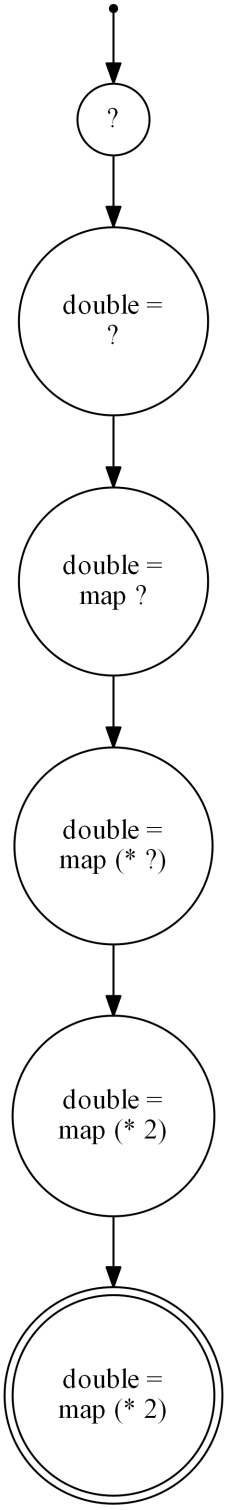


Figure 2.1: Example of an Ask-Elle strategy

are function parameters and the input domain consists of all possible values of the parameter types. In this case, differences in variable names are ignored, as well as differences in the structure of the code, as long as they have no influence in the output of the program. For Haskell, this also means that it does not take laziness into account, unless it has an impact on the output.

Finally, semantic equivalence *up to preconditions* relaxes previous definition. Instead of requiring that two programs produce the same output for all possible inputs, this kind of equivalence considers only inputs that satisfy the preconditions of the program. Note that we assume the same preconditions to hold for both programs.

2.3 PROGRAM UNIFICATION

The process of comparing two programs for semantic equivalence up to holes is called program unification. This is what Ask-Elle does when comparing a student program to a model solution.

Since the problem of program unification is undecidable in the case of a complex language such as Haskell (see Section 3.2), Ask-Elle's unification procedure has the following possible outcomes:

1. The programs are semantically equivalent;
2. One program is an incomplete version of the other;
3. Equivalence cannot be concluded.

At the core of Ask-Elle's unification mechanism is the idea of normalization. Before comparing the programs, they go through a series of semantics-preserving transformations that result in a normal form. This way, comparing the programs becomes as simple as syntactically unifying the resulting normal forms.

Consider, for instance, the `double` function we mentioned before. Figure 2.2 shows a model solution, a possible student answer and the normalized version of both programs. This is an example of how a small syntactical difference, an unnecessary anonymous function, is ignored thanks to semantics-preserving transformations.

```
-- Model solution
double = map (* 2)

-- Student answer
double = map (\x -> 2 * x)

-- Normalized version (similar for both)
double = map ((*) 2)
```

Figure 2.2: Example of unification by normalization

3 RELATED WORK

As mentioned in Chapter 2, the key characteristic of Ask-Elle is its ability to give stepwise hints and to check whether two programs are semantically equivalent. Here we provide an overview of the research related to these topics.

3.1 RELATED WORK IN PROGRAMMING TUTORS

We are interested in proving whether two programs are semantically equivalent. This involves obtaining information from the programs to check whether the conditions for equality hold. Gathering information about a program is the subject of program analysis, which is used pervasively in automated assessment tools [10].

Since program analysis can be classified as static or dynamic, the same holds for the feedback tools that rely on it. While in the past there used to be a clear distinction between static and dynamic tools [1], it is nowadays less clear, as many tools are combining both approaches [10]. For instance, Codewebs [14] and the tool presented by Huang et al. [9] are based on a combination of testing and static analysis. Another example is OverCode [7], which collects data from the execution of Python programs (dynamic) and combines it with data derived from the source code (static). Ask-Elle also uses a mixed approach, since it first attempts comparison by normalization (static) and resorts to property-based testing (dynamic) whenever the former is insufficient [6, 5].

Ask-Elle's static analysis is geared towards normalization of programs. Its purpose is to identify patterns in the code that can be rewritten in a canonical form. Currently, transformations are simple enough that little analysis is needed. For most transformations, Ask-Elle makes use of term-rewriting by equational reasoning [2], instead of true static analysis. While advanced static analysis has the potential of achieving better results, equational reasoning works well in most cases and keeps the implementation simple.

Relevant research on automatic assessment tools that rely on program normalization include Xu and Chee's work [19], who apply this method to compare Smalltalk programs. Also interesting is the work of Wang et al. [18], who implement a normalization-based tool to assess C programs. A more modern approach is given by the ITAP tutoring system [16], targeting Python programs.

From the tools mentioned, the most interesting for our purposes is ITAP, because it also supports style-based transformations tailored

to beginners. This is exactly one of the features we want Ask-Elle to support. Besides this, ITAP supports reconstructing the original AST from its normalized form to produce localized feedback, including suggestions to refactor the code.

Still, none of the tutors presented above offer stepwise feedback, except for Ask-Elle. In fact, a recent survey of 69 tools for learning programming by Keuning et al. [10] identifies only 10 that give this kind of feedback. From these, only Ask-Elle, the Prolog Tutor [8] and ADAPT [4] (also targeting Prolog) use program normalization. It is unclear whether the approach taken by the Prolog Tutor and ADAPT can be a source of inspiration for Ask-Elle, given the big differences between Haskell and Prolog.

3.2 RELATED WORK IN PROGRAM UNIFICATION

Program unification consists of comparing two programs to determine whether they are equivalent under a particular definition of equivalence. In the context of Ask-Elle, we are interested in semantic equivalence of Haskell programs, which is undecidable.

3.2.1 *Unification in the simply-typed lambda calculus*

The problem of unifying simply-typed lambda terms, of which Haskell is an extension, is known in the literature as *higher-order unification*. While undecidable in general [3], higher-order unification becomes decidable by imposing certain restrictions on the shape of the lambda terms. A well-known example is Miller's pattern unification [13], which achieves decidability by restricting beta-reduction.

3.2.2 *Unification in Haskell*

The Haskell language can be seen as a user-friendly version of the lambda calculus, with a series of extensions and syntactic sugar to facilitate programming. Since unification of simply-typed lambda terms is undecidable [3], the problem of unifying Haskell terms is undecidable as well.

To visualize how Haskell is an extension of the simply-typed lambda calculus we can look at its underlying hierarchy of languages. The simply-typed lambda calculus is at the base, followed by System F [15], System F_ω [15], System FC [17] and Haskell itself. Each language in this hierarchy is conceptually an extension of the previous one. Transitively, this means that Haskell is an extension of the simply-typed lambda calculus.

3.2.3 *Decidable alternatives*

Given undecidability of unification, programming tutors reach to decidable alternatives. Most of them give up on true semantic equivalence and go for relaxed definitions of equivalence instead [10]. A clear example is Wang's grading tool for C programs [18]. After doing normalization, it compares a program to the model solution based on metrics like program size and control flow statements used.

The main drawback of relaxed notions of equivalence is that they do not guarantee semantic equivalence. In the case of Ask-Elle, this approach would introduce the risk of giving incorrect hints to the users. Recall that program unification is at the core of Ask-Elle's ability to identify the next step in filling the holes of an incomplete program (see Chapter 2 for details). Clearly, resorting to a relaxed notion of equivalence is not a satisfying solution for a programming tutor with its main focus on stepwise feedback.

The alternatives that restrict the language, such as Miller's [13], are too impractical to consider, as they would require rejecting valid Haskell programs. Futhermore, most research on enabling higher-order unification by restricting the language is done at the level of the lambda calculus, while Haskell provides much higher-level constructs. The interaction between those restrictions and Haskell's features is uncharted territory, out of the scope of this research.

Another alternative is to attempt normalization-based unification. This approach turns the unification problem into a normalization problem so it becomes decidable. Under this unification scheme, two programs are considered equivalent if they both normalize to the same normal form. This notion of equivalence guarantees semantic equivalence as well, though it is incomplete, meaning that for some programs it is impossible to know whether they are equivalent. This is the approach followed by Ask-Elle [6].

4

METHODOLOGY

In Section 1.3, we set out to find the cases in which Ask-Elle's normalization does not work as expected. Based on this knowledge, we aim to improve the normalization procedure so it can handle a wider range of programs. We tackle the problem by iterating through the following steps:

1. Measure the effectiveness of Ask-Elle's normalization, relative to a set of student programs;
2. Analyze the set of student programs to reveal normalization shortcomings;
3. Enhance Ask-Elle's normalization to resolve the discovered shortcomings.

The analysis phase attempts to answer our first research question, while the enhancement phase tackles the second one. The measurement step provides a point of reference.

4.1 MEASURE

To quantify the initial state of Ask-Elle's normalization and our improvements, we define a series of measurements to be taken on a particular dataset.

4.1.1 *Dataset*

Our method requires analyzing sets of programs known to be semantically equivalent. To this purpose, we analyze the correct solutions to *Assignment I* of the functional programming course at Universiteit Utrecht in the year 2017/2018.

From the students that participated in the course, there are 111 who submitted a correct solution to the assignment and gave consent for it to be used for research purposes. Students were allowed to submit multiple solutions, but we considered only the final submissions in our research, as each of them is supposed to be the best solution the student was able to write.

In the assignment, the students are asked to implement 8 functions of growing complexity [11]. This translates to 8 Ask-Elle exercises, each one with 111 student solutions. Coincidentally, the functions in the assignment document are labeled as Exercise 1, Exercise 2, etc, up to Exercise 8.

Correctness of the answers was verified through a test suite, which turned out to allow some incorrect programs and thus required minimal clean-up of the data. Table 4.1 shows the amount of correct programs for each exercise, after filtering out those that were wrongly classified as correct.

Exercise number	Correct submissions
1	110
2	111
3	103
4	108
5	109
6	101
7	99
8	100

Table 4.1: Correct submissions per exercise

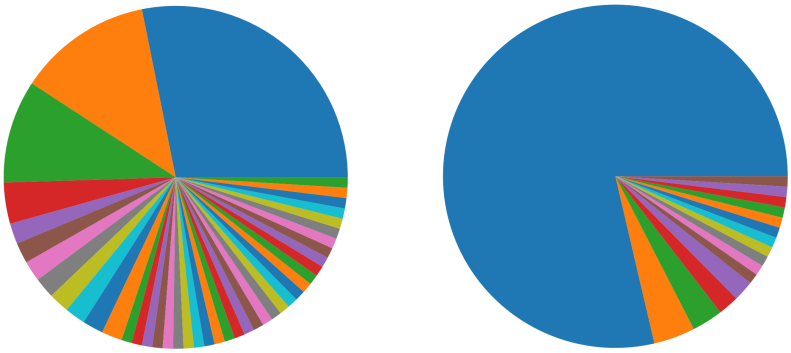
4.1.2 *Measurements*

Measurements are done on a per-exercise basis. As a first step, we classify the student submissions into clusters. A *cluster* is *a set of programs that share the same normal form, under a given normalization procedure*. This means that the programs in a cluster are seen as semantically-equivalent by Ask-Elle.

Based on the concept of clusters, we can define the *effectiveness* of normalization as *the amount of clusters and their size, after normalizing the student submissions*. As an example, Figure 4.1 compares the effectiveness for Exercise 3 before and after our research. Without our improvements, there are 51 clusters, where the biggest one contains 29 programs. After our improvements, there are 16 clusters, where the biggest contains 81 programs.

We say that a given normalization procedure is better or *more effective* than a second one when *it results in a lower amount of clusters for the same exercise*. In the case of Exercise 3, we can say that our improved normalization scheme is more effective than Ask-Elle's original one.

