MASTER'S THESIS

A Haskell Recursion Tutor

Nicasi, K (Kevin)

Award date:
2020

Link to publication

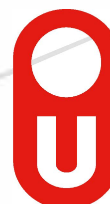
General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us at:
pure-support@ou.nl
providing details and we will investigate your claim.

Downloaded from https://research.ou.nl/ on date: 09. Sep. 2021

Open Universiteit
www.ou.nl



A HASKELL RECURSION TUTOR

Master Thesis

Software Engineering

Open University of the Netherlands

Faculty of Science

Author:	Kevin Nicasi	
Student number:		
Course code:	IM9906	
Presentation date:	November 2, 2020	
Thesis committee:	prof. dr. Johan Jeuring (chairman),	Open University of the Netherlands
	dr. Bastiaan Heeren (supervisor),	Open University of the Netherlands

Open Universiteit
de beste!
www.ou.nl

ABSTRACT

Recursion is not an easy subject to learn. In this thesis the literature is examined on the common difficulties and misconceptions students encounter when trying to understand recursion in the functional paradigm, and on the available tools for teaching and visualizing recursion. It is found that experts possess the copies model as their mental model and that novices often hold misconceptions about the passive flow and have difficulties identifying a correct base case or recursive step. The examined intelligent tutoring systems and visualizations show a variety of different approaches to illustrate code execution: some use graphical representations while others are text-based. Research indicates that some of the tools encourage students to spend more time with the learning material which leads to greater learning gains.

A prototype tool is proposed that supports the learning of recursion in Haskell through the step-wise evaluation of recursive functions and the displaying of the subsequent function calls in function tables. The prototype is aimed at alleviating the identified difficulties and misconceptions. The copies model is illustrated through the use of function tables that display the different instantiations of the function in the table rows. The passive flow is shown through step-wise normalization of the rows in the tables. The prototype has a unique way of illustrating the copies model and displays the passive flow explicitly. In the future the prototype could be augmented with additional features such as the handling of input and feedback.

Acknowledgments

First and foremost I would like my supervisor Bastiaan Heeren for his valuable input and inspiration. This thesis would not have been possible without his guidance. I would also like to thank Johan Jeuring for his valuable input and critical comments.

I would also like to thank my girlfriend Beatrix for her patience, love and enduring support.

Contents

1 Introduction . . . 9
1.1 Contribution . . . 10
1.2 Thesis overview . . . 12
2 Related Work . . . 13
2.1 Recursion in Haskell . . . 13
2.2 Evaluation in Haskell . . . 14
2.3 Mental Models and Conceptual Models . . . 15
2.4 The Notional Machine . . . 18
2.5 Intelligent Tutor Systems . . . 20
2.6 The Haskell Expression Evaluator and The IDEAS framework . . . 23
2.7 Literature reviews . . . 23
3 Research Design . . . 25
3.1 Research questions . . . 25
3.2 Research method . . . 25
4 Difficulties and misconceptions about recursion . . . 27
4.1 Mental models of recursion by Kahney . . . 27
4.2 Mental models of recursion by Götschi et al.. . . . 28
4.3 Limitations of the classification of mental models . . . 30
4.4 Difficulties and misconceptions found by Hamouda et al. . . . 31
4.5 Phenomenographic perspective by Booth . . . 32
4.6 Pedagogical suggestions . . . 32
4.7 Summary . . . 33
5 How do current educational tools illustrate recursion? . . . 35
5.1 Visualizations tools . . . 36
5.2 Tutoring Systems . . . 38
5.3 Level of Engagement . . . 43
5.4 Effectiveness . . . 44
5.5 Summary . . . 45

6 A tutor for recursion in Haskell 47
6.1 Using function tables to illustrate rewrite steps 47
6.2 Architecture of the prototype 48
6.3 Using algebraic data types to define and evaluate expressions 48
6.4 Walk-through 53
6.5 Getting the sub-expression form a path. 54
6.6 Summary 57
7 Discussion 59
7.1 Comparison with WinHIPE and HEE 59
7.2 Limitations and Future Work 63
8 Conclusions 69
References 71

1. INTRODUCTION

Learning to program is challenging (Carter & Jenkins, 1999; Lahtinen, Ala-Mutka, & Järvinen, 2005). Introductory courses on programming often fail to deliver: a lot of students struggle to produce a working program after a first course, independent of the institution (Venables, Tan, & Lister, 2009; Guzdial, 2011).

One of the hardest subjects to learn is recursion (Roberts, 1986; Gal-Ezer & Harel, 1998; Lahtinen et al., 2005). Students often struggle when they encounter the concept of recursion for the first time, it is a difficult concept to grasp. A lot of research in the field of computing education research has been done concerning the misconceptions and difficulties students run into when learning recursion (Kahney, 1983; Wu, Dale, & Bethel, 1998; Du Boulay, O'Shea, & Monk, 1999; Götschi, Sanders, & Galpin, 2003; AlZoubi, Fossati, Di Eugenio, & Green, 2014).

Recursion is an important programming technique (McCracken, 1987). Several algorithms, such as a search in a binary tree, are recursive in nature and are thus coded efficiently using recursion. Recursion is also used as an iteration construct in functional programming languages such as Haskell and Scheme.

To aid novices in the understanding of code execution several intelligent tutoring systems (ITS) and program visualization tools have been introduced over the past decades (Sorva, 2012). Research suggests that using ITS for teaching can be very effective (VanLehn, 2011). By using an ITS a student can study independently requiring less help from a teacher (Odekirk-Hash & Zachary, 2001). The systems are often able to give immediate feedback, which is not always possible in a classroom setting. Research suggests that immediate feedback is preferable to delayed feedback (Mory, 2004).

Some of the proposed tools have been targeted at recursion while others have a broader focus. Most of these tools are focused on the object-oriented paradigm and only a few are focused on the functional paradigm. None of those are specifically designed for teaching recursion.

The focus of this thesis lies within the subject of computing education: the teaching of recursion in the functional programming language Haskell. The goal of this research is to document what has been done by the community of programming educators on recursion and to design a prototype for an intelligent tutoring system that helps in the understanding of recursion by leveraging the widespread familiarity with mathematical function tables.

Function tables are a widely used visualization method in mathematics education (Martinez & Brizuela, 2006). Because of its widespread use it is an promising notation to leverage for illustrating function application, particularly for the demonstration of recursion.

