Extending a unification-based algorithm for use on real student data in Ask-Elle

I.T.B. Perdijk

28 June 2019

Bachelor Thesis - 7.5 ECTS
BSc Artificial Intelligence
Reasoning and Language
Utrecht University

Supervision:
prof. dr. J.T. Jeuring,
dr. ir. A. Serrano Mena

Second assessment:
dr. R.M. van Lambalgen

Abstract

Ask-Elle is an adaptable tutor, designed to help students learn the programming language Haskell. A teacher specifies a model solution and Ask-Elle uses this model to provide stepwise, interactive feedback to students while they solve the programming exercise. Ask-Elle uses normalisation to compare two programs for equivalence, to assess the correctness of a program and to provide hints. Normalisation is only suitable for comparing complete programs. Serrano Mena et al. are developing a unification-based algorithm UM, which is able to compare both incomplete and complete programs. By using algorithm UM we can potentially improve the feedback system of Ask-Elle. The main contribution of our work is to extend the unification-based algorithm to work on real student data. We analyse the situations in which the algorithm can not completely process the student data and extend the algorithm accordingly. We construct a pipeline to process student data in order to compare the results of unification to Ask-Elle's normalisation and a strategy-based check. We analyse unification results and successfully enhance algorithm UM.

Contents

1 Introduction **2**
- 1.1 Ask-Elle in Action . . . 2
- 1.2 Program Unification . . . 3
  - 1.2.1 Normalisation . . . 3
  - 1.2.2 Higher-Order Unification . . . 4
- 1.3 Relevance . . . 4
- 1.4 Research Questions . . . 4

2 Background Knowledge **5**
- 2.1 Related Work . . . 5
- 2.2 Strategies . . . 5
  - 2.2.1 Checking Against a Strategy . . . 5
- 2.3 Higher-Order Unification . . . 6
- 2.4 Algorithm UM . . . 7
  - 2.4.1 Local Functions in λ-Terms . . . 7
  - 2.4.2 Pragmatics-Aware Unification . . . 8
  - 2.4.3 Algorithm UM in Action . . . 8
- 2.5 Similarities . . . 10

3 Methodology **11**
- 3.1 Student Data . . . 12
- 3.2 Analysis . . . 12

4 The Pipeline **13**
- 4.1 Reading the Files . . . 13
- 4.2 Strategy-based Check . . . 13
- 4.3 Module to λ-term . . . 14
- 4.4 Comparison and Output . . . 16

5 Analysis **17**
- 5.1 Results . . . 17
- 5.2 Unexpected Observations . . . 17
  - 5.2.1 (Recursive) Let . . . 17
  - 5.2.2 Incomplete Programs . . . 19
- 5.3 Extending the Rule Set . . . 19
  - 5.3.1 Eta-Expansion . . . 19
  - 5.3.2 Miscellaneous . . . 20
- 5.4 Using the Extended Rule Set . . . 20

6 Conclusion **22**
- 6.1 Future Research . . . 22

A Assignment I **25**

1. Introduction

Ask-Elle is an Intelligent Tutoring System (ITS) for the domain of functional programming [3]. An ITS is computer software that is designed to provide immediate and customisable feedback to a user whilst performing a task, without the intervention of a human teacher.

Ask-Elle is an adaptable programming tutor, designed to help students learn the functional programming language Haskell by giving automated feedback [3]. Ask-Elle provides step-wise, interactive feedback to students while they solve programming exercises. A teacher specifies a model solution. Automatically generated hints and tips then guide the user towards this solution. To provide adequate feedback, a student program is compared with a model solution. If a student program can not be compared with a model solution, program properties are tested using QuickCheck, a lightweight tool for random testing of Haskell programs [2]. However, the feedback provided by QuickCheck is limited.

1.1 Ask-Elle in Action

Let us demonstrate the strengths and limitations of Ask-Elle's step-wise feedback with an example. The origin of this research is similar to previous research by Ochagavía [8]. As a result, this demonstration of Ask-Elle in action will resemble his, to some extent. We will take on the role of student and solve a simple exercise. While doing so, we conveniently let Ask-Elle guide our every step towards the solution. One of Ask-Elle's example functions is `dupli :: [a] -> [a]`, which duplicates every element in a list of arbitrary type `a`. Although Ask-Elle supports having multiple model solutions per exercise, assume there is only one specified model solution:

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

Figure 1.1 shows how we can use Ask-Elle's hints to make steps towards the specified model solution. We start with an empty program and fill in the *hole* (`?` represents a hole) every step, using the hint system. The process of filling in the holes is called *refinement*. Ask-Elle's hints consist of refinements that are automatically generated from the model solution [8]. As such, filling in the holes every step using Ask-Elle's hints will ultimately result in a solution that is equivalent to a model solution.

```
-- Let's start with an empty program (?  represents a hole)
?
-- Hint:  Introduce the function dupli.
dupli = ?
-- Hint:  Use the concatMap function.
dupli = concatMap ?
-- Hint:  Use the replicate function.
dupli = concatMap (replicate ?)
-- Hint:  Introduce the integer 2.
dupli = concatMap (replicate 2)
```

Figure 1.1: Using Ask-Elle's step-wise hints to solve an exercise.

Of course, using `concatMap` is just one of numerous solutions to the `dupli` exercise. Another, extensionally equivalent solution to the one in Figure 1.1, uses recursion:

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

Suppose we want to solve the exercise using recursion. Let us start with a non-empty program, by writing down the outline of a typical recursive structure. Imagine we are not sure how to continue and want to ask Ask-Elle for a hint. Sadly, Ask-Elle can not match our request. In this example, only the non-recursive model solution is provided to Ask-Elle. As a result, we can see in Figure 1.2 that Ask-Elle is unable to provide hints for our recursive approach. Providing a recursion-based model solution would in this case resolve the issue. In general, a naive approach to account for this lack of feedback is to supply Ask-Elle with different model solutions per exercise. However, this means a teacher has to specify many different solutions to account for all possible approaches. Taking into account variations in syntax, this is all but a feasible approach. We need a more dynamic solution. This is part of the focus of this thesis.

```
dupli []     = ?
dupli (x:xs) = ?
-- Feedback:  You have drifted from the strategy in such a way
that we can not help you any more.
```

Figure 1.2: No recursion-based model solution was provided to Ask-Elle. As a result, no refinements are available for this approach.

1.2 Program Unification

To assess a finished program, we need to determine whether it is equivalent to a model solution. One way to compare programs for equivalence is by means of program unification. Unification is not powerful enough to compare two seemingly different programs that have the same behaviour. In order to compare student solutions to model solutions, Ask-Elle uses *normalisation* to iron out the differences [4].