Informally, we can also visualize normalization effectiveness by looking at the percentage of student programs that are recognized by a given amount of model solutions. Again in Exercise 3, Ask-Elle used to recognize 54% of the submissions based on 4 model solutions. Now, Ask-Elle recognizes 85% of them based on 3 model solutions.



Before After

Figure 4.1: Effectiveness for Exercise 3 before and after improvements

4.2 ANALYZE

Identifying shortcomings in the normalization procedure becomes straightforward after classifying the programs into clusters. Informally, our approach consists of repeatedly following the steps below:

1. Pick one of the biggest clusters;
2. Pick a cluster that is smaller than the current one;
3. Compare the normal forms of both clusters;
4. Identify patterns that should have the same normal form, but do not;

As an example of a limitation we discovered, consider Exercise 1, which consists of implementing the function `parseTable`. Below we compare the normal forms of two clusters:

Cluster A (73 programs)	Cluster B (13 programs)

`parseTable = map words`	`parseTable =`
	`  (\x1 -> case x1 of`
	`  x2 -> map words x2`
	`  )`

Figure 4.2: Comparing two clusters of Exercise 1

We can see that both normal forms have the same semantics but are still syntactically different. In this case, it seems reasonable to think of a semantics-preserving tranformation that turns the right-hand side of Cluster B into `map words`. Figuring out which exact transformation we need is the role of the *enhance* step, explained in the next section.

While the normalization problem identified in the previous example is fairly low-level, we identify higher-level problems too. For instance, Section 5.1 explains the interaction between function preconditions and normalization.

4.3 ENHANCE

The enhance step bridges the gap between discovering shortcomings and improving Ask-Elle's normalization. Besides adding new transformations to the normalization pipeline, it needs to ensure they are semantics-preserving.

4.3.1 *Adding a new transformation*

Let us consider the normal forms from Figure 4.2. We could say that Cluster B is stuck somewhere in the normalization process, instead of arriving at `map words` as we would expect.

Ask-Elle has an eta-reduction transformation, which could simplify `\x1 -> map words x1` to `map words`. However, it fails to work in the case of Cluster B because there is a `case` expression.

Ask-Elle also has an inlining transformation, which replaces a variable usage by its definition. In the case of Cluster B, it seems sensible that `x2` could be replaced by `x1`, thereby getting rid of the `case`. However, given that `case` is a very expressive Haskell construct, the inlining transformation does not support it. Instead, it only supports `let` bindings.

One possible solution is to add a normalization step that, for any `case` expression consisting of a single alternative with only a var pattern, rewrites it in terms of `let`. In the case of Cluster B, this transformation would allow the inliner to simplify the code further in a later pass, which opens the door to eta reduction.

Before	After
`case e of` `  x -> e'`	`let x = e` `in e'`

Figure 4.3: Transforming a case into a let

4.3.2 *Soundness*

We say that a transformation is sound when it is semantics-preserving. That is, for any program, the transformed program is semantically equivalent to the original one.

To increase our confidence in the soundness of the new transformations, we resort to formal proofs and property-based testing.

FORMAL PROOFS Throughout our research, we added around 60 transformations to Ask-Elle. Formalizing and proving that all of them are semantics-preserving would be a research project of its own. Still, we formalized and proved the soundness property for 31 of them in Coq.

When choosing which transformations to prove we discarded trivial ones (e.g. rewrite rules that are true by definition) and others that were too complex (e.g. anything involving scoped variables, like inlining or dead code removal). Instead, we focused on transformations that can be expressed as rewrite rules in Coq's type system.

As an example, consider a hypothetical cluster of programs in Exercise 1 that has the following normal form:

```
parseTable =
    (\xs -> case xs of
        [] -> []
        ys -> map words ys
    )
```

Figure 4.4: Useless case matching before calling map.

We can prove that removing the `case` expression preserves the semantics of the program, as shown in the *empty_base_case* theorem on Figure 4.5. The proof says that calling `map` inside a `case`, as in Figure 4.4, has the same semantics as calling it directly. The proof is surprisingly simple using Coq's built-in tactics, as is the case for most of the proofs we constructed. We distinguish two possible cases: `xs` is empty or it is not. In both cases, beta-reduction results in an equality that is true by reflexivity.

PROPERTY-BASED TESTING The original assignment had a basic test suite that we discovered to be insufficient, since some incorrect student solutions were not flagged as such. Therefore we developed our own test suite, specifying properties that each exercise should satisfy.

By feeding the normalized programs to our test suite, we were able to verify that they still satisfy the properties required by the assignment. This proved to be an effective approach, as it caught a few bugs in the implementation of some transformations.

```
Definition bad_style_map
                {T U} (f : T -> U) (xs : list T)
                : list U :=
    match xs with
    | [] => []
    | ys => map f ys
    end.

Theorem empty_base_case : forall {T U} (f : T -> U) xs,
    bad_style_map f xs = map f xs.
Proof.
    intros. (* Introduce variables f and xs *)
    destruct xs. (* Case split on xs *)
        auto. (* Trivially true when xs = [] *)
        auto. (* Trivially true when xs = y :: ys *)
Qed.
```

Figure 4.5: Proving the soundness of a transformation.

5 ANALYSIS AND NEW TRANSFORMATIONS

We observe multiple issues that cause problems to normalization. First of all, many students struggle with the concept of preconditions. Additionally, we identify several domain-specific issues involving lists, booleans, the `Maybe` type and patterns. When possible, we add new transformations to alleviate those issues.

5.1 PRECONDITIONS

Throughout the assignment, some functions come with preconditions to simplify their implementation. For instance, knowing that a list will never be empty means that a student no longer has to write additional code to handle that case. Contrary to the expectations, preconditions caused confusion among students, instead of leading to simpler code.

5.1.1 *Explicit and implicit preconditions*

In the assignment document, Exercise 3 specifies that the function `printField :: Int -> String -> String` should satisfy the property $\forall n\ s.\ n \geqslant \mathtt{length}\ s \Rightarrow \mathtt{length}\ (\mathtt{printField}\ n\ s) \equiv n$. Here, the left side of the implication is the precondition of the function. Therefore, a good solution is expected to assume that $n \geqslant \mathtt{length}\ s$, without checking whether it holds, as illustrated by Figure 5.1.

```
printField n s
    | all isDigit s = padding ++ s
    | otherwise = s ++ padding
    where padding = replicate (n - length s) ' '
```

Figure 5.1: Exercise 3 - Model solution

Besides Exercise 3, no other exercise in the assignment mentions properties or preconditions explicitly, be it in a formal or informal way. However, it is possible to deduce many preconditions from the context, as summarized by Table 5.1. Note that, at this point, we are not interested in the exercises themselves, only in the kinds of preconditions they have. Table 5.1 shows that the preconditions are well-defined, which means that Ask-Elle could potentially take them into account when normalizing programs.

Exercise	Precondition
1	The first argument is a non-empty list
2	The first argument is a non-empty list
4	The first argument is a non-empty list
5	The first argument is a well-formed table
6	The first argument is a well-formed table
7	The third argument is a well-formed table
8	The second argument is a well-formed table

Table 5.1: Implicit preconditions

5.1.2 *Preconditions and semantics*

Preconditions allow functions with different semantics to be considered correct. In case the precondition does not hold, the program is allowed to do anything. Therefore, it is possible to have two programs that behave in the same way when the precondition holds, but do different things when it does not hold.

Notice that writing a submission to an exercise always involves specifying what to do when the precondition does not hold, even when there is no explicit checking for the precondition. Consider the solution to Exercise 3 that we present in Figure 5.1. If `n < length s`, then `n - length s < 0`, then `replicate (n - length s) ' ' = []`. Therefore, this particular implementation of `printField` will return `s` whenever the precondition does not hold.

5.1.3 *Precondition checking*

Many students add explicit checks to handle the case in which the precondition does not hold. Since the assignment does not specify what the function should return in such a case, students are allowed to write anything that seems sensible to them.

DIFFERING SEMANTICS Figure 5.2 shows a submission to Exercise 3 where the student adds a guard checking for `n < length s`. On the right-hand side of the guard, we observe that the student truncates the string.

Comparing this implementation to the model solution in Figure 5.1, we can see that the semantics have changed. The model solution returns `s` when the precondition does not hold, while this implementation returns `take n s`.

This is a case in which checking for the precondition leads to a program with different semantics when compared to the model so-

```
printField n s
    | n < length s  = take n s
    | all isDigit s = padding ++ s
    | otherwise     = s ++ padding
    where padding = replicate (n - length s) ' '
```

Figure 5.2: Exercise 3 - Explicit precondition checking; differing semantics

```
printField n s
    | n < length s  = s
    | all isDigit s = padding ++ s
    | otherwise     = s ++ padding
    where padding = replicate (n - length s) ' '
```

Figure 5.3: Exercise 3 - Explicit precondition checking; similar semantics

lution. In the context of Ask-Elle, this means that normalizing both programs results in different normal forms.

SIMILAR SEMANTICS In some cases, checking for the precondition did not result in different semantics relative to the model solution. Figure 5.3 shows a function that explicitly checks for the precondition and returns `s` when the precondition does not hold.

In this case, the semantics of the program are similar to those of the model solution, but normalizing both still yields different normal forms.

ADDITIONAL COMPLICATIONS Since a precondition is nothing more than a logical expression, it can be written in multiple ways. This offers even more challenges, because a precondition that is checked in the code could appear in any form as long as its meaning does not change. In fact, submissions to Exercise 3 sometimes check the precondition using its negation (`n < length s`), sometimes split the check in three guards (`n > length s`, `n == length s` or `otherwise`) and sometimes do the check in a completely different way.

Additionally, it is possible to implicitly check the precondition without resorting to `if` expressions or guards. For instance, in Figure 5.4 a student calls the `max` function to ensure that replicate never receives a negative argument. This is unnecessary, since it is guaranteed by the precondition of the function.

5.1.4 *Normalization up to preconditions*

The ideal solution to the problems outlined above is to integrate the preconditions in our reasoning of semantic equivalence. If we assume that the precondition always holds, then we can ignore the code that

```
printField n s
    | all isDigit s     = r ++ s
    | otherwise         = s ++ r
    where
        r = replicate a ' '
        a = max 0 (n - length s)
```

Figure 5.4: Exercise 3 - Implicit precondition checking

Before

```
parseTable [] = error "empty list in parseTable"
parseTable xs = map words xs
```

After

```
parseTable xs = map words xs
```

Figure 5.5: Exercise 1 - Normalization up to preconditions

is meant to handle the case when the precondition does not hold. More specifically, we can treat such code as dead, because we know it will never be executed.

The main challenge with this approach is that it requires identifying which parts of the code are dead according to the precondition. In the general case, this amounts to solving the halting problem[1]. Therefore, we limit ourselves to a heuristic that removes pattern matching on the empty list when we know it should be non-empty. This is illustrated by Figure 5.5.

5.2 LISTS

The assignment expects students to manipulate lists by combining list-specific functions. Therefore it is not surprising that many normalization problems come from the domain of lists.

5.2.1 *Function abstraction heuristic*

Many students fail to use higher-order functions and resort to recursion instead, occasionally reimplementing a higher-order function

1 Detecting dead code, even without taking preconditions into account, amounts to solving the halting problem. We can prove this by contradiction. Consider a program P that can detect dead code in the general case and a program A that we would like to check for termination. To this purpose, we add a line of code at the end of A. If P says the code is dead, we know that A does not terminate. If P says the code is live, we know that A does terminate.

```
parseTable = map words
```

Figure 5.6: Exercise 1 - Model solution

Student submission

```
parseTable [] = []
parseTable (x:xs) = words x : parseTable xs
```

Normalized version (before our improvements)

```
parseTable = fix (\rec xs -> case xs of
        [] -> []
        y : ys -> words y : rec ys
    )
```

Figure 5.7: Exercise 1 - Reimplementation of map

without realizing. We identified reimplementations of `map`, `filter` and `zip`.