To document the community's effort the available literature is studied focusing on two subjects: the difficulties encountered by students when introduced to recursion and the developed ITS and program visualization tools for teaching recursion. The output of the literature studies is used as inspiration for the prototype.

The tutor prototype is aimed at novices learning the functional language Haskell. The

basic method of computation in Haskell is function application (Hutton, 2017). Function evaluation can be viewed as a form of term rewriting (Pareja-Flores, Urquiza-Fuentes, & Velázquez-Iturbide, 2007). Several functional program visualizations tools illustrate the process of function evaluation as repeated term rewriting. The proposed prototype also follows this approach, but focuses on recursion and additionally displays function invocations and results in function tables.

1.1. CONTRIBUTION

The research proposed in this thesis makes a step towards helping students, who are new to Haskell, to understand recursion. The main research question we aim to answer is: How can a tutoring environment support students in understanding the evaluation of recursive functions in Haskell?

The contributions of this thesis are the summary and discussion of the findings of the community on the difficulties and misconceptions of novices concerning recursion in the functional paradigm, the compilation of an overview of tools that help in understanding recursion, and the design of a tutoring environment aimed at teaching recursion and that leverages function tables.

The tutoring environment calculates and displays evaluation steps of simple recursive functions and is aimed at novices. To our knowledge there is no tool that uses function tables to illustrate recursion. Figure 1 shows a screenshot of the tutor.

Tools such as the Haskell Expression Evaluator (Olmer, Heeren, & Jeuring, 2014) and WinHIPE (Pareja-Flores et al., 2007) also show evaluation steps and can be used to illustrate recursion, but lack the ability to display function tables. The Haskell Expression Evaluator focuses on higher-order functions defined in the Haskell Prelude, such as foldl and map, instead of explicit recursive functions that are typically covered before the more generalized functions in an introductory functional programming course (Lipovaca, 2011; Hutton, 2017).

Haskell Recursion Tutor

Choose an exercise:

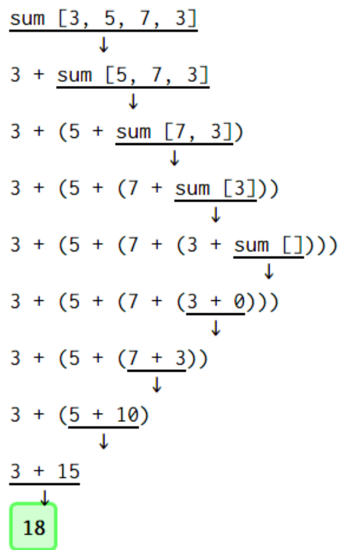
sum [3, 5, 7, 3]

Given are following function definitions:

```
sum []     = 0
sum (x:xs) = x + sum xs
```

View the rewrite steps below:

10/14



sum	
sum [3, 5, 7, 3]	3 + sum [5, 7, 3]
sum [5, 7, 3]	5 + sum [7, 3]
sum [7, 3]	7 + sum [3]
sum [3]	3 + sum []
sum []	0

add	
3 + 0	3
7 + 3	10
5 + 10	15
3 + 15	18

Figure 1: The proposed tutor prototype illustrating a call to the sum function. On the left the rewrite steps are shown. The expression that is rewritten is underlined. Next to it are the function tables. At this stage the table rows are not yet normalized, this can be done by the student by clicking the right pointing arrow.

1.2. THESIS OVERVIEW

In the next section, Section 2, we look at background concepts that are used in this thesis. Related work that has been done on intelligent tutoring systems, program visualizations, the Haskell Expression Evaluator and the IDEAS framework is discussed. In this section we also summarize some relevant literature reviews. The main research question is introduced in Section 3 and answered in Section 4, 5 and 6. Section 7 discusses the proposed prototype, compares it to two other tools, examines its limitations and identifies possible future work. The last section summarizes the answers to the research questions.

2. RELATED WORK

In this section we present related work and background concepts that are relevant to this thesis. First, we will give a brief introduction of recursion and evaluation order in Haskell. Secondly, we touch upon the concepts of mental and conceptual models proposed by cognitive psychology, identify some conceptual models for recursion and relate these concepts to the concept of the notional machine and talk about program visualizations. Next, some general properties of intelligent tutor systems, the Haskell Expression Evaluator and the IDEAS framework are briefly discussed. We conclude this section with an overview of some literature reviews that have been done on program visualizations and intelligent tutor systems.

2.1. RECURSION IN HASKELL

Recursion is the basic mechanism for coding iteration in Haskell (Hutton, 2017). A loop statement with an incrementing counter variable, a common statement in imperative languages, does not exist making recursion an essential programming technique.

The production of a new list from an existing list where from every element in the old list a new value is calculated by a function, or what is called a map, is one occasion where recursion can be used. Code Listing 1 shows an example of a recursive function definition, a function that squares all elements in a list of numbers. On line 1 is the function signature, which should be interpreted as follows: the function takes a list of integers and outputs a list of integers. Line 2 contains the base case of the recursion: the squares function called on an empty list returns an empty list. On line 3 is the recursive step. This line uses the colon for pattern matching: a list is matched against a pattern and deconstructed into its head (which gets assigned to x) and the tail (which gets assigned to xs). Line 3 should be interpreted as follows: to square all elements of a non-empty list, take the square of the first element of the list and use this as the head of the list, then apply the same actions on the remaining tail of the list. In short, the squares function maps the square function to every element in a list.

Note that the definition of the squares function comes in two equations, one for an empty list and one for non-empty lists. The equations are matched against the received arguments in order, when encountering an empty list the first match on line 2 is applied. When the parameter is a non-empty list the first case does not match, but the second case on line 3 will match, so this equation is applied instead.

```haskell
squares :: [Int] -> [Int]
squares []     = []
squares (x:xs) = square x : squares xs

square :: Int -> Int
square x = x*x
```

Listing 1: A recursive function definition in Haskell

RECURSIVE DATA TYPES

Function definitions are not the only places where recursion can be used in Haskell. Custom types created with newtype and data can also be declared recursively (Hutton, 2017).