1.2.1 Normalisation

Normalisation is the process of rewriting a program to a more general, *normal form* by a series of semantics-preserving transformations. After transforming the programs to their corresponding normal forms, it attempts to unify them. If it succeeds in making a syntactical unification, we can safely conclude the programs are equivalent.

In practice, Ask-Elle fails in many cases to provide hints or to assess the correctness of a finished program. As Ochagavía noted in his research, previous research by Gerdes et al. [3, 4] shows that part of the problem lies in limitations of Ask-Elle's normalisation procedure [8]. Ochagavía then continues to show that Ask-Elle's normalisation algorithm can be extended to enhance its effectiveness in comparing solutions. I will discuss the results of his research in section 2.1. Normalisation is, however, only suitable when comparing complete programs. In this research we explore a different approach to program unification that can compare incomplete programs as well as complete programs.

1.2.2 Higher-Order Unification

A natural framework for solving program equality problems is higher-order unification. Higher-order unification is undecidable, but there exist restricted versions that have acceptable behavior in practice [9]. Serrano Mena et al. are developing an algorithm (from here on referred to as 'algorithm UM') that works by means of a decidable, restricted version of higher-order *unification* [9]. Unification is a goal-oriented algorithmic process, performing step-wise substitutions to unify expressions (see Chapter 2 for details). As a result, a unification-based algorithm is able to compare both incomplete and complete programs. As such, algorithm UM could prove more useful than the default, normalisation-based algorithm. By using algorithm UM to compare incomplete student solutions to (complete) model solutions, we can potentially improve the feedback system of Ask-Elle.

1.3 Relevance

The subject of program unification and equivalence is an interesting subject within the field of Computer Science and Artificial Intelligence. In practice, programs are usually checked by testing program properties or running predefined test-cases and edge-cases. Testing, however, can not guarantee that a program is flawless or that two programs are equivalent. Unification can guarantee equivalence, making it a rigid way of checking programs. Implementing a unification-based algorithm in Ask-Elle or any other ITS could prove fruitful in providing users with better and more reliable feedback.

1.4 Research Questions

In this Bachelor Thesis, I will contribute to the development of Ask-Elle, by focusing on the following research questions:

- How can we extend the algorithm UM to work on real student data in Ask-Elle?
- How does algorithm UM compare against earlier approaches to unifying incomplete programs?

The structure of this thesis is as follows: in Chapter 2 we first take a look at related research on Ask-Elle. We then cover the necessary background information on Ask-Elle's strategy generation, on higher-order unification and algorithm UM itself. Chapter 3 explains our research methodology. We discuss the pipeline and implementation in Chapter 4 and analyse the results of running data through the pipeline in Chapter 5. We summarize our research and offer suggestions for future research in Chapter 6.

2. Background Knowledge

In this Chapter, we first take a look at related research on Ask-Elle. We then cover the necessary background information on Ask-Elle's automatic strategy generation, as well as on higher-order unification and in particular algorithm UM. We observe how the two are similar and how this can be used to discover cases in which strategy checking and unification differ in their results.

2.1 Related Work

As briefly stated in Chapter 1, the origin of this research is similar to previous research by Ochagavía [8]. In practice, Ask-Elle fails in many cases to provide hints or to assess the correctness of a finished program. Previous research by Gerdes et al. [3, 4] shows that part of the problem lies in limitations of Ask-Elle's normalisation procedure [8]. Gerdes et al. suggest implementing additional semantics-preserving transformations. Using this idea, Ochagavía sets out to find situations in which Ask-Elle's normalisation algorithm does not work as expected. Using this information, he then extends Ask-Elle's normalisation algorithm with additional transformations to handle a wider range of programs. The results are very positive, but the interaction of the new transformations with Ask-Elle's stepwise feedback system is unclear. The concept of finding new transformations is similar to the intent of our research. As such, we can draw inspiration from the relevant transformations.

2.2 Strategies

As mentioned in Chapter 1, Ask-Elle's hints consist of refinements that are automatically generated from a model solution. Ask-Elle generates a *programming strategy* that specifies a sequence of steps that go from an empty program to a model solution [4]. Applying all steps in a sequence in order will ultimately result in a solution that is equivalent to a model solution. We can think of a strategy as a tree consisting of refinement steps. Figure 2.1 visualizes the generated strategy for the example function `dupli`, which was introduced in Chapter 1. At the root of the tree is an empty program, represented with a hole (a question mark). A branch consists of a refinement step towards a specified model solution. At each step in the tree, Ask-Elle is able to provide the user with feedback, as demonstrated in Figure 1.1. In the case of our example, branches `B` and `C` are empty, as only one model solution was specified.

2.2.1 Checking Against a Strategy

One way to check whether a student solution can be refined into a model solution is by checking it against a strategy. We traverse the strategy tree and see if the student solution is equivalent to one of the nodes. If it is, we have found the student solution's position within the tree. This means the student solution can be refined into a model solution and Ask-Elle is able to provide hints.

In order to check if a student solution is equivalent to one of the nodes in the tree, we need to apply program unification. As discussed in Chapter 1, Ask-Elle first uses normalisation, whereby a program is rewritten to a more general, normal form by

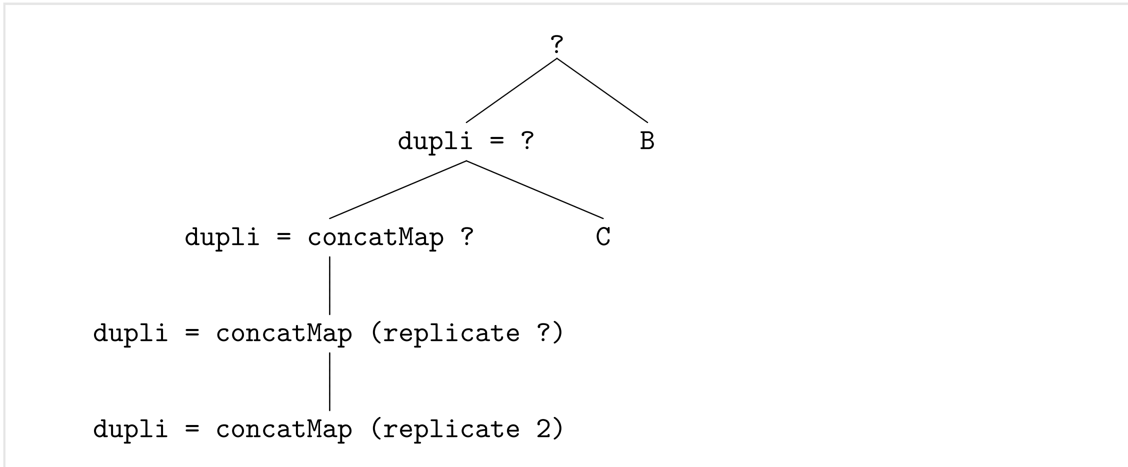

Figure 2.1: A strategy is automatically generated from a model solution and consists of refinement steps towards a model solution. `B` and `C` are potentially non-empty branches, depending on the specified model solution(s).