As an example of a function where the student has reimplemented `map`, let us consider Figure 5.7. It is clear that the normal form does not match the model solution in Figure 5.6.

To tackle this problem, we add a heuristic that is able to replace recursive functions by their standard library counterparts. The heuristic detects reimplementations of `map`, `filter` and `zip`.

With this heuristic in place, the function from Figure 5.7 is normalized to the same normal form as the model solution.

5.2.2 *Grouping recursive submissions in a single cluster*

The function abstraction heuristic works well for simple functions such as `map`, `filter` and `zip`. However, it is not clear whether more complex cases like `foldr` can be abstracted by a general-purpose heuristic. In such cases, we give up on trying to remove recursion and hope that regular normalization succeeds in grouping recursive submissions in a single cluster.

An example of this is provided by Exercise 2. Figure 5.8 shows two model solutions, representing the cluster of submissions using `foldr` and the cluster using recursion.

This approach works great when the recursive submissions are somewhat similar to each other. In fact, the biggest cluster in Exercise 2 after our improvements consists of recursive submissions and accounts for 35% of the total submissions.

Still, there are always creative submissions that cannot be normalized to the same cluster. Two common problems here are caused by

Model solution using foldr

```
printLine = foldr (\x xs -> '+' : replicate x '-' ++ xs) "+"
```

Recursive model solution

```
printLine []     = "+"
printLine (x:xs) = "+" ++ replicate x '-' ++ printLine xs
```

Figure 5.8: Exercise 2 - Model solutions

preconditions and by using guards instead of pattern matching. The submission to Exercise 2 in Figure 5.9 features both:

PRECONDITIONS As discussed in Section 5.1.2, preconditions in exercises allow functions with different semantics to be considered correct. Semantic differences are introduced because of variations in the way the exceptional case is handled.

In Exercise 2, the precondition of `printLine` is that the list cannot be empty. Because of the this, we see a variation between the solution in Figure 5.8 and Figure 5.9. The former returns `"+"` when the list is empty, while the latter crashes. This means that no amount of semantics-preserving transformations can bring both programs to the same normal form.

GUARDS AND PATTERN MATCHING Using guards instead of pattern matching does not alter the semantics of the program. This means that it is possible to implement a semantics-preserving transformation to replace the guards by pattern matching.

In the example, the student checks whether the list is empty using the `length` function. This check could clearly be replaced by a `case` expression that matches on the list, where the empty case jumps to the right-hand side of the first guard and the other case jumps to the second.

We do not implement such a transformation, however, because it seems too ad-hoc. It works only for lists and only when the condition is `length xs == 0`. Ideally, we would like to have a generalized version, working on other data types as well (such as `Maybe`) and able to handle a wider range of conditions. Because of time constraints and the low number of submissions that would benefit from such a transformation, we leave it as a candidate for future work.

5.2.3 *Desugaring*

Some of the exercises involve using list and string literals. Since the empty string can be expressed as `""` or `[]`, this results in different

```
printLine (w:ws)
    | (length ws == 0) = "+" ++ line ++ "+"
    | otherwise = "+" ++ line ++ printLine ws
    where line = concat (replicate w "-")
```

Figure 5.9: Exercise 2 - Recursive submission with multiple issues

normal forms in otherwise equivalent programs. We also observe variations in the way singleton lists and strings are represented by students, which lead again to different normal forms.

To resolve this issue, we specify a single normal form for list and string literals. We achieve this through the following transformations:

From	To
String literal	List of characters
List literal	Sequence of cons

Besides the issue of list and string literals, we observe that some students use list comprehensions instead of higher-order functions. This again results in different normal forms in otherwise equivalent programs. To tackle this problem, we add a normalization step that desugars list comprehensions into list literals and higher-order functions. To this purpose, we use the algorithm described in Section 3.11 of the Haskell Report [12].

5.2.4 *List laws*

In many cases, the differences between two semantically-equivalent normal forms can be resolved by a simple rewrite rule. Based on the student submissions, we identify a series of laws and implement them as rewrite rules in Ask-Elle's normalization procedure.

Table 5.2 shows the new transformations. Most of them are aimed to simplify the expressions on the left, but some of them are needed to impose an order when multiple representations are possible.

Besides these laws, we also identify slightly more complex ones that require reasoning at the scope level. We describe them below.

Simplify concatMap and mapMaybe

A pattern we observe in some student submissions is using `concatMap` when `map` would be sufficient. The same happens with the `mapMaybe` function. We add the transformations from Table 5.3 to normalize both into `map`. Notice that `e` can be any Haskell expression and that it may contain references to `x`.

From	To
`xs ++ []`	`xs`
`map f . map g`	`map (f . g)`
`concatMap f . map g`	`concatMap (f . g)`
`concat (replicate x [e])`	`replicate x e`
`take n (cycle [e])`	`replicate x e`
`intercalate []`	`concat`
`foldr (++) []`	`concat`
`foldr (:)`	`flip (++)`
`concat . map`	`concatMap`
`map id`	`id`
`concatMap (flip (:) [])`	`id`
`(!! 0)`	`head`
`(xs ++ ys) ++ zs`	`xs ++ (ys ++ zs)`
`transpose . map (map f)`	`map (map f) . transpose`

Table 5.2: List laws added to the normalization procedure

From	To
`concatMap (\x -> [e])`	`map (\x -> e)`
`mapMaybe (\x -> Just e)`	`map (\x -> e)`

Table 5.3: Simplifying `concatMap` and `mapMaybe`

From	To
`concatMap (\x -> if cond x then [e] else [] )`	`filter cond`

Figure 5.10: Turn `concatMap` into `filter`

```
printLine xs = "+" ++ intercalate "+" lines  ++ "+"
    where
       columns = length xs
       lines = [replicate (xs !! index) '-'
                | index <- [0..(columns-1)]]
```

Figure 5.11: Exercise 2 - List indexing

Turn concatMap into filter

Another pattern that caused problems with normalization is reimplementing `filter` in terms of `concatMap`. The transformation in Figure 5.10 solves the problem. Again, `e` can be any Haskell expression and it may contain references to `x`.

5.2.5 *List indexing*

A particularly problematic pattern we observe in a few student submissions is list indexing. While inoffensive on its own, list indexing becomes a normalization nightmare when used to write imperative-like code. Consider, for instance, the submission to Exercise 2 in Figure 5.11. Even though the semantics are the same as a properly functional implementation (as in Figure 5.8), the enormous syntactic differences make it very difficult for both to reach the same normal form.

In the particular example we are considering, we can see that the list comprehension is reimplementing `map`. While it would be possible to write a transformation that applies to this particular case, it seems too ad-hoc. Also, a generalized version of this transformation would benefit only a few submissions. Therefore we leave this problem as future work.

5.3 BOOLEANS

Given the crucial role of booleans in the control flow of a program, normalization problems on this domain have a far reaching effect.

Before

```
printField n s
    | all isDigit s = padding ++ s
    | otherwise     = s ++ padding
    where padding = replicate (n - length s) ' '
```

After

```
printField n s = if all isDigit s then padding ++ s
                                   else s ++ padding
    where padding = replicate (n - length s) ' '
```

Figure 5.12: Total guards are rewritten as a top-level `if`

```
printField n s
    | all isDigit s       = padding ++ s
    | not (all isDigit s) = s ++ padding
    where padding = replicate (n - length s) ' '
```

Figure 5.13: Total guards using `not`

While there are less issues compared to lists, we still identify and address many of them.

5.3.1 *Guard rewriting*

One of Ask-Elle's normalization steps rewrites guards as `if` expressions. This is only possible if the guards are total, as illustrated by Figure 5.12.

Ask-Elle knows that guards are total if the last one is `True` or `otherwise`. While this heuristic works well in many cases, it fails when a program uses total guards expressed in terms of a condition and its negation (see Figure 5.13). For this reason, we enhance Ask-Elle's guard rewriting to support this new way of expressing total guards.

5.3.2 *Boolean laws*

Similar to lists, there are many cases in which the differences between two semantically-equivalent normal forms can be resolved by a simple rewrite rule. Table 5.4 summarizes the laws we implement as rewrite rules in Ask-Elle's normalization procedure.

From	To
`(==) True`	`id`
`(&&) True`	`id`
`(==) False`	`not`
`foldr (&&) True`	`and`
`all id`	`and`
`and . map f`	`all f`
`if cond then True else False`	`cond`
`if not cond then y else x`	`if cond then x else y`

Table 5.4: Boolean laws added to the normalization procedure

From	To
`notElem`	`not . elem`
`isNothing`	`not . isJust`

Table 5.5: Function negation

5.3.3 *Function negation*

Some Haskell functions are equivalent to the negation of other functions. For instance, `isNothing` is equivalent to `not . isJust`. This is important knowledge, as some of the transformations described above operate on expressions that involve the `not` function. From this perspective, expressing a function in terms of `not` increases the chances that a program can be normalized further. Table 5.5 shows new rewriting transformations directed towards this goal.

As an example of how this helps drive normalization further, consider Figure 5.14. The original program, based on `isNothing`, ends up in the same normal form as a similar program based on `isJust`.

```
if isNothing x then y else z      rewrite in terms of not
if not (isJust x) then y else z   switch if branches
if isJust x then z else y
```

Figure 5.14: Interaction between function negation and boolean laws

From	To
`(==) Nothing`	`isNothing`
`(/=) Nothing`	`isJust`
`maybeToList x == []`	`isNothing x`
`maybeToList x /= []`	`isJust x`
`maybeToList x !! 0`	`fromJust x`

`case m of`
`  Nothing -> d`
`  Just x  -> x`
	`fromMaybe d m`

Table 5.6: `Maybe` laws

`case m of`
`  Nothing -> Nothing`
`  Just x -> e`
	`fmap f m`

Figure 5.15: Abstracting `fmap` for `Maybe`

5.4 MAYBE

Exercises 7 and 8 rely on the `Maybe` type to handle edge cases. Students are expected to use functions from the standard library to manipulate `Maybe` values. This results in multiple issues.

5.4.1 *Function abstraction through rewrite rules*

One of the main problems we observe is caused by function reimplementation. Many students write their own version of `fromMaybe`, `fromJust`, `isJust`, `isNothing` and `fmap`.

In most cases, functions can be abstracted through a simple rewrite rule, as illustrated by Table 5.6. We implement this in a similar way to list and boolean laws (tables 5.2 and 5.2 respectively).

The case of `fmap` is a bit more complex, as shown in Figure 5.15. On the left-hand side, `e` is an arbitrary expression that may or may not use the `x` variable. On the right-hand side, `f` is a lambda with a single parameter, where the body is the `e` expression with all occurrences of `x` replaced by its parameter. Figure 5.16 offers an example.

```
case m of
    Nothing -> Nothing
    Just x -> x + 1
```

```
fmap (\a -> a + 1) m
```

Figure 5.16: Example of abstracting `fmap` for `Maybe`

```
maybe d f m = case m of
    Nothing -> d
    Just x -> f x
```

Figure 5.17: Definition of `maybe`

5.4.2 *Unfolding the maybe function*

Some student submissions use the `maybe` function instead of `case` when manipulating `Maybe` values. While abstracting `maybe` from `case` expressions is possible, unfolding `maybe` yields even fewer clusters when normalizing. Therefore we add an unfolding transformation to Ask-Elle's normalization, according to the definition of the function in 5.17.

The improvement caused by this transformation derives from its interaction with normalization passes related to patterns and `case` expressions. These transformations are described in Section 5.5.

5.4.3 *The isJust / fromJust combination*

Another normalization problem is caused by the combination of `isJust` and `fromJust`. With this pattern, a student first checks whether a value is `Just` or `Nothing`. In the first case, the program extracts the value using `fromJust` and passes the result to an expression. In the latter case, a different branch of the code is executed. Figure 5.18 illustrates the problem and proposes an idiomatic alternative.