An example of a custom version of a list declared with the data keyword, taken from Hutton (2017), is shown in Listing 2. Here a new parameterised data type List is declared, which can be an empty list (Nil) or a list of the form Cons x xs where x is a value in the list and xs is another list.

```haskell
data List a = Nil | Cons a (List a)
```

Listing 2: A recursive data type in Haskell

A list of our own type List containing integers can then be defined as follows:

```haskell
l :: List Int
l = Cons 3 (Cons 2 (Cons 1 Nil))
```

Listing 3: The application of a recursive data type in Haskell

Recursive data types and recursive functions work well together. We can define our own version of the length function to calculate the length of a list recursivly (Hutton, 2017). Such a function could look like this:

```haskell
len :: List a -> Int
len Nil          = 0
len (Cons _ xs) = 1 + len xs
```

Listing 4: A recursive function for calculating the length of a list

2.2. EVALUATION IN HASKELL

A Haskell expression that can be evaluated further by performing beta reduction is called a redex, short for reducible expression (Hutton, 2017). Haskell uses lazy evaluation, this means that functions are evaluated with outermost evaluation. Outermost evaluation prescribes that the outermost redex, that is the redex that is not contained in another redex, is evaluated first. When there is more than one unencapsulated redex the leftmost is evaluated first (Hutton, 2017). Below a step-wise evaluation of a call to the squares function is shown, the redexes that are being evaluated are underlined.

```
squares [13, 6]
↓
square 13 :  squares [6]
↓
(13 * 13) :  squares [6]
↓
```

```
169 :  squares [6]
↓
169 :  square 6 :  squares []
↓
169 :  (6 * 6) :  squares []
↓
169 :  36 :  []
↓
[169, 36]
```

Since Haskell is a pure language the result of an expression does not depend on the order of evaluation, although the evaluation strategy can have implications for the termination behaviour (Hutton, 2017). Because Haskell has no side effects the evaluation order is in general unimportant for the result, thus a recursive function can be evaluated in any chosen order. From this property follows that there are different correct ways in which a student can trace a recursive function.

2.3. Mental Models and Conceptual Models

Mental Models

Cognitive psychology offers an interesting perspective for looking at computing education. This branch of psychology makes a distinction between the represented world and representations in one's mind of the represented world (Gentner & Stevens, 2014). One of the key concepts in cognitive psychology are mental models.

The concept of the mental model springs from the idea that the experiences of things around us, the representations in our mind, are different from the real life things causing them (Forrester, 1971). Obviously, when one thinks about a real-life thing such as a car, the car is not physically present in one's mind, we build a mental image or model of the car in our mind.

According to cognitive psychology a mental model describes the mental structure, the structure in one's mind, representing an aspect of one's environment. It is theorized that people can have mental models of all kind of things, although the primary subject for mental model theorists has been causal systems and the interaction between humans and such systems, such as digital interfaces and computers (Gentner & Stevens, 2014). Or, what is more in line with the subject of this thesis, a computer programming technique.

When learning a programming technique, students build their own mental models of this technique. A mental model provides an explanation and enables the student to make predictions (Wu et al., 1998). Someone who possesses a mental model of a process is able to mentally visualize the process (Götschi et al., 2003). A mental model can be "run" as a mental simulation of the program (Gentner & Stevens, 2014). In short, a mental model of a programming technique is how one views and understands this technique.

Note that a mental model can be incorrect. A faulty mental model makes wrong predictions or fails to correctly explain the process.

Conceptual Models

To facilitate the forming of a correct mental model teachers use an explanation of a system or process: a conceptual model (Turner & Belanger, 1996). In the views of Turner and Belanger (1996) a conceptual model does not need to be strictly accurate, consistent or complete as long as it aids in the explaining and understanding of the system. In this sense an analogy or metaphor can be a viable conceptual model. Götschi et al. (2003) have a stricter view: they define a conceptual model as an accurate, consistent and complete model used by experts. We will use this strict view in the following discussion unless noted otherwise.

The mental model constructed by the student can be consistent or be at odds with the conceptual model. A mental model that is consistent with the conceptual model is called a viable model, a mental model at odds is called non-viable.

Mental and Conceptual Models of Recursion

Different conceptual models can exist for a programming construct. Wu et al. (1998) name five conceptual models that can be used for introducing recursion to novice programmers: Russian dolls, process tracing, stack simulation, mathematical induction and structure templates. Some of these models are not strict conceptual models in the sense used by Götschi et al. (2003).

The Russian dolls metaphor, where the image of a doll containing other dolls is used to visualize recursion, is not a strict conceptual model but an analogy. It can, however, be used to ease the introduction of recursion. Mathematical induction is also not a strict conceptual model. In this model prior mathematical knowledge is leveraged to introduce recursion by using the proof by induction as an analogy. It too can be used to ease the introduction but is not an accurate nor complete description of the programming technique. Structure templates, where similar recursive problems are solved using prior recursive functions as templates, is not a strict conceptual model either. This model can only be used to solve recursive problems of a certain form and is therefore not complete. This leaves process tracing and stack simulation as strict conceptual models.

Kahney (1983) identified different mental models used by students for recursion. The identified mental models have been expanded by Götschi et al. (2003). Viable and non-viable models were discovered by looking at submissions of student tests by both Kahney and Götschi et al. (2003). Non-viable models include the looping, active flow, step, return value, syntactic or magic, algebraic, and odd models. These non-viable models are discussed in more detail in Section 4. The copies model is seen as both a correct mental model and a usable conceptual model. In the copies model "recursive procedures are seen to generate new instantiations of themselves, passing control, and possibly data forward to successive instantiations and back from terminated ones". It is considered the expert view. The copies model is a form of process tracing, as mentioned by Wu et al. (1998). Figure 2 shows a visualisation of the merge-sort algorithm using the copies model (Wilcocks & Sanders, 1994).