a series of semantics-preserving transformations. After both programs are transformed to their corresponding normal forms, unification is applied at a syntactical level. If both normal forms are syntactically equal, we have found a match. We can then access all transitions from the current node to the next node(s) to provide the user with feedback. Figure 2.2 shows a basic example of normalisation from previous research on the subject by Ochagavía [8]. In this example, an unnecessary anonymous function in the student solution is normalised by means of a semantics-preserving transformation. Both the model solution and the student solution are transformed to the same normal form.

```
-- Model solution               -- Student solution
double = map (* 2)              double = map (\x -> x * 2)
-- Normalised model solution    -- Normalised student solution
double = map ((*) 2)            double = map ((*) 2)
```

Figure 2.2: Unification after program normalisation. Both the model solution and the student solution are transformed to the same normal form using semantics-preserving transformations. Syntactical equivalence is obvious.

2.3 Higher-Order Unification

Unification is a goal-oriented algorithmic process, performing step-wise substitutions to unify expressions. In higher-order unification, an expression can contain higher-order variables called functions. In λ-calculus, function names are abstracted away, leaving just the structure of the function. We assume the reader to be familiar with the concept of Church's λ-calculus [1]. Higher-order unification is a process of finding a substitution σ, given two λ-terms λ_1 and λ_2, for the metavariables (also known as free or unbound variables) of the two terms. This substitution should be such that $\sigma(\lambda_1)$ is equivalent to $\sigma(\lambda_2)$ under the conversion rules of lambda calculus [10]. A λ-term t is defined by the following syntax:

$$
\begin{array}{rcll}
t & = & F & \text{(metavariable)} \\
 & | & x & \text{(variable)} \\
 & | & c & \text{(constant)} \\
 & | & \lambda x.t & \text{(abstraction)} \\
 & | & t_1 t_2 & \text{(application)}
\end{array}
$$

Application is left-associative, meaning `F x y` should be read as `((F x) y)`. A variable `x` is bound by abstraction : λ`x.x`. The above syntax technically allows for "loose" bound variables such as in λ`x.y`. However, such loose bound variables normally only occur at intermediate stages of a computation [7]. A metavariable (or free variable) is not bound by abstraction, which makes it suitable for substitution. With Chapter 1 in mind, a metavariable corresponds to a question mark (a hole) in a program. It's value is variable and yet to be determined.

Higher-order unification is undecidable, but for most practical applications we need not implement full higher-order unification [6]. There exist decidable, restricted versions that have acceptable behaviour in practice [9]. One of these restricted versions is pattern unification [7].

2.4 Algorithm UM

Pattern unification of λ-terms is at the core of Algorithm UM, developed by Serrano Mena et al. [9]. *Patterns* are λ-terms in which any occurrence of a metavariable `F` is only applied to unique variables. For instance, λ`x.`λ`y.F x y` is a pattern, because `x` and `y` are unique variables. Examples of non-patterns are `F c` (where `c` is a constant), λ`x.F x x` (application of `F` to non-unique variables), λ`x.F (F x)` (`F` is applied to non-variable `(F x)`) [7]. Algorithm UM implements an extended version of pattern unification. We will provide a concise overview of the extensions made. The technical details can be read in the corresponding research by Serrano Mena et al. [9].

2.4.1 Local Functions in λ-Terms

Programming languages based on λ-calculus, such as Haskell, include a notion of *local functions*. Local functions bind a name to (part of) a term. This allows for ease of reference and enhances legibility. In Haskell, local functions are defined using `let`. To accommodate for `let`, the syntax of λ-terms from the previous section is extended:

$$
\begin{array}{rcl}
t & = & ... \\
 & | & ... \\
 & | & let\ \overline{x = t_1}\ in\ t_2
\end{array}
$$

This new syntactic rule not only allows for local functions using `let`, but also for recursive `let`, in which the bodies of local functions refer to each other. This is analogous to the functionality of Haskell. The extended syntax is reflected in the implementation, by extending the pattern unification algorithms to support for recursive `let`. The algorithm is further extended to increase awareness of certain pragmatic equalities that are regularly found in practice. We call this pragmatics-aware unification.

2.4.2 Pragmatics-Aware Unification

Algorithm UM is extended to increase awareness of certain pragmatic equalities that are regularly found in practice. This allows a student a certain degree of freedom in constructing their program, without the algorithm being unable to recognize their solution. As a result, the algorithm can handle a wider range of programs. The pragmatic equalities recognized by the algorithm are the following:

1. Permutation of arguments in local functions. Consider the following λ-terms:

```
λp.λq.let f  = λx.λy.g x y  in  f  p q
λp.λq.let f' = λy.λx.g x y  in  f' p q
```

   We see that the above terms are extensionally equivalent, but the order of arguments in the local function is permuted. The algorithm handles permutation of arguments in local functions, if they have the same amount of arguments.

2. Constant arguments in local functions. (1) Covers equality for local functions that have the same amount of arguments. This, however, does not cover situations in which in one of the terms one of the local functions uses constant arguments. Consider the following λ-terms:

```
λg.λp.let f  = λx.λy.g x y  in  f  0 p
λg.λp.let f' =    λy.g 0 y  in  f'   p
```

   The terms are extensionally equivalent. The difference between `f` and `f'` is a result of inlining the constant argument `0`. Cases such as these are successfully handled by the algorithm.

3. Unification of terms with local bindings. In some cases we would like to unify two terms in which one of the terms refers to a local binding, but the other does not. Consider the following λ-terms:

```
λf.λx.f x
λf.λx.let f' = f x in f'
```

   We can see the terms are extensionally equivalent. The difference here lies in inlining the local function `f'`. The algorithm deals with cases in which one of the terms refers to a local binding, but the other does not.