Ideally, the normalization procedure should be able to rewrite the `isJust` pattern as a `case` expression. Such a transformation would apply to expressions in the form `if isJust m then e else e'`, replacing all occurrences of `fromJust m` in `e` by the unwrapped value.

isJust / fromJust	Idiomatic alternative

`if isJust m`
`  then f (fromJust m)`
`  else d`
	`case m of`
	`  Nothing -> d`
	`  Just x -> f x`

Figure 5.18: The `isJust` / `fromJust` combination

```
let x = fromMaybe (-1) m
in if x /= (-1) then f x else d
```

Figure 5.19: Disguised `isJust` / `fromJust` combination

Because of its interaction with other transformations, adding this normalization step is more complex than it seems. Considering its expected low impact, we leave it as future work.

5.4.4 *Disguised isJust / fromJust combination*

We observe also a variation of the `isJust` / `fromJust` pattern, where students use `fromMaybe` with a magic default value, which is later checked by an if statement. In case the output of `fromMaybe` turns out to be the magic value, then the original `Maybe` object was `Nothing`. Otherwise, it was `Just`. The resulting code is presented in Figure 5.19.

In this case, a normalization step that transforms the expression into a `case` is much more complex, for two reasons:

- The pattern assumes that the value bound to `m` is never `Just (-1)`;
- Checking whether the magic value was returned can be done in multiple ways (e.g. students use `(==)`, `(/=)` and even `(>)`).

The first point implies that a semantics-preserving transformation must be able to prove that the value bound to `m` is never `Just (-1)`. The second point makes this problem even worse. For instance, if a student writes `x > (-1)` as the `if` condition, the normalizer would also need to prove that `m` can never contain any number lower than `-1`. Inferring this kind of knowledge requires more advanced program analysis and is beyond the scope of this research.

5.5 PATTERNS

Pattern matching is at the core of the Haskell language. Since patterns are very expressive, there are many different ways to write semantically equivalent programs. Again, this is a source of problems from a normalization perspective.

5.5.1 *Pattern simplification*

A straightforward simplification is to remove wildcard patterns bound to a name. That is, a pattern in the form `x@_` is simplified to `x`. Since a name matches the same values as a wildcard, this transformation is sound.

Another useful simplification is to remove nested *as* patterns. For instance, a pattern like `xs@ys@(z:zs)` could be simplified to `xs@(z:zs)`,

From	To
`head xs`	`let (x : _) = xs in x`
`tail xs`	`let (_ : ys) = xs in ys`
`fst x`	`let (a, _) = x in a`
`snd x`	`let (_, b) = x in b`
`fromJust m`	`let (Just x) = m in x`

Table 5.7: Unfolding functions that pattern match

Nested

```
case a of
    [] -> []
    xs -> let (y : _) = xs
          in e
```

Collapsed

```
case a of
    [] -> []
    xs@(y : _) -> e
```

Figure 5.20: Nested pattern matching

thereby removing the binding to `ys`. Note that, in this case, additional care must be taken to rename all usages of `ys` to `xs`.

5.5.2 *Unfold functions that pattern match*

Another source of syntactic differences are functions that pattern match on their parameters, with an undefined case when the pattern does not match the definition. This allows writing the same program by calling such functions or by using `case` or `let` inline. Table 5.7 shows the transformations we add to improve normalization in these cases.

5.5.3 *Nested patterns and pattern lifting*

Nested pattern matching is another source of syntactic differences that cause normalization problems. Sometimes it originates from student submissions and in other cases it is produced by other normalization steps (e.g. the unfolding transformation mentioned in Section 5.5.2).

Ideally, we would like to collapse nested pattern matches into one, which then becomes the normal form. Figure 5.20 shows an example of this, where the program to the left uses nested pattern matching and the one to the right does not. Notice, by the way, how the pattern matching on the left corresponds to an unfolded version of `head`.

Before	After

`let xs = e`
`in if xs == []`
`  then let (y:_) = xs in y`
`  else 42`
	`let xs@(y:_) = e`
	`in if xs == []`
	`  then y`
	`  else 42`

Figure 5.21: Unsound pattern lifting

INTRODUCING PATTERN LIFTING Our proposed solution to the problem of nested pattern matching is *pattern lifting*. This mechanism detects pattern matches on a variable that has been bound in a previous match. Then, it lifts the nested binding and merges it with the pattern it comes from.

Consider again Figure 5.20 as an example. We can see that `xs` is bound in the second alternative of the `case` and that `y` is bound by matching on `xs`. This means we can lift the pattern where `y` is bound, thereby merging the `xs` and `(y : _)` patterns into one. As you can see on the right of the figure, the resulting pattern is `xs@(y : _)`.

CONSERVATIVE AND AGGRESSIVE VARIANTS Pattern lifting as described above turns out to be unsound, as shown by the counterexample in Figure 5.21. In the original program, the head of the list is only taken if the list is not empty. However, pattern lifting moves the match upwards, thereby changing the semantics of the program. In the resulting code, an attempt to access the head of the list will be made even when the list is empty. We call this aggressive pattern lifting.

To prevent this problem, we define a conservative version of pattern lifting, which only supports irrefutable patterns. Irrefutable patterns are patterns that always match, like variables and wildcards. Also tuples containing other irrefutable patterns are themselves irrefutable. The same is valid for *as* patterns. With this constraint, pattern lifting is always semantics-preserving[2].

As expected, conservative pattern lifting results in a less effective normalization in comparison to aggressive pattern lifting. Table 5.8 shows the difference in the amounts of clusters depending on the pattern lifting mode used. In this comparison, all other transformations we add to Ask-Elle are enabled. Exercises where there is no difference are ignored.

IN DEFENSE OF AGGRESSIVE PATTERN LIFTING As mentioned above, aggressive pattern lifting is unsound. This means that it may

2 While laziness may be affected, the output of the program is not influenced.

Exercise	Conservative	Aggressive
1	3	2
4	13	12
6	17	16
7	30	28
8	20	18

Table 5.8: Effectiveness of pattern lifting

change the meaning of a program, forcing a pattern match earlier than specified in the original code.

In the context of normalization, this unsoundness means that an incorrect program could have the same normal form as a correct one. Allowing unsound transformations could lead Ask-Elle to say that two programs are equivalent, when in fact they are not. This is the case illustrated by Figure 5.21.

We think, however, that an incorrect program with the necessary characteristics is very unlikely to pass a basic test suite. Such a program would need to pattern match too early on a value, leading to an easy to detect crash. In light of this, we treat aggressive pattern lifting as a sound transformation in practice, even though it is technically unsound.

5.5.4 *Tuple deconstruction and reconstruction*

We oberve another normalization issue where students deconstruct and reconstruct a tuple instead of matching on its expression. This results in code like `let (a, b) = e in f (a, b)`, when `f e` would have been sufficient. We add a transformation to remove pattern matching in such a case.

5.5.5 *Tuple unrolling*

Ask-Elle's normalization inlines bindings to names, not to patterns. A consequence of this is that it will fail to inline names that are introduced inside a tuple pattern. Below we consider the problem from the perspectives of `let` and `case`, and we present a new transformation to solve it.

LET In the case of `let`, Figure 5.22 shows on the left a group of variables being introduced inside a tuple pattern. On the right is an equivalent version, where the tuple has been unrolled into separate bindings, one for each variable. We enhance Ask-Elle with a normalization step that performs this transformation.

Original	Unrolled

`let (a, b, c) = (x, y, z)`
`in e`
	`let a = x`
	`    b = y`
	`    c = z`
	`in e`

Figure 5.22: Tuple unrolling for `let`

Original	Unrolled

`case (a, b, c) of`
`    ([], _, _) -> []`
`    (x : xs, y, z) -> e`
	`let y = b`
	`    z = c`
	`in case a of`
	`    [] -> []`
	`    x : xs -> e`

Figure 5.23: Tuple unrolling for `case`

CASE Tuple unrolling for `case` is a bit more complicated than for `let`, as the patterns may influence the control flow of the program. Figure 5.23 shows on the left a `case` that illustrates the problem. It is clear that we cannot unroll the binding to `a`, as it is being used to choose which alternative to execute. Still, it is possible to unroll `b` and `c`, as shown on the right of the figure.

5.5.6 *Other case transformations*

Additional transformations to simplify `case` expressions include rewriting a single-alternative `case` as a `let` (Figure 5.24), removing the last alternative if it leads to `undefined` (Figure 5.25) and removing unreachable alternatives (Figure 5.26).

In the latter case, we use a conservative heuristic. An alternative is unreachable if the previous alternative matches on a:

- Wildcard;
- Variable;
- Tuple of wildcards or variables.

5.6 MISCELLANEOUS

Besides the domain-specific transformations mentioned above, we also add transformations that are not tied to a particular domain. Sim-

From	To
`case xs of` `  y:ys -> e`	`let (y:ys) = xs in e`

Figure 5.24: Rewriting `case` as `let`

From	To

`case xs of`
`  [] -> e`
`  y:ys -> undefined`
	`case xs of`
	`  [] -> e`

Figure 5.25: Removing last alternative leading to `undefined`

ple ones include inlining of non-recursive functions and dead code removal. The rest is described below.

5.6.1 *Beta reduction*

Since many of our transformations involve abstracting functions, it is likely that generalized beta reduction would result in infinite loops within the normalization procedure. Therefore we only allow beta reduction of whitelisted functions: `id`, `const`, `++`, `map`, `concat`, `foldr` and `flip`.

5.6.2 *Order arguments in commutative functions*

By definition, commutative functions like `(+)` or `(*)` return the same result independently of the order of their arguments. From a normalization perspective, this is a source of problems, because it results in programs which are semantically equivalent but syntactically different. For this reason, we add a transformation that sorts the arguments passed to a commutative function. While advanced algorithms exist to unify expressions up to associativity and commutativity, our ap-

From	To

`case xs of`
`  ys -> e`
`  zs -> e'`
	`case xs of`
	`  ys -> e`

Figure 5.26: Removing unreachable alternative

From	To
`\x -> x`	`id`
`(>>>)`	`flip (.)`

Table 5.9: Abstracting and unfolding general purpose functions

From	To
`flip (\x y -> e)`	`\y x -> e`

Figure 5.27: Swith the parameter order in a lambda when applying `flip`

proach works well-enough in practice. Again, laziness is not taken into consideration

5.6.3 *Abstract and unfold functions*

Similar to domain-specific transformations, we abstract and unfold general purpose functions. The transformations are summarized in Table 5.9.

Notice that the transformation involving `(>>>)` is possible because Ask-Elle restricts the operator to only work on functions. Any attempt to use it on different types results in a compile error.

5.6.4 *Transformations involving flip*

Besides beta-reducing `flip`, we add a transformation to remove it whenever it is applied to a commutative function. Since commutative functions do not care about the order of the arguments, removing `flip` is semantics-preserving.

Another transformation involving `flip` is switching the order of the parameters when it is applied to a lambda. This is illustrated by figure 5.27.

5.6.5 *Eta reduction*

We enhance the eta-reduction algorithm by allowing it to insert `flip` when necessary. Figure 5.28 shows the result of applying transformation to one of the student submissions.

Before

```
printLine xs =
    "+" ++ intercalate "+" (map (\x -> replicate '-' x) xs)
        ++ "+"
```

After

```
printLine xs =
    "+" ++ intercalate "+" (map (flip replicate '-') xs)
        ++ "+"
```

Figure 5.28: Flip-aware eta-reduction

6 MEASURED IMPROVEMENTS

After implementing the transformations, we measure the effectiveness of the new normalization strategy. We look at the results from the perspective of submission coverage and normalization effectiveness.

6.1 SUBMISSION COVERAGE

Recall that Ask-Elle considers two programs to be semantically equivalent if their normal forms are syntactically similar. Additionally, a program is considered correct if it is equivalent to one of the model solutions.