One of the important properties of the copies model is what Götschi et al. (2003) call the winding and unwinding phase. In the winding phase new instantiations of the functions are called, "passing control and possibly data forward". The unwinding phase typically con-

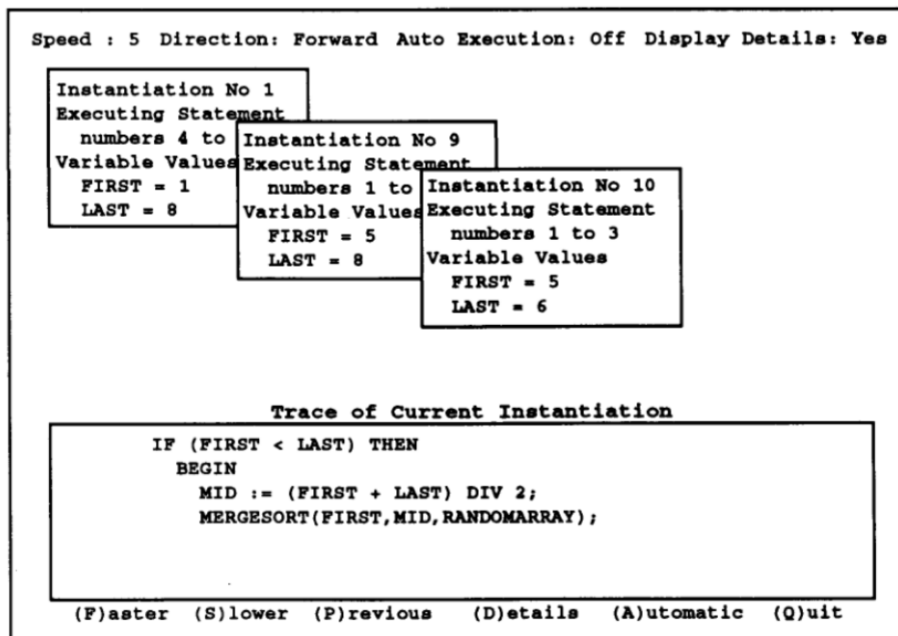


Figure 2: An example of a visualisation of a recursive call using the copies model (Wilcocks & Sanders, 1994)

sists of the return statements where the value is substituted and control is passed back to the previous calling function.

CONSTRUCTIVISM AND PHENOMENOGRAPHY

Mental and conceptual models are two concepts of cognitive psychology. This branch of psychology has much more interesting concepts to offer regarding what it is to learn such as the theories of working memory, cognitive load and schemata, to name a few. These interesting topics will remain untouched in this thesis.

Other points of view of which we will highlight some ideas, that will be used later in this document, are constructivism and phenomenography.

Constructivism emphasises active learning and its social context. Several ideas have emerged from constructivism, such as the insight that by learning a student constructs its own knowledge based upon the student's prior knowledge. Another insight is that effective learning is an active construction of knowledge (Phillips, 1995). Instead of passive consumers of knowledge students should be active participators in order to maximize the learning gains.

Phenomenography investigates the different ways in which students understand or experience phenomena, in our case the programming technique recursion. During a phenomenographic study different qualities of understanding are identified through interviews. Typically, the different understandings or insights of the student group under investigation is hierarchically structured with increasing levels of inclusion. The study of Booth (1992), of which a small part is summarized in this document, is a phenomenographic study.

As is the case with cognitive psychology, the constructivist theory and the phenomenographic perspective have much more to offer. For a good introduction related to computing

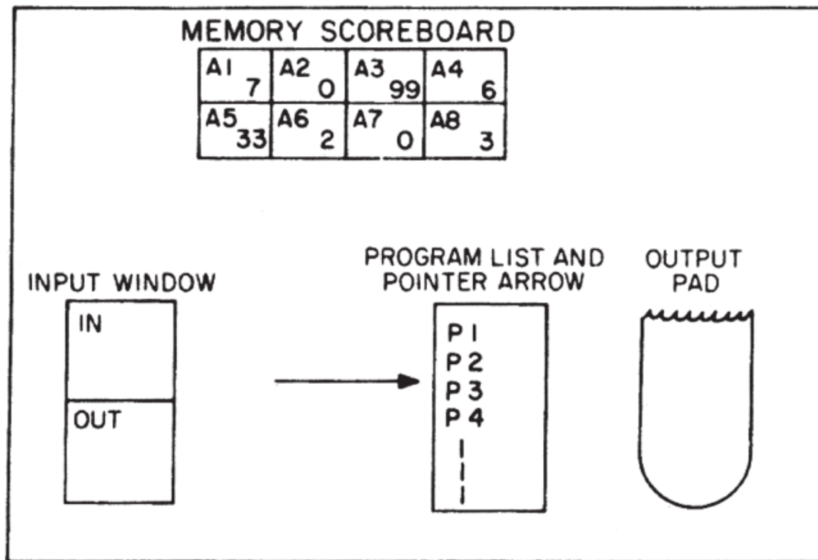


Figure 3: Visualization of a notional machine (Mayer, 1981)

education see Sorva (2012).

2.4. The Notional Machine

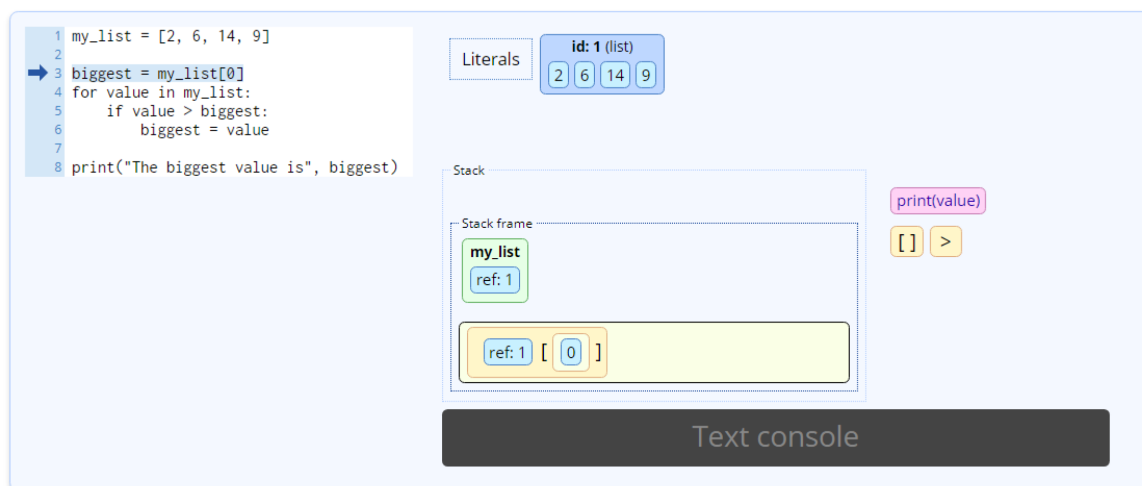
Closely related to mental and conceptual models of programming techniques is the concept of the notional machine. In 1981 Du Boulay coined the term notional machine for "the idealized, conceptual computer whose properties are implied by the constructs in the programming language employed" (Du Boulay et al., 1999). In an educational context educators present a visualization of a notional machine at the appropriate level of abstraction to encourage the forming of viable mental models.