2.4.3 Algorithm UM in Action

A successful run of algorithm UM means that the terms can be made extensionally equivalent via substitution [9]. Figure 2.3 shows the pragmatics-aware pattern unification as implemented in algorithm UM. For consistence, this example again uses the example function `dupli`, but now with a top-level `let`, to abide by the extended λ-term grammar. Note how the progress unfolds in a goal-directed way. Algorithm UM is able to find a substitution to make an incomplete student solution extensionally equivalent to a model solution.

When comparing complete solutions, we can have three different situations. The

first being that the two solutions are exactly equal in syntax. As such, no substitution is needed and a simple syntactical unification is sufficient. In the second case a student solution is incorrect and a unification by means of substitution can not be made. In a third case, a student solution is correct, but it's implementation is different from a specified model solution. In this case we can try to rewrite the solution using the available rewrite rules. Figure 2.4 shows how this works for our simple example function `dupli`. This does not mean that any two extensionally equivalent terms can be transformed into one another using unification. However, if two terms can be unified, we can be sure they are extensionally equivalent. Our test data for comparing complete solutions consists of strictly correct submissions (see section 3.1). This means a student solution is either equal or extensionally equivalent to a model solution.

```
λ1 = let dupli1 = (concatMap (replicate 2)) in dupli1
λ2 = let dupli2 = ?1 in dupli2

unify (λ1, λ2):
  -- call1:  ?1
  -- assumed eq:  <var dupli1, var dupli2>
  -- unify:  (?1, concatMap ?2)
  -- call2:  ?2
  -- unify:  (?2, replicate ?3)
  -- call3:  ?3
  -- unify:  (?3, 2)
  -- success
  -- exit3:  ?3 = 2
  -- exit2:  ?2 = replicate 2
  -- exit1:  ?1 = concatMap (replicate 2)
unification found:
σ= {<?3,2>, <?2,replicate 2>, <?1, concatMap (replicate 2)>,
assumed eq <var dupli1, var dupli2>}
```

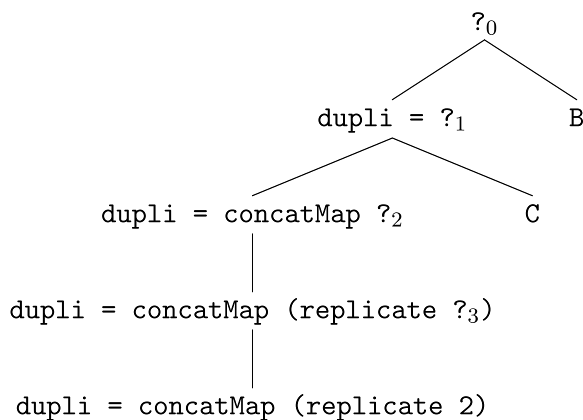

Figure 2.3: A unification is made between two λ-terms. There exists a sequence of step-wise substitutions from the student solution to a model solution. This means that the terms can be made extensionally equivalent.

```
λ3= let dupli3 xs = concatMap (replicate 2) xs in dupli3
λ4= let dupli4 xs = concat (map (replicate 2) xs) in dupli4

unify(λ3, λ4):
σ={λf.λxs.(concatMap f xs.concat (map f xs)) (replicate 2) (xs),
assumed eq <var dupli3, var dupli4>}
```

Figure 2.4: Unification on two λ-terms without metavariables. The terms can be made extensionally equivalent by introducing an anonymous function.

2.5 Similarities

In pragmatics-aware unification, a hole is represented as a metavariable. A metavariable can be substituted by other λ-terms. A successful run of algorithm UM means that two λ-terms can be made extensionally equivalent [9]. This means that there is a sequence of step-wise substitutions from the student solution to a model solution. This sequence results in a substitution for each metavariable in play.

As mentioned in section 2.2, a programming strategy specifies a sequence of steps that go from an empty program to a model solution. Applying all steps in a sequence in order will ultimately result in a solution that is equivalent to a model solution.

Both approaches thus give a sequence of steps from a student solution to a model solution, if such a sequence exists. We can see how this is useful for comparing the results of both approaches. If a strategy-based check results in a sequence of steps, a unification should be available too. We can use this to expose short-comings of the current version - and reveal potential benefits of the algorithm UM. Pragmatics-aware unification can be useful when comparing incomplete programs. Much like a strategy-based check, unification can be used to provide step-wise feedback. If we can extend the algorithm to handle a wider range of programs than the strategy-based approach, this could prove a fruitful new method indeed.

3. Methodology

In this chapter we explain our research methodology. As formulated in section 1.4, our aim is to extend the algorithm UM to work on real student data in Ask-Elle. In order to do this, we use a set of student data, consisting of student solutions to programming exercises in Haskell. We use the unification-based algorithm UM and extend this algorithm to process student data. We analyse the situations in which the algorithm can not completely process the student data and extend the algorithm UM accordingly. This will most likely involve adding cases for abstract syntax constructs that are not covered yet in the current algorithm, such as pattern matching.

To use the student data, we need to develop a pipeline to read and process student data in a meaningful way. This processing step is two-fold: for use of a strategy-based check and for use of algorithm UM. We then attempt to unify the student solutions with the corresponding model solutions. Figure 3.1 shows a concept of the pipeline.

We want to the compare results of algorithm UM on both complete and incomplete programs. By using complete programs, it is unnecessary to check the intermediate steps in a strategy tree. Instead, we compare a student program to the leaves of said tree using normalisation. Thus, by using complete programs, we are essentially comparing unification to normalisation. Previous research by Ochagavía [8] summarizes the results of Ask-Elle's naive normalisation on the student data. We compare the results of unification on complete programs to the results of Ask-Elle's naive normalisation. We compare the results of incomplete programs (programs with holes) for algorithm UM to the strategy-based check. We will discuss the implementation of the pipeline and the process of extending algorithm UM to include cases for abstract syntax constructs such as pattern matching in Chapter 4.

We analyse the sets of student and model programs to reveal shortcomings of the algorithm UM. Based on our findings, we can enhance the algorithm UM to resolve the discovered shortcomings.