While it is possible to have two model solutions that share the same normal form, this does not increase the amount of programs that are recognized. In such a case, removing one of the model solutions would have no impact.

If we only consider model solutions with different normal forms, we can associate each of them to a different cluster of programs. If the submissions to an exercise are grouped in 10 clusters, then having 10 model solutions (one per cluster) would recognize 100% of programs.

With *submission coverage*, we measure the percentage of programs that are recognized using 5 model solutions or less. Figures 6.1 through 6.8 show graphs for each exercise. The circle shows the percentage of solutions originally recognized by Ask-Elle and the cross shows the same measurement after our improvements. Notice how additional model solutions offers diminishing returns.

6.2 NORMALIZATION EFFECTIVENESS

As mentioned in 4.1.2, we say that a given normalization procedure is more effective than a second one when it results in a lower amount of clusters for the same exercise. Our results show that, for all exercises, the new normalization is more effective than Ask-Elle's original one.

Table 6.1 shows the amount of clusters per exercise. We also provide figures 6.9 through 6.16 that present the numbers in a more graphical way. Additionally, the figures show the differences in size of each cluster. Notice how the amount of clusters diminishes, while the size of the clusters grows.

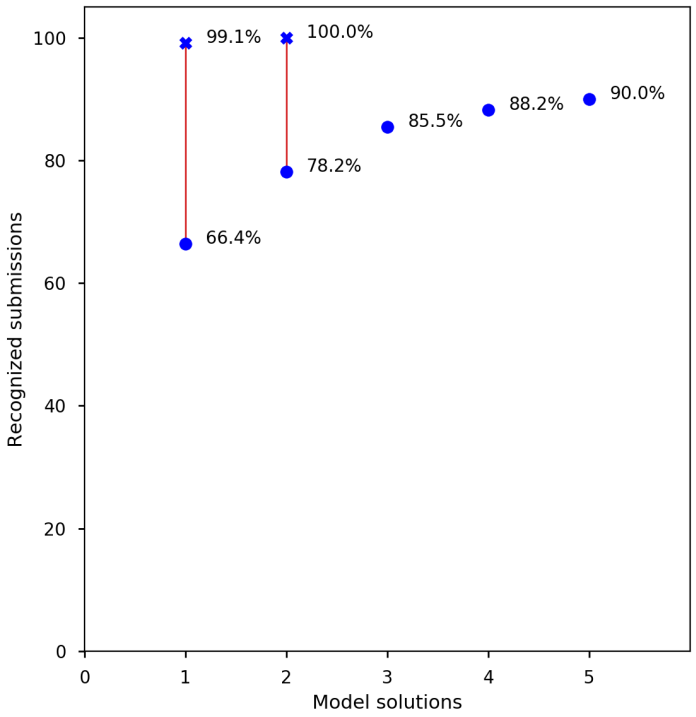


Figure 6.1: Exercise 1 - Improvement in submission coverage

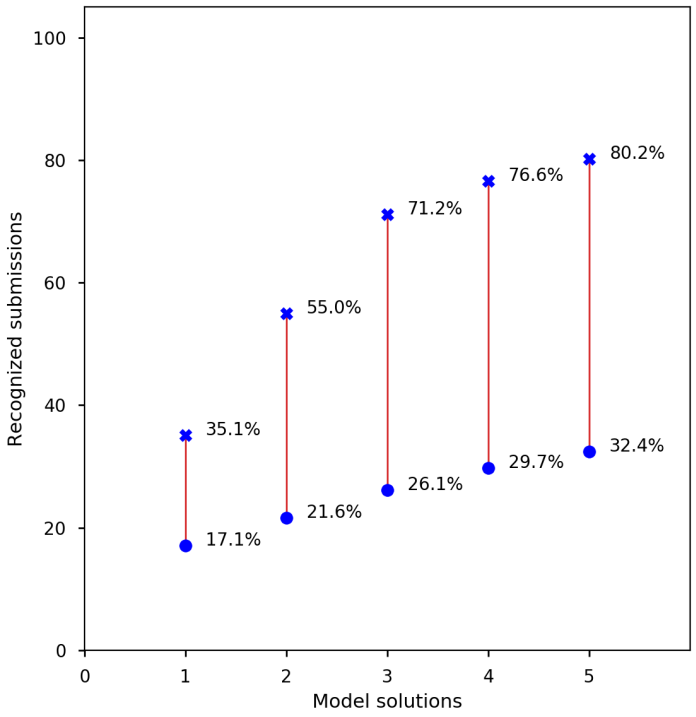


Figure 6.2: Exercise 2 - Improvement in submission coverage

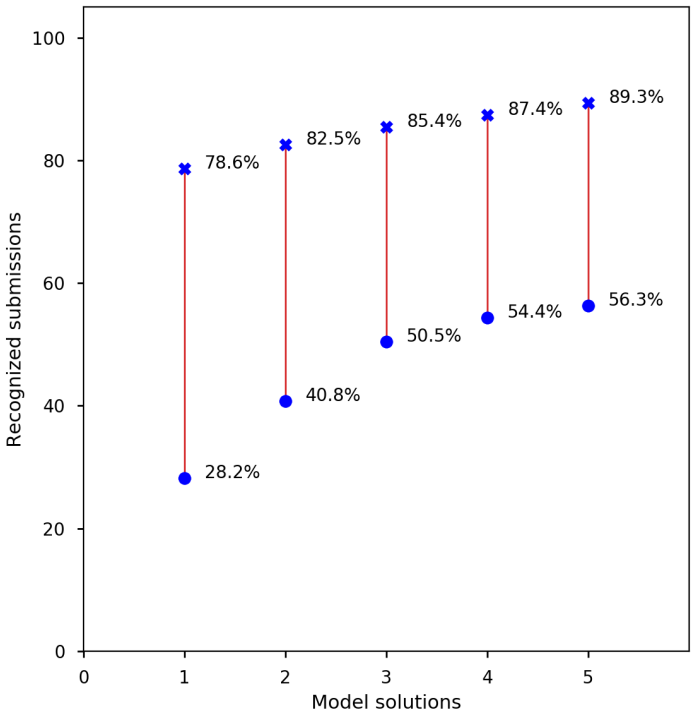


Figure 6.3: Exercise 3 - Improvement in submission coverage

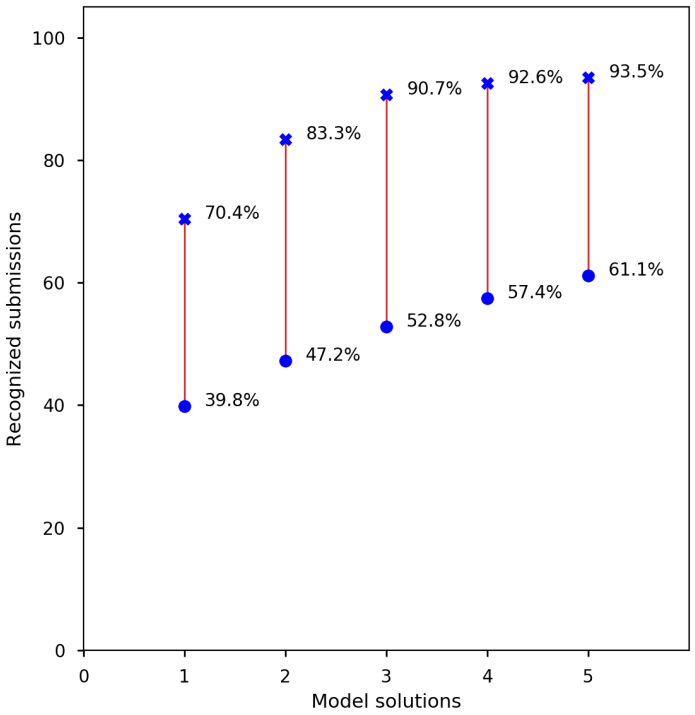


Figure 6.4: Exercise 4 - Improvement in submission coverage

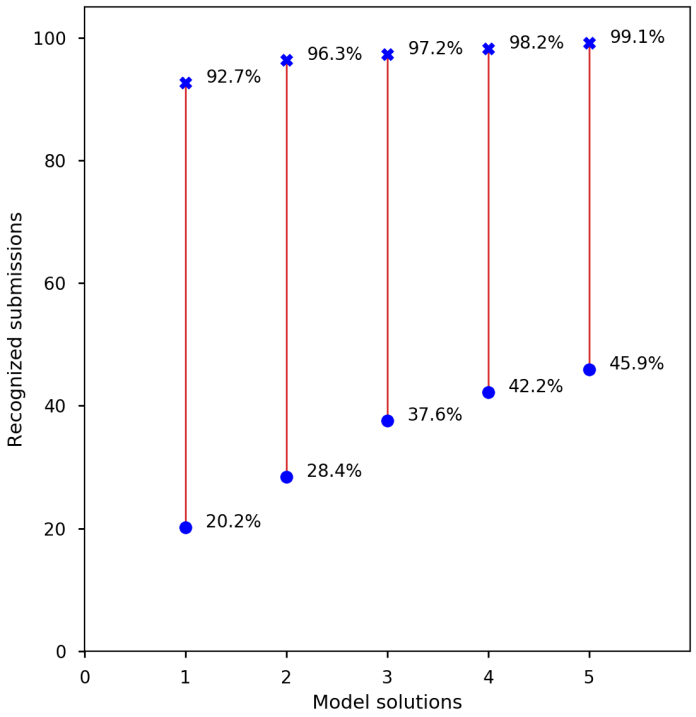