Program Visualizations

For novices the runtime behaviour of code is not immediately obvious from looking at code. It is common to visualize a notional machine as a simplified computer in textbooks, slides or drawn on a black board (Naps et al., 2003) to make the behaviour of code clearer. Figure 3 shows a visualisation used by (Mayer, 1981). He found that novices to whom he introduced this model, together with an accompanying explanation and early in the learning process, were better able to solve code problems requiring creative solutions.

Through the years a lot of different visualizations of notional machines were proposed. Next to static, on paper visualizations, as used by Mayer (1981), a lot of interactive visualization software programs have been developed. Most of these programs focus on the imperative or the object-oriented paradigm (Sorva, Karavirta, & Malmi, 2013).

A typical visualization of an imperative notional machine consists of a memory stack with changeable content, similar to the sketch used by Mayer. A recent interactive visualization generated by the Jsvee library is shown in Figure 4 (Sirkia, 2016). Jsvee is looked at in more detail when comparing different visualization tools.



Fetching value 0 - ready.

Figure 4: Visualization of imperative code in Jsvee (Sirkia, 2016)

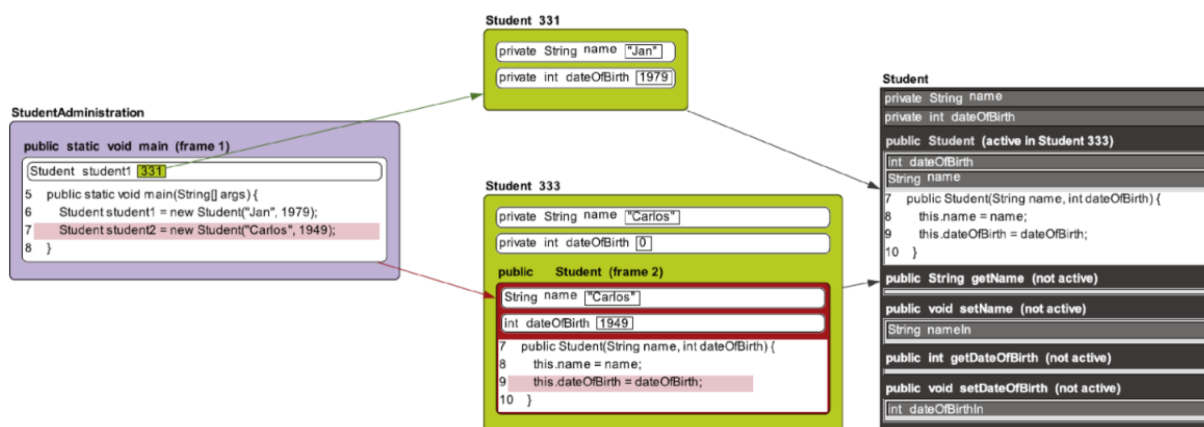


Figure 5: Visualization of object-oriented code in Evizor (Moons & De Backer, 2013)

An example of an early tool aimed at visualizing recursion is Recursion Animator by Wilcocks and Sanders, which uses overlayed windows to visualize the copies model, see Figure 2. Here the focus is the displaying of recursive function calls including the content of local variables.

A visualization of an object-oriented notional machine often displays classes, instantiated objects and their relations, additionally to the memory stack. An example is Evizor by Moons and De Backer (2013) where students can visualize the class and object structure they programmed, see Figure 5. Two other notable examples of the many object-oriented visualizers are Jeliot (Moreno, Myller, Sutinen, & Ben-Ari, 2004) and BlueJ (Kölling, Quig, Patterson, & Rosenberg, 2003).

Since functional programming is less involved with changeable memory content, but stays closer to mathematical functions, functional notional machines focus more on func-

tions and their inputs and outputs rather than on a changeable memory stack. KIEL is such a visualization tool for the language Standard ML (Berghammer & Milanese, 2001). KIEL displays an expression tree that can interactively be rewritten by substitution and simplification. Figure 6 shows a screenshot of KIEL illustrating the construction of a binary search tree. Another example using a similar approach is WinHIPE which we will look at in more detail in the Section 5 where we compare different tools.

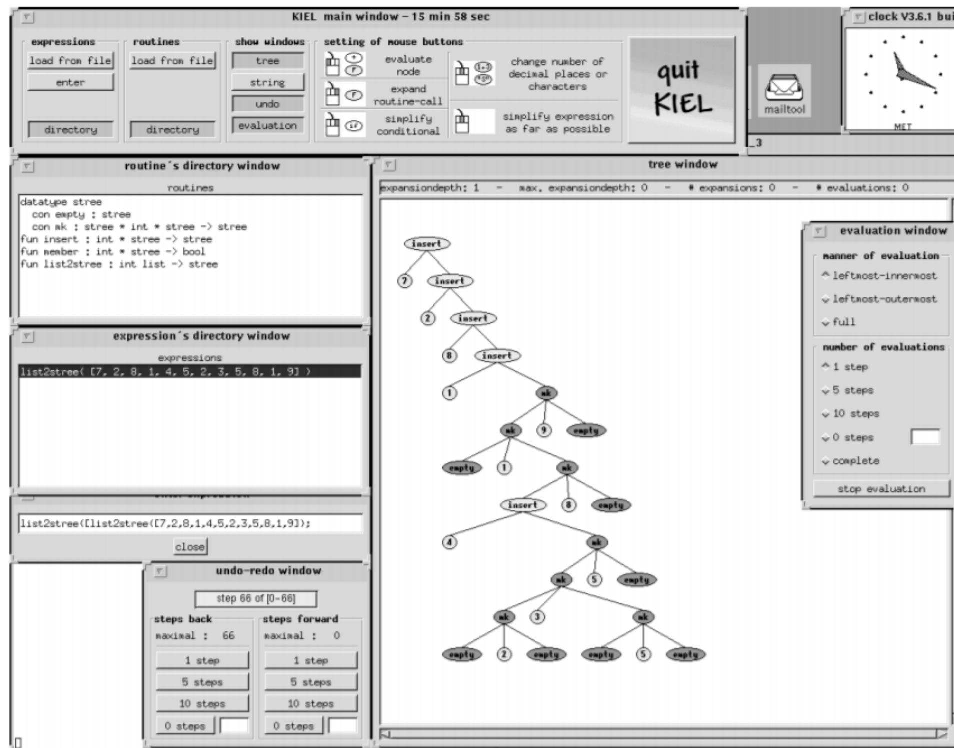


Figure 6: Visualization of functional code in KIEL (Moons & De Backer, 2013)

A somewhat different approach is used by RainbowScheme (Sho-Huan, Ching-Tao, Wing-Kwong, & Jihn-Chang, 2001). This tool visualises recursion in the Scheme language. The intermediate expression-environments are calculated via four semantic rules, a subset of Scheme called visualcode. Dynamically generated environments are color coded illustrating the folding and unfolding of recursive evaluation. Figure 7 illustrates the evaluation of a factorial function call.