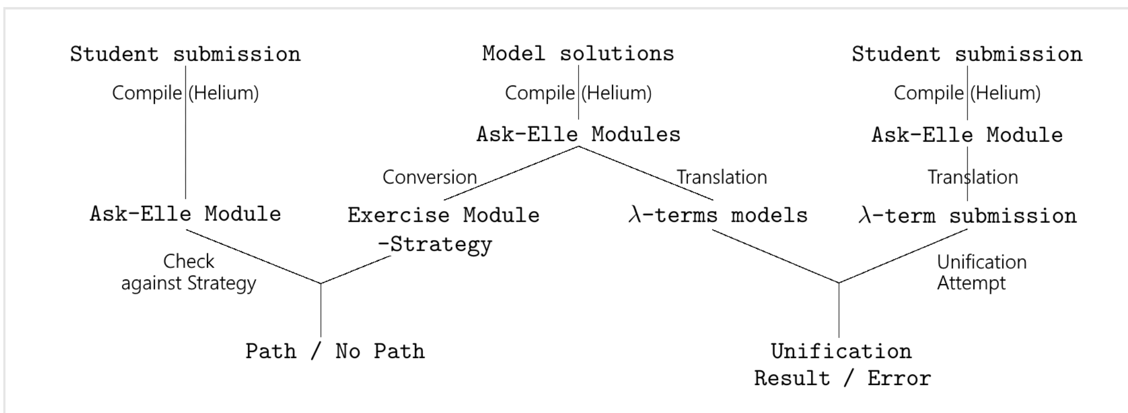

Figure 3.1: A concept of the pipeline. The student submission on the left and right is the same submission, but has been duplicated to make the comparison methods more apparent.

3.1 Student Data

We analyse the solutions to Assignment I of the Functional Programming (INFOFP) course at Utrecht University, during the academic year of 2017-2018. These solutions were all correct and participants gave consent for their solution to be used for research purposes. In the assignment, students are asked to implement 8 functions of growing complexity on the subject of lists. The functions in the assignment are labeled Exercise 1 to 8 for convenience. The document to Assignment I is included in Appendix A.

We use the model solutions as supplied by the teachers of the course. Most exercises have multiple model solutions to compare with student solutions. The model solutions are assumed to be generally better solutions than the student solutions. The data set is summarized in Table 3.1.

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

Table 3.1: Model solutions and correct submissions per exercise.

3.2 Analysis

In Chapter 5 we will determine the effectiveness of the current version of algorithm UM by reading in student data. We attempt to unify the student data with the corresponding model solutions. We measure the number of student solutions that can be unified per exercise. To identify the shortcomings of the algorithm, we examine the unification error and analyse the student data to look for a pattern. Based on our findings, we can enhance the algorithm by introducing new unification rules. An example of such a rule can be seen applied in Figure 2.4, where the function `concatMap` is rewritten in terms of `concat` and `map`. We measure the effect of the extended rule set by running the student data through the pipeline once more.

4. The Pipeline

In this chapter we describe the construction of the pipeline (as shown in Figure 3.1) which allows us to read and process student data in a meaningful way. This processing step is two-fold: for use of the strategy-based check and for use of the new algorithm UM. The latter will require us to make a complete translation from Ask-Elle's abstract syntax trees (which we shall call *modules* from now on) to the λ-term used by the unification algorithm. This will involve adding cases for abstract syntax constructs such as pattern matching. We describe how we compare each student solution to the corresponding model solutions and how we output the results.

4.1 Reading the Files

To compare student solutions to model solutions, we first need to set up a system to read the files from a specified directory and compile the solutions to *modules* using *Helium*. A module is an abstract representation of a program. Helium is a compiler for Haskell. Compiling the solutions ensures they are type correct. This way, we can automatically exclude faulty 'solutions'. *Type signatures* are an explicit way of describing the input and output type of a function. In most cases, Haskell does not need such an explicit description, because it uses type inference. Although including type signatures for functions is considered good practice, in our case it will result in compiler errors. As such, type signatures should be excluded from all models and submissions. For this reason, we filter every solution to ignore included type signatures before compiling.

To run an exercise through the pipeline, the user conveniently supplies the system with an exercise number via the command line. All solutions in both the model and student solution directory corresponding to the exercise number are then read and compiled. To successfully compile the solutions and generate a strategy tree from the model solutions, we need to supply the compiler with a custom set of *imports*. An import is a tuple, consisting of a function or data type and its corresponding type signature. This set of imports should contain anything that occurs in the solutions and is not included in Haskell's standard library *Prelude*. An example of such an import is the function `isDigit`, which is used to indicate whether a given character is a digit:

```
("isDigit", "Char -> Bool")
```

Some of the exercises in the data set also make use of custom data types and functions. We need to include these in the set of additional imports. Without these imports, compilation would result in errors and automatic strategy generation would incorrectly assume the custom data types and function names to be variable, resulting in false positives when checking against a strategy.

4.2 Strategy-based Check

After compiling the model solutions of an exercise to modules using Helium, we would like to generate the corresponding strategy tree. In order to do this, we need to convert

the module into Ask-Elle's `Exercise Module`. This will generate the strategy tree using the specified model solutions. To check a student solution against a strategy, we need to implement a way to traverse the strategy tree and compare the solution with each node. To traverse the tree, we have implemented depth-first search. Breadth-first search would be more efficient when using incomplete student programs, but implementing it in Haskell would be much more involved and beyond the scope of this research. In order to compare the student solution to one of the nodes, we apply some normalisation. If we find the student solution on our way down, we return the path up until that point. If we complete traversing the tree without finding a match, we return that no path can be found.

4.3 Module to λ-term

To use Algorithm UM, we first need to make a translation from module to λ-term. This will require some technical gymnastics, because making a suitable translation for some abstract syntax constructs is less than straight forward. Haskell is an expressive language and λ-syntax is limited, even after extending it with a rule for `let`, as introduced in subsection 2.4.1. We will cover the key aspects of making the translation.

A function's arguments are explicitly stated. As a result, modules are of the form: `function x y = ...` To abide by λ-term syntax, we need to rewrite function arguments into λ-abstractions inside the function and ultimately introduce a top-level `let`. The final result should look something like this:

```
let function = \x y -> ..  in function
```

We start by pre-processing all modules. This includes rewriting function bindings to a pattern binding with a *case*, as shown in Figure 4.1. A `case` is a way of introducing alternatives, based on the value of an expression. We will see how this is a relevant step when we discuss pattern matching. After pre-processing we can start translating the module to a λ-term.

```
-- Before pre-processing
f x      = g
f (x:xs) = h
```

```
-- After pre-processing
f = \y -> case y of
               x      -> g
               (x:xs) -> h
```

Figure 4.1: Rewriting function bindings to a pattern binding with a `case`.

Figure 4.2 shows two ways of defining a local function in Haskell. One way is to use `let`, which defines a local function prior to it's use. Converting a local `let`-binding using our new λ-syntax `let`-rule is straightforward.