Figure 6.5: Exercise 5 - Improvement in submission coverage

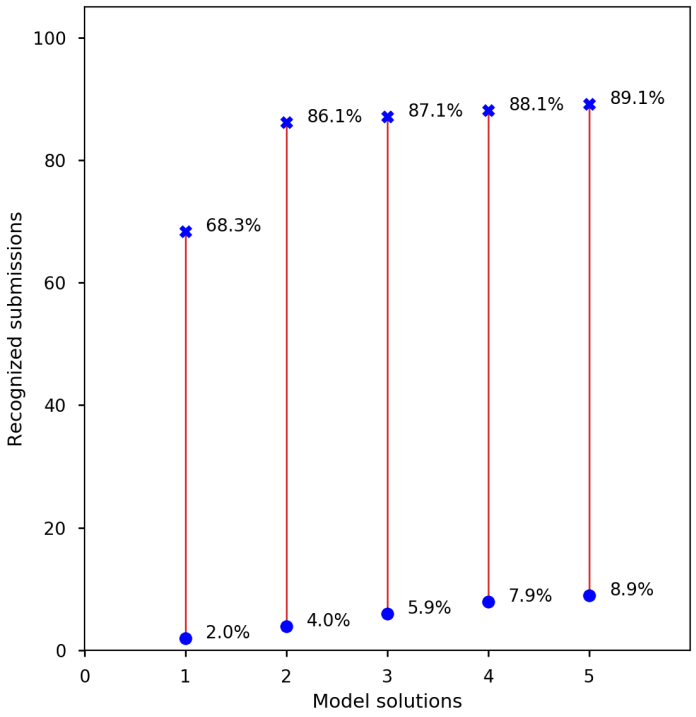


Figure 6.6: Exercise 6 - Improvement in submission coverage

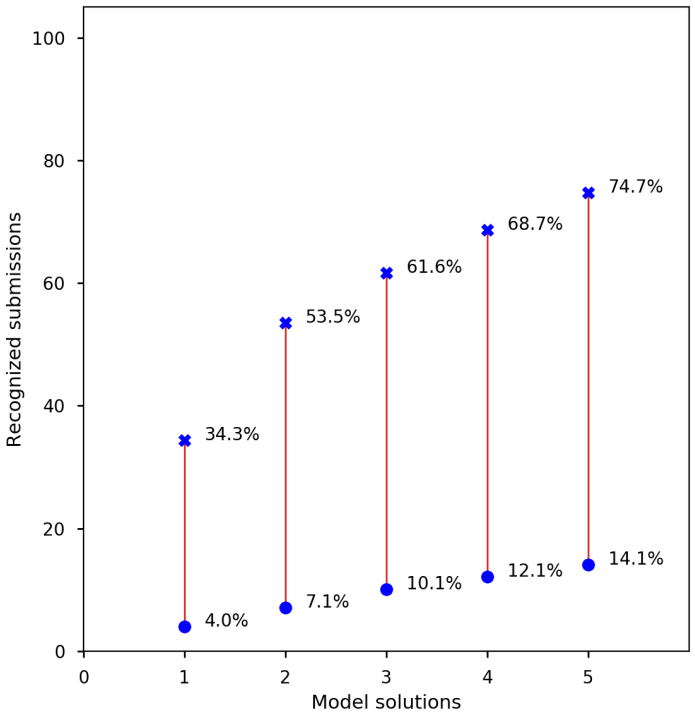


Figure 6.7: Exercise 7 - Improvement in submission coverage

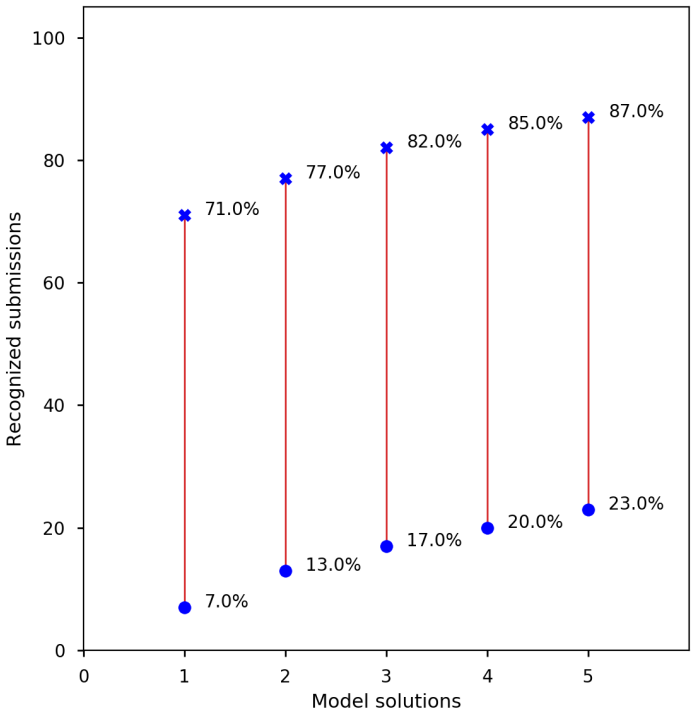


Figure 6.8: Exercise 8 - Improvement in submission coverage

Exercise	Before	After
1	15	2
2	64	19
3	43	16
4	38	12
5	43	6
6	97	16
7	84	28
8	76	18

Table 6.1: Improvement in normalization effectiveness

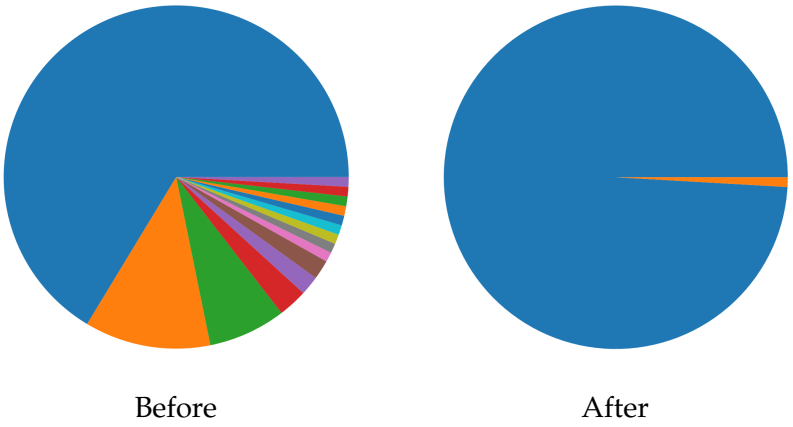


Figure 6.9: Exercise 1 - Improvement in normalization effectiveness

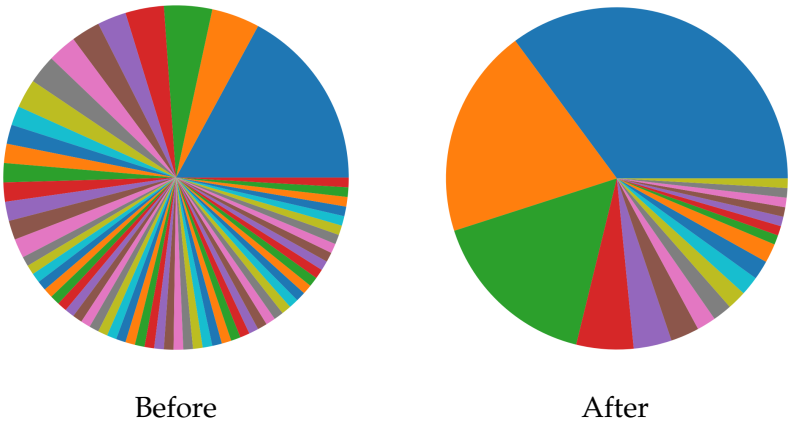
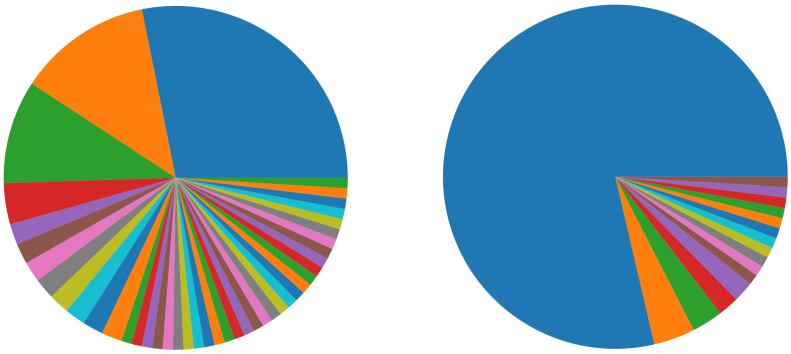
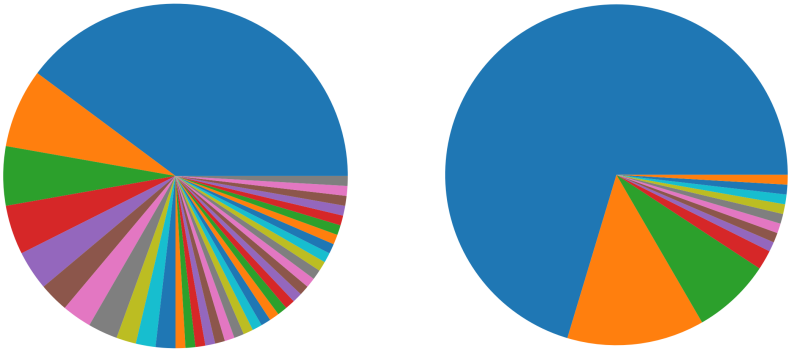


Figure 6.10: Exercise 2 - Improvement in normalization effectiveness



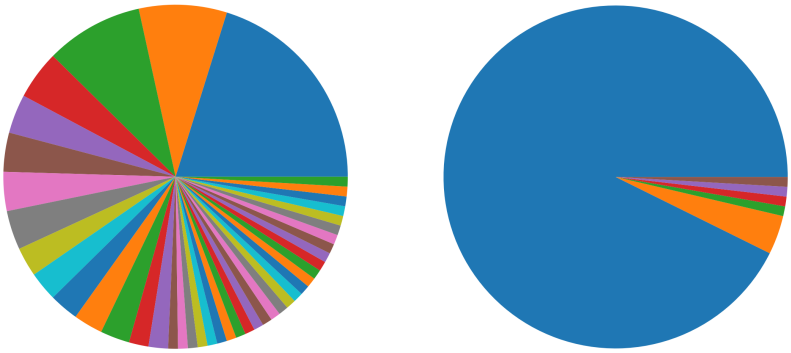
Before After

Figure 6.11: Exercise 3 - Improvement in normalization effectiveness



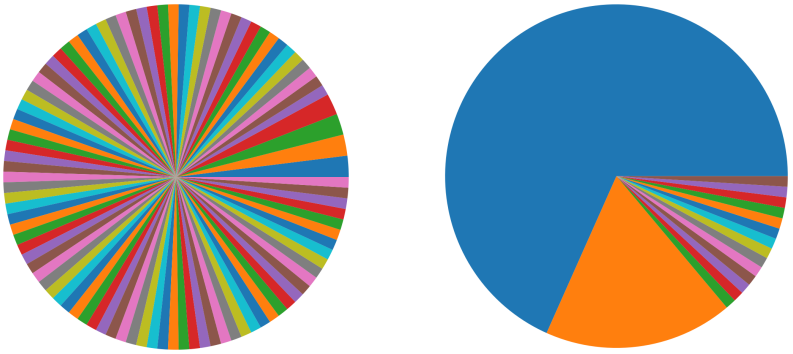
Before After

Figure 6.12: Exercise 4 - Improvement in normalization effectiveness



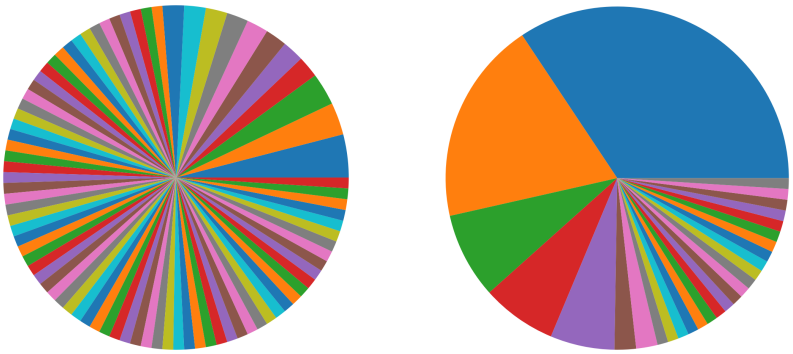
Before After

Figure 6.13: Exercise 5 - Improvement in normalization effectiveness



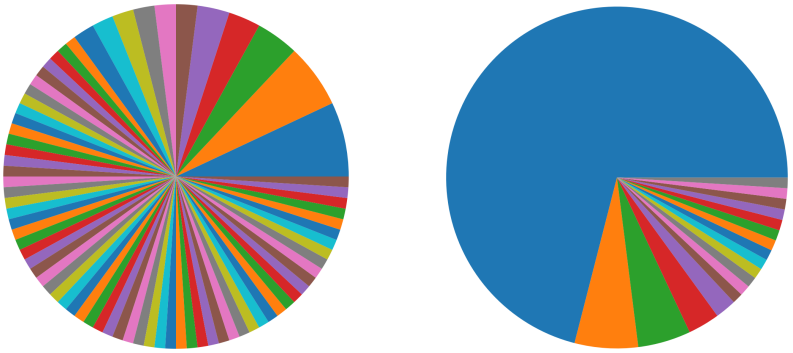
Before After

Figure 6.14: Exercise 6 - Improvement in normalization effectiveness



Before After

Figure 6.15: Exercise 7 - Improvement in normalization effectiveness



Before After

Figure 6.16: Exercise 8 - Improvement in normalization effectiveness

7 CONCLUSION

In Section 1.3, we set out to find the cases in which Ask-Elle's normalization does not work as expected. Based on this knowledge, we aimed to improve the normalization procedure so it could handle a wider range of programs.

7.1 RESULTS

The results from sections 5 and 6 are very positive. We discovered many cases in which normalization did not work as expected and achieved substantial improvements without resorting to advanced analysis techniques.