2.5. INTELLIGENT TUTOR SYSTEMS

Closely related to visualization tools are intelligent tutor systems (ITS). The distinctive feature between a visualization tool and an ITS is that the latter has the ability to offer exercises and assess the students' input as being correct or incorrect.

OUTER AND INNER LOOP

In The Behaviour of Tutor Systems VanLehn (2006) describes the general structure of ITS as having an inner loop and an outer loop. The outer loop selects a task, in most cases an exercise, for the student to solve. The inner loop contains the different steps of the exercise.

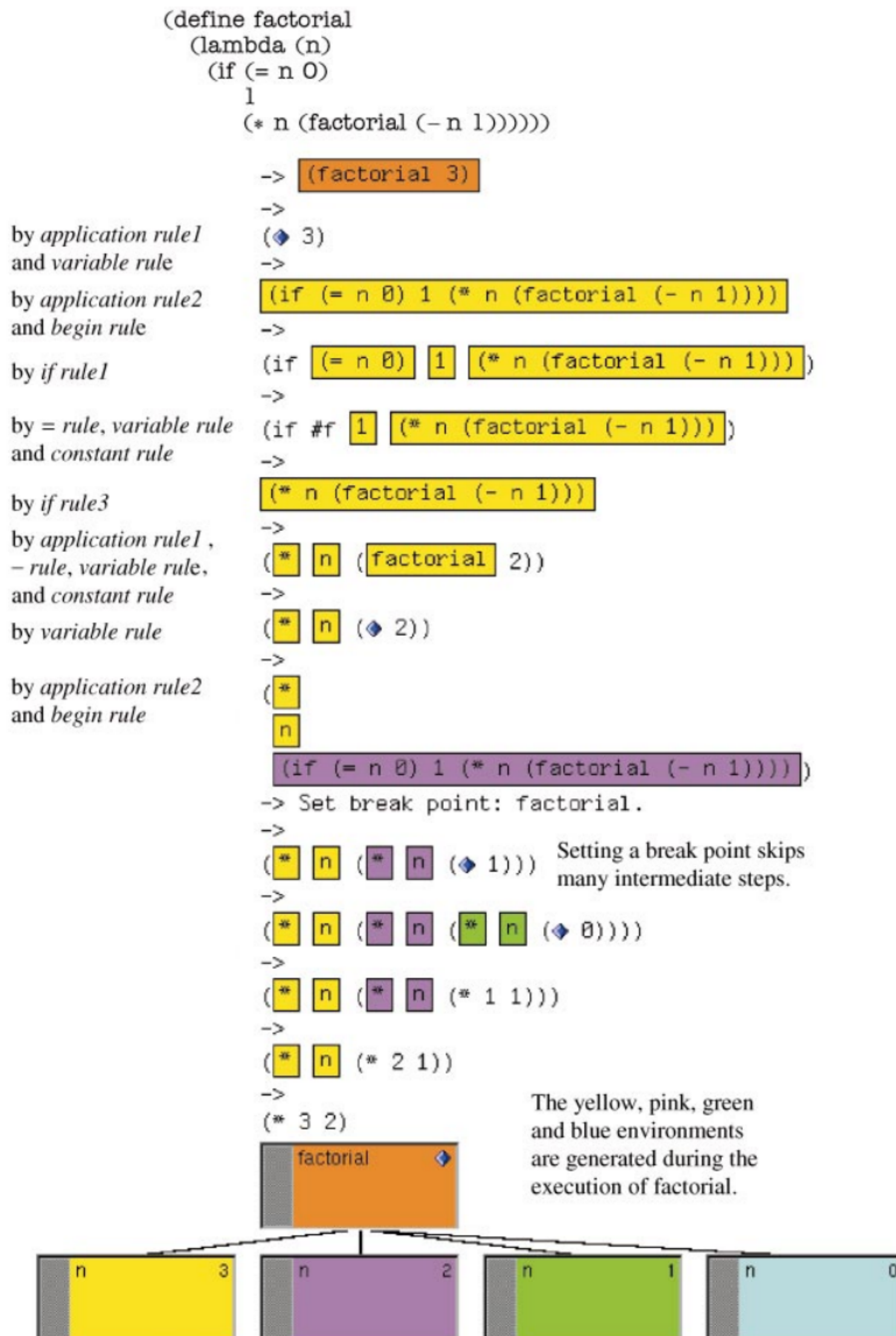


Figure 7: Visualization of a call to the factorial function in RainbowScheme (Sho-Huan et al., 2001)

The outer loop could be simply controlled by the student: the student selects the problem he wants to solve via a menu or a next exercise button. Other more sophisticated outer loops exist. Some ITS store student information in a student model. With a sophisticated student model an ITS can keep track of the student's strengths and weaknesses instructing the outer loop to select tasks specifically to strengthen the student's weaknesses (VanLehn, 2006).

Components of an Intelligent Tutor System

In general, intelligent tutor systems consist of 4 main components (Nwana, 1990):

- Tutoring module
- Expert knowledge module
- Student model module
- User interface module

An overview of the basic structure of an ITS is taken from Heeren and Jeuring (2014) and displayed in Figure 8.

The user interface provides an environment for the student to complete a task. The tutoring module selects the tasks for the student based on the student's input, the student model and the expert knowledge module. The expert knowledge model contains the exercises and checks the student's attempts for correctness (Heeren & Jeuring, 2014).