Another way to define a local function is using *where*. The `where`-construct can be used to define a local function after it's use. When we define a local function using `let`, declarations are made before the expression in which they are used. In a local function using `where`, the expression is first used and the corresponding declarations are made afterwards. We can translate a local function using `where` to `let` by compensating for the order of declarations and expression. As such, extending the λ-syntax to include a syntactic rule for `where` is unnecessary.

Pattern matching is at the core of Haskell. As a result, there are many different ways to match on a pattern. To process real student data, we need to deal with pattern matching. To translate pattern matching of any shape and form to a λ-term, we need to

```
-- Defining a local function f using let    -- Abstract form
let f = concatMap (replicate 2)             let declarations
in  dupli = f                               in  expression

-- Defining a local function f using where  -- Abstract form
dupli = f                                   expression
  where f = concatMap (replicate 2)           where declarations
```

Figure 4.2: In Haskell, local functions can be defined using `let` and `where`. We convert one into the other by compensating for the order of declarations and the expression in which they are used

improvise. The easiest way to deal with pattern matching is to assume there are some 'magic functions' which perform the pattern matching. These 'magic functions' are *folds* of the corresponding data types. A fold is a higher-order function used to process a data structure in some systematic way and return a value.

A clever way to ensure we apply the right fold, is by generating it from the underlying pattern. We obtain a fake name from the names of the patterns in a pattern match and introduce as many arguments as branches. In each branch we introduce as many λ-abstractions as variables bound in the corresponding pattern. This way we can translate a pattern match to a function application of the generated fold to the corresponding arguments. We don't need to define the functionality of the folds. As long as the way of obtaining the fake fold name is consistent, we can compare their results. This same idea applies for tuples. A similar idea is applied to create folds for other constructs such as `if-then-else` and `guards`. Figure 4.3 shows some examples of how we obtain a fake fold name from the names of the patterns in a pattern match and apply it to the corresponding arguments. The figure also shows why pre-processing the modules was a good idea.

```
-- Pattern match on a list          -- Generated fold application
case e of []     -> f               case-[]-(:)  (e f, \x xs -> g)
          (x:xs) -> g

-- Pattern match on a tuple         -- Generated fold application
case (x,y) of                       case-t-[]-(:)  (x, y,
          ([],ys) -> f                              \ys    -> f,
          (xs,ys) -> g                              \xs ys -> g)

-- Pattern match on if              -- Generated fold application
if e then f                         if (e f g)
     else g

-- Pattern match on guards          -- Generated fold application
...  | e         = g                guards (g-if (e g),
     | otherwise = h                        g-if (otherwise h))
```

Figure 4.3: This figure shows examples of how we obtain a fake fold name from the names of a pattern in a pattern match and apply it to the corresponding arguments.

As mentioned earlier, every program is ultimately translated to a top-level let. As a final step to the translation, we post-process the newly created λ-term by finding all of it's free variables (variables that are not bound by λ-abstraction) and substituting them with constants of the same name. This ensures that all operators and functions, including our generated folds, are constant, which is necessary for the unification process.

4.4 Comparison and Output

Processing a student solution is done in two steps. First we check the student solution against the automatically generated strategy tree as explained in section 4.2. If the student solution matches one of the nodes, we return the corresponding path. If we have traversed the whole tree without finding a match, we return no path was found.

We then translate the student solution and corresponding model solutions to λ-terms. We use algorithm UM on these λ-terms and attempt to unify the student solution with any of the available model solutions. We do not need a student solution to unify with every available model solution. If a unification is found between a student solution and at least one of the corresponding model solutions, unification was a success. We return the unification rule(s) that resulted in successful unification. If no unification can be found between a student solution and a model solution, we return the λ-terms and the unification error. All output per student solution is written to both the command line interface and exercise-specific output files. We can use the latter to store the results and inspect them at a later time.

We now know for every student solution whether a match was found using the strategy-based check and whether a unification was found using algorithm UM. Finally, we count the number of successful matches using the strategy-based check and the number of successful unifications for each exercise. We write these results and the total amount of submissions and available model solutions to the command line interface and the output files. This allows us to compare the results.

5. Analysis

In this chapter we present the results of running student data through the pipeline. We compare the results to Ask-Elle's normalisation [8]. We discuss and analyse unexpected observations. We analyse the student data to compose new unification rules to enhance algorithm UM. We measure the effect of the new rules by running the pipeline on the student data using an extended set of unification rules.

5.1 Results

Table 5.1 shows the results of running the student data through the pipeline using the current version of algorithm UM compared to Ask-Elle's normalisation. The number of successful unifications was expected to be similar to the results of naive normalisation. For some exercises this number is much lower than expected. Upon inspection of the attempted unifications, we made some unexpected observations that could explain this discrepancy.

Exercise	Models	Submissions	Unification	Normalisation
1	1	110	75	73
2	6	111	12	36
3	3	103	14	52
4	4	108	58	62
5	1	109	10	22
6	2	101	2	4
7	6	99	8	14
8	5	100	2	23

Table 5.1: The results of running the student data through the pipeline using algorithm UM compared to Ask-Elle's normalisation.

5.2 Unexpected Observations

In this section we will address some unexpected issues with (recursive) `let` and our strategy-based check. These issues limit our results in multiple ways.

5.2.1 (Recursive) Let

Figure 5.1 shows a surprising result. Remember that a local function using `where` is converted to a `let`. The difference between the model solution and the student solution then lies in inlining the local function `parseRow`. As explained in subsection 2.4.2, we should be able to unify a student solution using a local function with a model solution without said local function. The unification error could be due to an oversight in our translation from module to λ-term.

When attempting to unify two recursive (complete) solutions that are equal, a unification can be found. However, when attempting to unify recursive (complete) solutions

```
-- Model solution                 -- Student solution
parseTable = map words            parseTable = map parseRow
                                    where parseRow = words

> Unification not found.
```

Figure 5.1: No unification can be found between a student solution which refers to a local binding and a model solution which does not.

that are not exactly equal, no unification can be found. Figure 5.2 shows that no unification can be found even though the solutions are equivalent (the unification rule applied to rewrite the function `replicate` is discussed in section 5.3). However, when we remove the recursive call, a unification can be found. This indicates an issue with recursive let. This unification error could be due to an oversight in the translation from module to λ-term. It could also be due to an oversight in the implementation of recursive let in algorithm UM. This issue has an unfortunate impact on recursive solutions.

Using recursion and defining local functions using `where` is common practice in Haskell. As such, the observed unification errors could account for an unexpected low number of successful unifications.

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

-- Student solution
printLine []     = "+"
printLine (x:xs) = "+" ++ take x (cycle "-") ++ printLine xs

> Unification not found.