In the domain of preconditions, we observed that they cause more problems than they solve, as unification becomes more challenging and confusion creeps among students. Due to the complexity of this domain, we only added a simple heuristic to remove unnecessary pattern matching on the empty list.

Regarding lists, there were multiple issues, as expected by the nature of the assignment. We added a function abstraction heuristic, a transformation to desugar literals and a transformation to apply list laws.

The domain of booleans also benefited from new transformations. We now apply boolean laws, reason about negation and rewrite guards as `if` expressions in more cases.

Regarding the `Maybe` type, we added a function abstraction heuristic, a transformation to apply `Maybe` laws and a transformation to unfold the `maybe` function.

The domain of patterns saw improvements with transformations to simplify patterns, unfold functions that pattern match, simplify tuple matching, simplify `case` expressions and lift nested patterns.

Miscellaneous improvements include enabling beta reduction for whitelisted functions, enhancing eta reduction, specifying an order for arguments to commutative functions, abstracting and unfolding general-purpose functions and adding `flip` laws.

7.2 FUTURE WORK

While our results are very positive, there is still room for improvement. In the first place, we did not address all normalization issues and left out potentially useful transformations. Secondly, the interaction of the new transformations with Ask-Elle's stepwise feedback

feature is unclear. Also, it would be worthwhile to measure the improvements with a different dataset. Finally, our experience with Coq proofs suggests that exploring alternative unification approaches may yield interesting results.

7.2.1 *New transformations*

As mentioned above, we did not implement some transformations that would have probably resulted in improvements. The main factors that influenced this decision were time constraints and the expected low number of submissions that would benefit. In some cases, time constraints led us to leave out a transformation even though it would have had a bigger impact. Below we present some of these cases.

PRECONDITION-AWARE PROGRAM SLICING Preconditions are particularly problematic in the context of unification, as they allow programs with different semantics to be considered equivalent. A mechanism that performs unification up to preconditions would need to ignore the case in which the precondition does not hold.

Within Ask-Elle's normalization approach to unification, precondition-aware program slicing offers a promising solution. In many cases, program slicing should be able to identify the part of the program that is executed when the precondition does not hold. With this knowledge, a transformation could remove said part of the program. We call such a transformation semantics preserving up to preconditions.

FROM CHECKS TO PATTERN MATCHING Using checks instead of pattern matching does not necessarily alter the semantics of the program. Therefore it is possible to implement a semantics-preserving transformation to replace checks by pattern matching. Such a transformation would have been beneficial in the domain of lists and `Maybe`.

A typical case where this would be useful is accessing the head of a list after checking that it is not empty. While the idiomatic solution is to pattern match on the list, it is also possible to use the `null` function in combination with `head`.

Another case is accessing the value inside a `Maybe`. Again, one of the idiomatic solutions involves pattern matching, but it is also possible to use the `isJust` function in combination with `fromJust`.

In both cases, replacing checks by pattern matching would result in a lower number of clusters, thereby improving the effectiveness of the normalization procedure.

FROM LIST INDEXING TO FUNCTIONAL STYLE One of the submissions to Exercise 2 reimplements `map` through list indexing and list comprehensions (see Figure 5.11 for details). While it would be straightforward to add an ad-hoc transformation that handles this

case, it would be interesting to investigate the possibility of a transformation able to abstract multiple list functions from code that uses indexing.

7.2.2 *Stepwise feedback*

In Section 2, we explain how programming strategies are closely related to normalization. Particularly, the mechanism to identify the step of the strategy where a program is involves normalizing the program. Since strategies are used to provide hints to incomplete programs, they involve normalizing incomplete programs as well.

Our research focuses on programs that do not contain any holes. Therefore, the improvements we achieve do not necessarily apply to stepwise feedback. For instance, if the model solution has the form `parseTable = map words`, then `parseTable xs = ?` does not unify with it, even though `parseTable xs = map words xs` does.

An interesting approach is to integrate transformations themselves as parts of the strategy, which has the added benefit that they can be used to provide hints. For instance, a transformation that suggests replacing `concat` instead of `intercalate []` would help students with their style.

7.2.3 *Alternative unification approaches*

The proofs we wrote to ensure the soundness of the new transformations turned out to be surprisingly simple (see Appendix B). This hints at the possibility of using automated theorem provers to unify Haskell programs. An interesting approach would be to automatically rewrite student programs as Coq programs and let Coq unify them, eventually making use of a custom tactic. Afterwards, the results could be compared against Ask-Elle's normalization-based approach.

7.2.4 *Additional proofs*

When constructing proofs for the new transformations we made a choice to keep them simple. Particularly, we focused on transformations that can be expressed as rewrite rules in Coq's type system. This leaves out transformations that involve reasoning about variables and their scope, such as function inlining, dead code removal and pattern lifting. It would be interesting to develop a model in which these transformations can be proven correct.

APPENDIX

A

ASSIGNMENT DOCUMENT

This appendix contains a copy of the document describing *Assignment I*, used in the functional programming course at Universiteit Utrecht in the year 2017/2018. To keep the original formatting, the document starts in next page.

Functional Programming 2017/2018
Assignment 1: Lists

Ruud Koot

In this exercise we will read in a database, perform a simple query on it and present the results to the user in an aesthetically pleasing form. Most exercises can be completed by combing functions from the *Prelude* and the libraries *Data.Char*, *Data.List* and *Data.Maybe* and contain a hint on which functions you could use from these libraries; often a completely different solution, not using these functions, is also possible. A starting framework and the sample database can be found on the Assignments page on the course website.

1 Parsing

A plain text database consists of a number of lines (each line is called a *row*), with on each line a fixed number of *fieds* separated by a single space. The first row a database table is called the *header* and contains the names of the columns in the table. An example of such a database would be:

```
first last gender salary
Alice Allen female 82000
Bob Baker male 70000
Carol Clarke female 50000
Dan Davies male 45000
Eve Evans female 275000
```

One way of modeling such databases in Haskell would be using the following types:

$$\begin{aligned}&\textbf{type } \mathit{Field} = \mathit{String}\\ &\textbf{type } \mathit{Row} = [\mathit{Field}]\\ &\textbf{type } \mathit{Table} = [\mathit{Row}]\end{aligned}$$

A field is always modeled as a string (even though the database may contain strings that look very much like numbers), a row is a list of fields and a table a list of rows. The head of this list corresponds to the header of the table. (A valid table always has a header and always has at least one column.)

There are several "problems" with this model: for example, it does not enforce that each of the rows in the table must have the same number of fields. However, for the purposes of this first assignment it will suffice. You may assume that all the databases that are presented to program will be well-formed, that is to say, they will always have the same number of fields on each line.

The form in which data is stored inside a file, printed or written on paper, or entered from the keyboard is called its *concrete syntax*. The form in which data is manipulated inside a program is called its *abstract syntax*. The process of transforming some object represented in its concrete syntax into its representation in abstract syntax is called *parsing*.

Exercise 1. *Write a function* $\mathit{parseTable} :: [\mathit{String}] \to \mathit{Table}$ *that parses a table represented in its concrete syntax as a list of strings (each corresponding to a single line in the input) into its abstract syntax. (Hint: use the function words from the Prelude.)*

2 Pretty printing

In the previous exercise we have seen how we can turn concrete syntax into abstract syntax. The reverse operation—turning abstract syntax into concrete syntax—is often called *pretty printing* or *compilation*.

In our case we do not want to convert our abstract syntax into the original concrete syntax, but into a different concrete syntax that is easier to read for humans:

```
+-----+------+------+------+
|FIRST|LAST  |GENDER|SALARY|
+-----+------+------+------+
|Alice|Allen |female| 82000|
|Bob  |Baker |male  | 70000|
|Carol|Clarke|female| 50000|
|Dan  |Davies|male  | 45000|
|Eve  |Evans |female|275000|
+-----+------+------+------+
```

An apt name for this process might be "prettier printing". Note that we have done several things to make the result look nice:

1. We have made the width of each column exactly as wide as the widest field in this column (including the name in the header).
2. We have added a very fancy looking border around the table, the header and columns.
3. We have typeset the names of the columns in the header in uppercase.
4. We have right-aligned fields that look like (whole) numbers.

Exercise 2. *Write a function* $printLine :: [Int] \to String$ *that, given a list of widths of columns, returns a string containing a horizontal line. For example,* $printLine\ [5,6,6,6]$ *should return the line* `"+-----+------+------+------+"`. *(Hint: use the function replicate.)*

If you can write this function using *foldr* you will get more points for style.

Exercise 3. *Write a function* $printField :: Int \to String \to String$ *that, given a desired width for a field and the contents of a fields, returns a formatted field by adding additional whitespace. If the field only consists of numerical digits, the field should be right-aligned, otherwise it should be left-aligned. (Hint: use the functions all, isDigit and replicate.)*

The function *printField* should satisfy the property:

$$\forall n\ s.n \geqslant length\ s \Rightarrow length\ (printField\ n\ s) \equiv n$$

Later in the course we shall see how we can use these properties to test the correctness of a program, or even proved that such properties must always hold for a given program.

Exercise 4. *Write a function* $printRow :: [(Int, String)] \to String$ *that, given a list of pairs—the left element giving the desired length of a field and the right element its contents—formats one row in the table. For example,*

$$printRow\ [(5, \texttt{"Alice"}), (6, \texttt{"Allen"}), (6, \texttt{"female"}), (6, \texttt{"82000"})]$$

should return the formatted row

```
"|Alice|Allen |female| 82000|"
```

(Hint: use the functions intercalate, map and uncurry.)

Exercise 5. *Write a function* $columnWidths :: Table \to [Int]$ *that, given a table, computes the necessary widths of all the columns. (Hint: use the functions length, map, maximum and transpose.)*

Exercise 6. *Write a function* $printTable :: Table \to [String]$ *that pretty prints the whole table. (Hint: use the functions map, toUpper and zip.)*

3 Querying

Finally we will write a few simple query operations to extract data from the tables.

Exercise 7. *Write a function* $select :: Field \to Field \to Table \to Table$ *that given a column name and a field value, selects only those rows from the table that have the given field value in the given column. For example, applying the query operation*

```
select "gender" "male"
```

to the table

```
+-----+------+
|FIRST|GENDER|
+-----+------+
|Alice|female|
|Bob  |male  |
|Carol|female|
|Dan  |male  |
|Eve  |female|
+-----+------+
```

should result in the table

```
+-----+------+
|FIRST|GENDER|
+-----+------+
|Bob  |male  |
|Dan  |male  |
+-----+------+
```

If the given column is not present in the table then the table should be returned unchanged. (Hint: use the functions (!!)*, elemIndex, filter and maybe.)*

Exercise 8. *Write a function* $project :: [Field] \to Table \to Table$ *that projects several columns from a table. For example, applying the query operation*

```
project ["last","first","salary"]
```

to the table

```
+-----+------+------+------+
|FIRST|LAST  |GENDER|SALARY|
+-----+------+------+------+
|Alice|Allen |female| 82000|
|Carol|Clarke|female| 50000|
|Eve  |Evans |female|275000|
+-----+------+------+------+
```

should result in the table

```
+------+-----+------+
|LAST  |FIRST|SALARY|
+------+-----+------+
|Allen |Alice| 82000|
|Clarke|Carol| 50000|
|Evans |Eve  |275000|
+------+-----+------+
```

If a given column is not present in the original table it should be omitted from the resulting table. (Hint: use the functions (!!)*, elemIndex, map, mapMaybe, transpose.)*

4 Wrapping up

We can tie parsing, printing and two query operations together using:

```
exercise :: [String] → [String]
exercise = parseTable ⋙ select "gender" "male"
                      ⋙ project ["last","first","salary"] ⋙ printTable
```

and have the program reads and write from and to standard input and standard output using:

```
main :: IO ()
main = interact (lines ⋙ exercise ⋙ unlines)
```