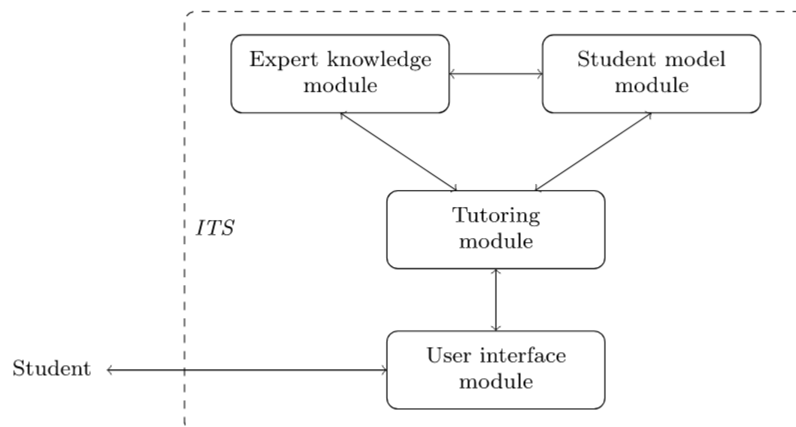


Figure 8: Components of an Intelligent Tutor System (Heeren & Jeuring, 2014)

ITSs come in all kinds of forms and sizes. The components listed above are not always present or some components might be clustered into one component. Other components could be present such as an authoring environment where teachers can add, update, or delete exercises or a monitoring environment where the progress of students can be tracked (Heeren & Jeuring, 2014).

SCAFFOLDING

Scaffolding is the support a tutor, be it a human or computer, provides for a student while trying to execute a task. Analogous to the construction of a building where scaffolding is necessary the student might not be able to accomplish the task at hand without the offered support (van de Pol, Volman, & Beishuizen, 2010).

Fading the scaffolding is the pedagogical technique where the support is gradually diminished. When the student becomes more proficient she learns to solve the exercises independently. In "the behavior of tutoring systems" VanLehn (2006) describes a tutoring system called Steve. Steve provides different modes, a demonstration mode, a mode where every step is hinted and a mode without unsolicited hints. This can be seen as an application of fading the scaffolding.

Fading the scaffolding is one form of scaffolding adjustment. The adjustment can also be made in the opposite direction, by offering more support instead of less, or in other words: adding scaffolding (Belland, 2017). The ability to reveal hints is a form of adding scaffolding on request. Ideally several hints are available per step where the hints should gradually offer more support.

2.6. THE HASKELL EXPRESSION EVALUATOR AND THE IDEAS FRAMEWORK

The Haskell Expression Evaluator (HEE) is a visualisation tool and tutor that can calculate, verify and demonstrate the step-wise evaluation of Haskell expressions (Olmer et al., 2014). In the demonstration mode the user can submit an expression and see the calculated evaluation steps. One of the goals of HEE is to show the differences between innermost and outermost evaluation. The user is able to display these two evaluation methods.

Besides a demonstration mode HEE features a practice mode where the user is asked to provide the next evaluation step. HEE evaluates the user's input and provides feedback. The tool can also hint on the next step to apply on request.

Similarly to the proposed prototype in this thesis HEE uses a recursive data type internally to encode Haskell expressions. HEE is intended to be used in an introductory course and supports Haskell features such as pattern matching, recursive functions, higher order and different evaluation strategies.

HEE uses the IDEAS framework, a generic framework for defining an expert knowledge module, also called a domain reasoner, for an ITS (Heeren & Jeuring, 2014). IDEAS is an acronym for "Interactive domain-specific exercise assistant". With the IDEAS framework it is possible to define rewrite strategies as a context-free grammar. The domain reasoner can provide automated feedback on solutions proposed by students.

HEE is included in the comparison of the tools in Section 5.

2.7. LITERATURE REVIEWS

Several literature reviews concerning program visualisation and ITS have been published. In this part we give an overview of some of the literature reviews that inspired this thesis

and provides interesting pointers for discussing such digital educational tools.

An overview describing and discussing a large volume of code visualisations tools and their evaluative studies was done by Sorva et al. (2013). The study found that the visualisation tools were often short-lived research prototypes. The investigated evaluative studies suggest that the tools positively contribute to the learning process. In the current study we use the engagement framework as proposed by Sorva et al. while answering research question 2.

Keuning, Jeuring, and Heeren (2018) performed a systematic literature overview on a large volume of exercise tools that are able to generate automatic feedback on student solutions. Pre-defined selection criteria were used to select 101 tools identified by examining 17 prior literature overviews and additionally querying 2 databases. It was found that most of the tools are not easily adaptable by teachers. In general the tools are focused on identifying mistakes and not on giving feedback on the next step to take. However, automatic feedback generation is an active field of research, new techniques are being used in the recent tools that lead to more supportive feedback. The researchers pointed out that the studies on the effectiveness of the tools were often lacking in quality.

Nesbit, Adesope, Liu, and Ma (2014) reviewed the literature on the effectiveness of ITS. Twenty-two studies were selected that met pre-defined selection criteria. Across these studies it was found there was a significant advantage by using an ITS instead of applying teacher-led classroom instruction or non-ITS computer-based instruction.

Crow, Luxton-Reilly, and Wuensche (2018) reviewed fourteen tutor systems and documented their supplementary features. The documented features were:

- Questions the user needed to answer such as multiple choice question in a quiz-like form
- Plans or visualisations the tool asked the user to draw
- Plans or visualisations the tool presented to the user
- Lesson content or reference material provided by the tutor
- Worked-out solutions provided by the tutor

The review concludes that supplementary features could be valuable, but are often absent in ITSs. To embed the tutor in a knowledge domain together with reference material would allow the student to see the bigger picture and to connect the dots. Together with a student model this would allow the student to see where her weaknesses are and to decide what to study next. Also a case is made for including user-generated plans that should reduce structural problems in the student's code.

3. Research Design

3.1. Research questions

The main research question is:

How can a tutoring environment support students in understanding the evaluation of recursive functions in Haskell?

This question is subdivided as follows:

- RQ1. What are difficulties and misconceptions encountered by students when trying to understand recursive functions?
- RQ2. How do current educational tools illustrate recursive functions?
- RQ3. How can a tutoring environment be designed to illustrate recursion in Haskell?

3.2. Research method

RQ1. What are difficulties and misconceptions encountered by students when trying to understand recursive functions?

A literature study in the next section distills difficulties and misconceptions encountered by novices when introduced to recursion. Four studies are examined. We will also look at pedagogical good practices proposed in the examined studies.

RQ2. How do current educational tools illustrate recursion?

Some of the available tutoring environments and program visualizations are investigated. Since the number of tools aimed at the functional paradigm is limited the investigation has a broader scope and includes other programming paradigms as well.