-- Replacing the variable argument 'xs' with a constant list
'[1,2,3]' in the recursive call of printLine for both solutions:
[...]
[...]  ++ printLine [1,2,3]

> Unification not found.

-- Replacing the recursive call of printLine with a constant
string 'test' for both solutions:
[...]
[...]  ++ "test"

> Unification found:  replicate n xs -> take n (cycle xs)
```

Figure 5.2: When two recursive and complete solutions using recursive let are not exactly equal, no unification can be found.

5.2.2 Incomplete Programs

This research helped discover a minor issue with Ask-Elle's strategy diagnosis. For any two (exactly) equal programs, no path would be returned. This would suggest that the student solution did not match any of the nodes in the strategy tree. However, we know the student solution should match one of the model solutions and a path of some sort should be available. This issue has since been resolved and equal programs now return a path.

As part of the pipeline, we wrote a strategy-based check using Ask-Elle's strategy diagnosis. This strategy-based check would allow us to compare algorithm UM to (some form of) normalisation for both incomplete and complete solutions. We search the strategy tree using depth-first search. The strategy-based check implements some normalisation to find a matching node, as discussed in section 2.2. However, when running the pipeline on student data, the search has a tendency to run indefinitely. This suggests that the search space is too extensive or our search gets stuck in an infinite loop. The problem only occurs when a strategy is generated using more than one model solution. A loop could be caused by potential cycles in the strategy tree. It could also be that the same part of the tree is processed over and over, because of the normalisation applied to the nodes while traversing the tree. In an attempt to fix the problem, we limited the depth of search. This showed some result, but did not get rid of the problem entirely. We also checked if the current node had already been processed before adding their children to the search. This should prevent repetition. Unfortunately, this did not change performance. Further effort to fix the strategy-based check did not fit within the scope of our research. As such, we can not currently use the strategy-based check to compare results for incomplete programs. This means our second research question as mentioned in section 1.4 will remain unanswered.

5.3 Extending the Rule Set

In this section we analyse the student data and unification results to compose new unification rules to enhance algorithm UM. We will use these rules to extend the unification rule set. We can then use the extended rule set and run the pipeline on the student data, the results of which are covered in the next section.

5.3.1 Eta-Expansion

When a function takes multiple arguments, these arguments need to be explicitly stated in the function definition. When a function takes a single argument, Haskell allows this argument to be implicit. For instance, we can define our example function `dupli` in one of two ways:

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

Remember from Chapter 1 that the type signature of `dupli` is `[a] -> [a]`. The expected argument is a list of type `a`. This argument is made explicit in version 2 by writing a variable `xs`. The lack of explicit argument in version 1 was already covered by one of the unification rules in algorithm UM. This rule applied *eta-expansion*. Using eta-expansion, we introduce a λ-abstraction on a fresh variable, rewriting a function `f` into λ`x.f x`.

Preprocessing a module sometimes leads to an unnecessary extra layer that uses a fresh variable abstraction and a corresponding `case` construct for a single alternative. We can see how this looks if we ignore the second argument in Figure 4.1. This redundant

case limits the use of the original eta-expansion rule. To compensate for a redundant case, we implemented a new version of the eta-expansion rule. Another way to solve this issue is to adjust the module to λ-term translation to get rid of the redundant case. Ideally, we adjust pre-processing to prevent introducing a redundant case in the first place.

5.3.2 Miscellaneous

Based on the unification results and the student data, we identified a number of unification rules. Most of these rules are implementing Haskell syntactical or functional equivalence. We can use these rules in an attempt to rewrite a model solution into a student solution. By doing so, we assume the model solution to be the better solution. Table 5.2 shows the unification rules that we found. These rules are combined with the default unification rules to form an extended rule set. This set is by no means exhaustive, but should suffice to show some improvement in the number of successful unifications.

Rewrite	Into
`f`	`λy.((case-t y) (λx.f x))`
`intercalate f xs`	`concat (intersperse f xs)`
`map (f . g)`	`map f . map g`
`flip f a b`	`f b a`
`replicate n xs`	`take n (cycle xs)`
`list a`	`"a"`
`foldr (++) []`	`concat`
`case-t-[]-(:) (a, b, c)`	`case-t-(:)-[] (a, c, b)`

Table 5.2: An extension of the unification rule set. The listed rules were found by analysing the unification results and the student data.

5.4 Using the Extended Rule Set

We run the pipeline on the student data using the extended unification rule set for algorithm UM. Table 5.3 shows that extending the unification rule set increased the effectiveness of algorithm UM for exercise 1 and 2. The results for unification on exercise 1 are now convincingly superior to normalisation.

Exercise	Models	Submissions	Default	Extended
1	1	110	75	88
2	6	111	12	14
3	3	103	14	14
4	4	108	58	58
5	1	109	10	10
6	2	101	2	2
7	6	99	8	8
8	5	100	2	2

Table 5.3: The results of running the student data through the pipeline using the extended rule set for algorithm UM.

Figure 5.3 shows the overall results of running the student data through the pipeline. Extending the rule set shows an increase in effectiveness for algorithm UM, but is in general not sufficient to effectively close the gap on normalisation. The difference between normalisation and unification is presumably due to the flaws mentioned in section 5.2. If we can fix these issues, we can expect to see an overall increase in effectiveness for unification.

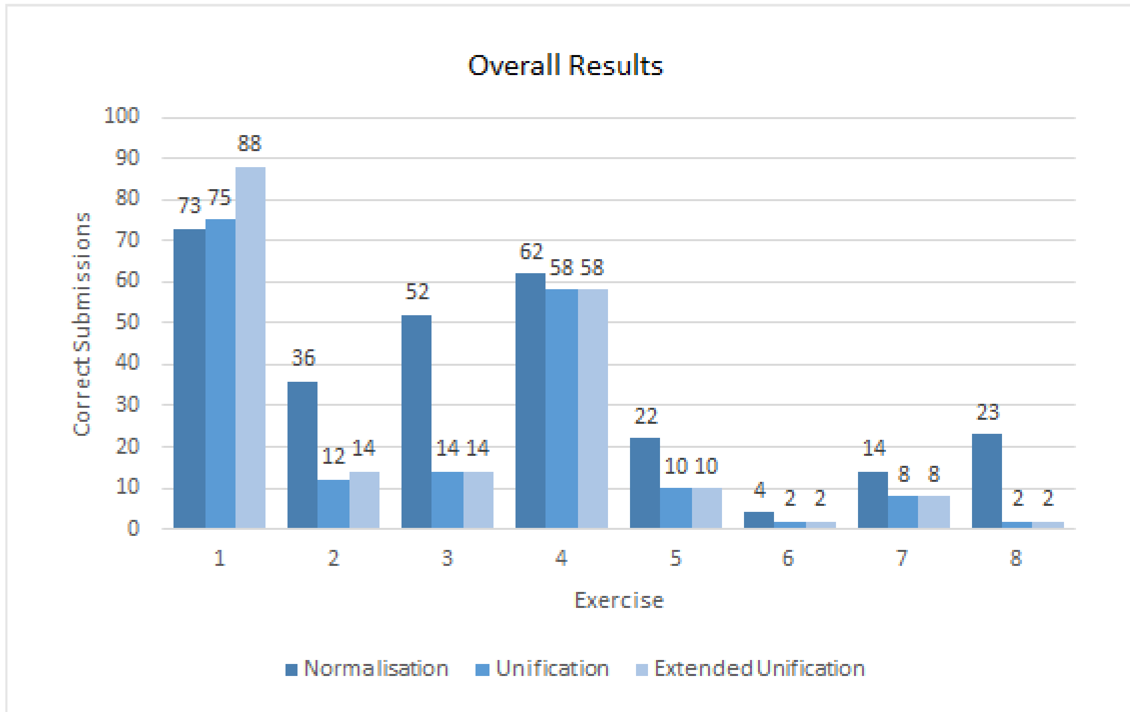