B

PROOFS

```
Add LoadPath ".".

Load CpdtTactics.
Require Import Coq.Bool.Bool.
Require Import Coq.Lists.List.
Require Import Coq.Program.Basics.
Import ListNotations.

(* An equivalent of Haskell's intercalate function *)
Fixpoint intercalate {T : Type} (x : list T) (xs : list (list T)) : list T :=
  match xs with
  | [] => []
  | [y] => y
  | y :: ys => y ++ x ++ intercalate x ys
  end.

(* An equivalent of Haskell's concatMap function *)
Definition concatMap {T U : Type} (f : T -> list U) (xs : list T) : list U :=
  concat (map f xs).

(* An equivalent of Haskell's replicate function *)
Fixpoint replicate {T} n (x : T) : list T :=
  match n with
  | 0 => []
  | S n' => x :: replicate n' x
  end.

(* An equivalent of Haskell's mapMaybe function *)
Fixpoint mapMaybe {T U} (f : T -> option U) (xs : list T) : list U :=
  match xs with
  | [] => []
  | y :: ys => match f y with
               | Some y' => y' :: mapMaybe f ys
               | None => mapMaybe f ys
               end
  end.

(* An equivalent of Haskell's and function *)
Fixpoint and (xs : list bool) : bool :=
  match xs with
  | [] => true
```

```
  | false :: _ => false
  | true :: ys => and ys
  end.

(* An equivalent of Haskell's fmap function *)
Definition fmap {T U} (f : T -> U) (x : option T) : option U :=
  match x with
  | Some y => Some (f y)
  | None => None
  end.

(* An equivalent of Haskell's isNothing function *)
Definition isNothing {T} (x : option T) : bool :=
  match x with
  | Some _ => false
  | None => true
  end.

(* An equivalent of Haskell's isJust function *)
Definition isJust {T} (x : option T) : bool :=
  match x with
  | Some _ => true
  | None => false
  end.

(* An equivalent of Haskell's maybeToList function *)
Definition maybeToList {T} (x : option T) : list T :=
  match x with
  | Some y => [y]
  | None => []
  end.

Lemma id_works : forall {T} (x : T), id x = x.
  Proof. auto. Qed.

Lemma cons_cong : forall {T} (x : T) xs ys,
                  xs = ys <-> x :: xs = x :: ys.
Proof. intros. split. congruence. congruence. Qed.

Theorem append_identity : forall {T} (xs : list T), xs ++ [] = xs.
  Proof. intuition. Qed.

Theorem intercalate_concat : forall {T} (xs : list (list T)),
                             intercalate [] xs = concat xs.
  Proof. intros. induction xs.
    auto.
```

```
    simpl. rewrite IHxs. induction xs.
      unfold concat. rewrite app_nil_r. reflexivity.
      reflexivity.
  Qed.

Theorem foldr_append : forall {T} (xs : list (list T)),
    @fold_right (list T) (list T) (@app T) [] xs = concat xs.
  Proof. auto. Qed.

Theorem foldr_cons : forall {T} (xs ys : list T),
    fold_right cons xs ys = ys ++ xs.
  Proof. intros. induction ys.
    auto.
    simpl. congruence.
  Qed.

Theorem concat_map : forall {T U} (f : T -> list U) xs,
    concat (map f xs) = concatMap f xs.
  Proof. auto. Qed.

Theorem concat_map_filter :
    forall {T} (cond : T -> bool) (xs : list T),
    concatMap (fun x => if cond x then [x] else []) xs = filter cond xs.
  Proof. intros. induction xs.
    auto.
    simpl. rewrite <- IHxs. unfold concatMap. simpl. destruct (cond a).
      auto.
      auto.
  Qed.

Theorem zero_index : forall T (xs : list T), nth_error xs 0 = hd_error xs.
  Proof. auto. Qed.

Theorem map_id : forall (T : Type) (xs : list T), map id xs = id xs.
  Proof.
    intros T xs.
    induction xs.
      auto.
      simpl. unfold id. apply (cons_cong a (map id xs) xs). auto.
  Qed.

Theorem concat_map_to_map :
    forall {T} (xs : list T),
    concatMap (fun x => [x]) xs = map (fun x => x) xs.
  Proof. intros. induction xs.
    auto.
```

```
    simpl.
      rewrite <- (cons_cong a (concat (map (fun x => [x]) xs)) _).
      rewrite <- IHxs.
      rewrite (concat_map _ _).
      reflexivity.
  Qed.

Theorem concat_map_comp :
    forall {A B C} (f : B -> list C) (g : A -> B) (xs : list A),
    concatMap f (map g xs) = concatMap (compose f g) xs.
  Proof. intros. induction xs.
    auto.
    simpl. unfold compose.
      rewrite <- (concat_map f _).
      rewrite <- (concat_map _ (a :: xs)).
      simpl.
      rewrite concat_map.
      rewrite concat_map.
      rewrite IHxs.
      reflexivity.
  Qed.

Theorem concat_map_flip : forall T (xs : list T),
                          concatMap (flip cons []) xs = id xs.
  Proof. intros. induction xs.
    auto.
    unfold id in IHxs. simpl. unfold id. apply cons_cong. auto.
  Qed.

Theorem concat_replicate : forall {T} n (x : T),
                           concat (replicate n [x]) = replicate n x.
  Proof. intros. induction n.
    auto.
    simpl. congruence.
  Qed.

Theorem map_map_comp :
    forall {A B C} (f : B -> C) (g : A -> B) (xs : list A),
    map f (map g xs) = map (compose f g) xs.
  Proof. intros. induction xs.
    auto.
    simpl. rewrite IHxs. unfold compose. reflexivity.
  Qed.

Theorem map_maybe_fmap :
    forall {A B} (f : nat -> B) (xs : list A),
```

```
    mapMaybe (fun x => fmap f (Some 42)) xs
      = map f (mapMaybe (fun x => Some 42) xs).
  Proof. intros. induction xs.
    auto.
    simpl. rewrite <- cons_cong. auto.
  Qed.

Theorem map_maybe_just :
    forall T (xs : list T),
    mapMaybe (fun x => Some 42) xs = map (fun x => 42) xs.
  Proof. intros. induction xs.
    auto.
    simpl. congruence.
  Qed.

Theorem cons_append : forall T (x : T) (xs ys : list T),
                       (x :: xs) ++ ys = x :: (xs ++ ys).
  Proof. auto. Qed.

Theorem append_assoc : forall T (xs ys zs : list T),
                        (xs ++ ys) ++ zs = xs ++ (ys ++ zs).
  Proof. crush. Qed.

Theorem is_nothing : forall T (x : option T),
                      x = None <-> isNothing x = true.
  Proof. intros. unfold isNothing. destruct x. unfold iff.
    split. discriminate. discriminate.
    split. auto. auto.
  Qed.

Theorem is_just : forall T y (x : option T),
                   x = Some y -> isJust x = true.
  Proof. crush. Qed.

Theorem maybe_to_list_1 : forall T (x : option T),
                           maybeToList x = [] <-> isNothing x = true.
  Proof. intros. destruct x.
    simpl. split. discriminate. intros. contradict H. auto.
    simpl. split. auto. auto.
  Qed.

Theorem maybe_to_list_2 : forall T (x : option T),
                           maybeToList x <> [] <-> isJust x = true.
  Proof. intros. destruct x.
    simpl. split. auto. intros. pose proof nil_cons as P. auto.
    simpl. split. auto. intros. contradict H. auto.
```

```
  Qed.

Theorem maybe_to_list_3 : forall T (x : option T),
                           nth_error (maybeToList x) 0 = x.
  Proof. intros. destruct x. auto. auto. Qed.

Theorem not_is_just : forall T (x : option T),
                      isNothing x = negb (isJust x).
  Proof. intros. destruct x. auto. auto. Qed.

Theorem true_id : forall b, eqb b true = id b.
  Proof. destr_bool. Qed.

Theorem false_neg : forall b, eqb b false = negb b.
  Proof. destr_bool. Qed.

Theorem and_id : forall b, andb b true = id b.
  Proof. destr_bool. Qed.

Theorem foldr_and : forall xs, fold_right andb true xs = and xs.
  Proof. auto. Qed.

Theorem and_map : forall T (f : T -> bool) xs,
                  and (map f xs) = forallb f xs.
  Proof. intros. induction xs.
    auto.
    simpl. unfold andb. rewrite IHxs. reflexivity.
  Qed.

Theorem all_id : forall xs, forallb id xs = and xs.
  Proof. crush. Qed.

Theorem simplify_if : forall (cond : bool),
                      (if cond then true else false) = cond.
  Proof. destr_bool. Qed.

Theorem switch_if_branches :
    forall T cond (x y : T),
    (if negb cond then x else y) = (if cond then y else x).
  Proof. destr_bool. Qed.
```

BIBLIOGRAPHY

[1] Kirsti M Ala-Mutka. A survey of automated assessment approaches for programming assignments. *Computer science education*, 15(2):83–102, 2005.

[2] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[3] Henk Barendregt, Wil Dekkers, and Richard Statman. *Lambda calculus with types*. Cambridge University Press, 2013.

[4] Timothy S Gegg-Harrison. Adapt: Automated debugging in an adaptive prolog tutor. In *International Conference on Intelligent Tutoring Systems*, pages 343–350. Springer, 1992.

[5] Alex Gerdes, Bastiaan Heeren, Johan Jeuring, and L Thomas van Binsbergen. Ask-Elle: an adaptable programming tutor for haskell giving automated feedback. *International Journal of Artificial Intelligence in Education*, 27(1):65–100, 2017.

[6] Alex Gerdes, Johan T Jeuring, and Bastiaan J Heeren. Using strategies for assessment of programming exercises. In *Proceedings of the 41st ACM technical symposium on Computer science education*, pages 441–445. ACM, 2010.

[7] Elena L Glassman, Jeremy Scott, Rishabh Singh, Philip J Guo, and Robert C Miller. Overcode: Visualizing variation in student solutions to programming problems at scale. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 22(2):7, 2015.

[8] Jun Hong. Guided programming and automated error analysis in an intelligent prolog tutor. *International Journal of Human-Computer Studies*, 61(4):505–534, 2004.

[9] Jonathan Huang, Chris Piech, Andy Nguyen, and Leonidas Guibas. Syntactic and functional variability of a million code submissions in a machine learning mooc. In *AIED 2013 Workshops Proceedings Volume*, page 25, 2013.

[10] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. Towards a systematic review of automated feedback generation for programming exercises. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '16, pages 41–46, New York, NY, USA, 2016. ACM.

[11] Ruud Koot. Functional programming 2017/2018. assignment 1: Lists. Available at http://www.staff.science.uu.nl/~f100183/fp/practicals/Assignment1.pdf. Accessed 28 May 2018.

[12] Simon Marlow et al. Haskell 2010 language report. *Available online https://www.haskell.org/definition/haskell2010.pdf*, 2010.

[13] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of logic and computation*, 1(4):497–536, 1991.

[14] Andy Nguyen, Christopher Piech, Jonathan Huang, and Leonidas Guibas. Codewebs: scalable homework search for massive open online programming courses. In *Proceedings of the 23rd international conference on World wide web*, pages 491–502. ACM, 2014.

[15] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

[16] Kelly Rivers and Kenneth R Koedinger. Data-driven hint generation in vast solution spaces: a self-improving python programming tutor. *International Journal of Artificial Intelligence in Education*, 27(1):37–64, 2017.

[17] Martin Sulzmann, Manuel MT Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System f with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 53–66. ACM, 2007.

[18] Tiantian Wang, Xiaohong Su, Yuying Wang, and Peijun Ma. Semantic similarity-based grading of student programs. *Information and Software Technology*, 49(2):99–107, 2007.

[19] Songwen Xu and Yam San Chee. Transformation-based diagnosis of student programs for programming tutoring systems. *IEEE Transactions on Software Engineering*, 29(4):360–384, 2003.