Figure 5.3: The overall results of running the student data through the pipeline using unification with the default rule set and unification with the extended rule set. These results are compared to Ask-Elle's normalisation.

6. Conclusion

As mentioned in section 1.4, the aim of our research was to extend algorithm UM to work on real student data in Ask-Elle. We used a set of student data, consisting of student solutions to programming exercises in Haskell. We set up a pipeline to read and process student data in a meaningful way. This processing step was two-fold; for use of a strategy-based check and for use of algorithm UM.

We used algorithm UM and analysed situations in which it could not completely process student data and extended the algorithm accordingly. This involved adding cases for abstract syntax constructs such as pattern matching. We constructed a dynamic and consistent way of generating folds for the relevant abstract syntax constructs. We wrote a complete translation from module to λ-term. We discovered a minor issue with Ask-Elle's strategy diagnosis which has since been fixed.

We ran the pipeline on the student data and encountered an issue with the implementation of the strategy-based check which we could not resolve within the scope of this research. This meant we could only compare results for complete programs, leaving our second research question of how algorithm UM compares against earlier approaches to unifying incomplete programs unanswered. The unification results were lower than expected when compared to Ask-Elle's naive normalisation. We analysed the unification errors and student data and made some unexpected observations. These observations included flaws we think originate from either our translation from module to λ-term or algorithm UM itself.

We analysed the student data and identified a number of unification rules. We combined these rules with the default rules of algorithm UM to create an extended unification rule set. We ran the pipeline on the student data using said extended rule set. We measured an increased number of successful unifications when using the extended rule set for algorithm UM.

6.1 Future Research

As mentioned above, there are still some kinks in the pipeline. It would be beneficial to pinpoint and resolve the observed unification issues involving (recursive) let. These issues most likely originate from our translation from module to λ-term or possibly from the inner workings of algorithm UM itself. Resolving these unexpected unification errors should result in an increase in overall effectiveness of algorithm UM. A next step is then to analyse the unification results and student data to find new unification rules to further extend algorithm UM.

Ask-Elle's feedback system can potentially be improved by using algorithm UM to compare incomplete student solutions to (complete) model solutions. To research this possibility, a strategy-based check was implemented into the pipeline. As explained in section 2.5, this strategy-based check is similar to the unification-based approach and can be used to compare results for incomplete student solutions. Alas, the strategy-based check could not be made fully functional within the scope of this research. This meant there were no results for earlier approaches to unifying incomplete programs to compare with. As such, the second research question in section 1.4 remains unanswered. Future research should resolve the current issue in the implementation of the strategy-based check. A

functional strategy-based check would unlock the full potential of the pipeline: to compare the results of algorithm UM to the strategy-based check on both incomplete and complete solutions. If algorithm UM can be extended to handle a wider range of both incomplete and complete programs than, respectively, the strategy-based approach and Ask-Elle's naive normalisation, it could prove a fruitful new method indeed.

Bibliography

[1] Alonzo Church. *The calculi of lambda-conversion.* Princeton University Press, 1941.

[2] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. *Acm sigplan notices*, 46(4):53–64, 2011.

[3] 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.

[4] 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*, SIGCSE '10, pages 441–445, New York, NY, USA, 2010. ACM.

[5] Ruud Koot. Functional programming 2017/2018, assignment 1: Lists. Available at `http://www.staff.science.uu.nl/~f100183/fp/practicals/Assignment1.pdf`. Accessed 1 June 2019.

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

[7] Tobias Nipkow. Functional unification of higher-order patterns. In *[1993] Proceedings Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 64–74. IEEE, 1993.

[8] Adolfo J. Ochagavía Hubner. Improved normalization for ask-elle through semantics-preserving transformations. Master's thesis, 2018.

[9] Alejandro Serrano Mena and Johan Jeuring. Pragmatic unification of λ-terms with recursive let (unpublished).

[10] Wayne Snyder. *Higher Order Unification*, pages 123–153. Birkhäuser Boston, Boston, MA, 1991.

A. Assignment I

This thesis makes use of a data set consisting of student submissions to Assignment I, as described in section 3.1. The assignment document [5] is included as an appendix and can be viewed on the following pages.

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:

```
type Field = String
type Row   = [Field]
type Table = [Row]
```

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 parseTable* :: [*String*] → *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, `"Alice"`), (6, `"Allen"`), (6, `"female"`), (6, `"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* → *Field* → *Table* → *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*] → *Table* → *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:

$$
\begin{aligned}
&\mathit{exercise} :: [\mathit{String}] \to [\mathit{String}] \\
&\mathit{exercise} = \mathit{printTable} \\
&\qquad \circ\ \mathit{project}\ [\texttt{"last"}, \texttt{"first"}, \texttt{"salary"}] \\
&\qquad \circ\ \mathit{select}\ \texttt{"gender"}\ \texttt{"male"} \\
&\qquad \circ\ \mathit{parseTable}
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathit{main} :: \mathit{IO}\ () \\
&\mathit{main} = \mathit{interact}\ (\mathit{unlines} \circ \mathit{exercise} \circ \mathit{lines})
\end{aligned}
$$