The results of RQ2 are used as inspiration for the design of the prototype when answering RQ3.

RQ3. How can a tutoring environment be designed to illustrate recursion in Haskell?

A part of the research is the design of a tutor prototype aimed at learning the application of recursive functions in Haskell. The aim of the prototype is to illustrate recursion in Haskell while trying to alleviate the identified difficulties and misconceptions in RQ1.

From the researched tools those that follow a similar approach are selected and compared to the prototype. The comparison will focus on the illustration of recursion and the mitigating of the identified difficulties and misconceptions.

4. DIFFICULTIES AND MISCONCEPTIONS ABOUT RECURSION

In this section we first give an overview of the mental models of recursion identified by Kahney (1983) and expanded by Götschi et al. (2003). Afterwards we will summarize the difficulties and misconceptions found by Hamouda, Edwards, Elmongui, Ernst, and Shaffer (2018) and summarize some of the findings of the phenomenographic study by Booth (1992). We conclude with pedagogical suggestions derived from both the mentioned and other studies.

4.1. MENTAL MODELS OF RECURSION BY KAHNEY

Kahney (1983) investigated the understanding of a group of novices and a group of experts about recursive evaluation. A question was devised to identify the differences of understanding of the passive flow of a recursive call in the SOLO-language, a beginner-friendly, LOGO-like database manipulation language. Figure 9 shows the abridged question asked by Kahney. In the complete original question there was a third solution that did not achieve the required result.

Recently I needed a programme which would make the following inference: if somebody 'X' has 'flu' then whoever 'X' kisses also has 'flu,' and whoever is infected spreads the infection to the person he or she kisses, and so on. Starting with the database given in Figure A, I needed a programme which would change the Figure A database into the Figure B database.

```
JOHN---KISSES--->MARY---KISSES--->TIM---KISSES--->JOAN

                        FIGURE A.

JOHN---KISSES--->MARY---KISSES--->TIM---KISSES--->JOAN
 |                |                |                |
HAS              HAS              HAS              HAS
 |                |                |                |
 |------------>--|---->FLU<----|--<----------------|

                        FIGURE B.
```

I have been provided with two solutions to the problem, both called 'TO INFECT /X/' and these are labelled SOLUTION-1, SOLUTION-2, below. I want you to consider each programme in turn and say (A) whether or not the programme will do what I want it to do, and (B) if it will, say how it does it (in your own words), or, if it won't, say why it doesn't (again in your own words).

```
SOLUTION-1:                          SOLUTION-2:

TO INFECT /X/                        TO INFECT /X/
1 NOTE /X/ HAS FLU                   1  CHECK /X/ KISSES ?
2 CHECK /X/ KISSES ?                 1A If Present: INFECT * ; CONTINUE
2A If Present: INFECT * ; EXIT       1B If Absent:  CONTINUE
2B If Absent:  EXIT                  2  NOTE /X/ HAS FLU
DONE                                 DONE
```

Please write your answers on the pages provided overleaf.
Thank you for cooperating.

Figure 9: The abridged question asked by Kahney (1983).

THE EXPECTED MODELS: COPIES AND LOOPING MODEL

Kahney hypothesized that experts possess the copies model as a mental model. This is the viable model that takes into account both the active and the passive flow. A new function call spawns a new function instance in the active or winding phase until the base case is

reached. The last called function returns its value to the previous function in the passive or unwinding phase until all called functions have returned.

In the non-viable looping model the recursive process is not seen as spawning new functions instantiations, but rather as a form of iteration in only one function instance. The base case is seen as the stopping condition of the loop. This model can lead to correct results, but fails in some cases. Kahney expected that novices possess this mental model.

When the respondent selected both solutions as achieving the required result and also commented on the order of the side-effect changing the database, that is the line "NOTE /X/ HAS FLU", the response was categorized as showing strong evidence for possession of the copies model.

On the other hand, when only solution 1 was selected and solution 2 was rejected on the basis that only JOAN would be affected, this was categorized as showing strong evidence for the possession of the looping model.

Outcome and conclusion

Kahney found that 8 out of the 9 questions experts showed strong evidence for possessing the copies model, whereas only 1 out 30 novices showed strong evidence for the copies model. 16 out of 30 novices selected solution 1 and rejected solution 2, but only 4 mentioned JOAN as being the only one affected. Kahney concluded there was only weak evidence that the other novices possessed the looping model. It could be those novices possessed other mental models.

From the respondents' comments Kahney identified other possible models. Some respondents argued it is invalid that the function definition includes a call to itself, rejecting the possibility of recursion. These models were categorized as null models. When the respondents thought that the flow of control statement 'EXIT' acts as the stopping rule for recursion the mental model was categorized as the odd model. The last identified model was the syntactic magic model where students wrongly based the outcome of the function call on the position of the different program segments.

4.2. Mental models of recursion by Götschi et al.

In the research by Götschi et al. (2003) students' submissions to two code tracing questions were analysed. One question was specifically aimed at discovering if students understood the passive flow of a recursive call: we will focus on this question. Similar code to the code accompanying Götschi's question is shown in listing 5. The question asked was to trace the function call f(8).

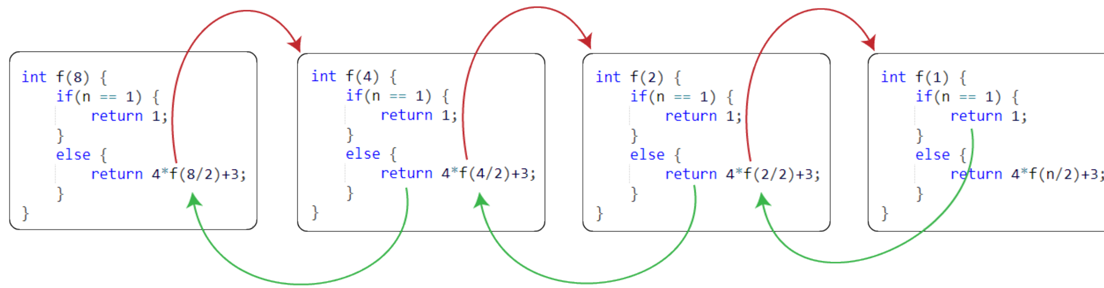


Figure 10: The copies model. Red arrows indicates the active flow, green the passive flow

```
int f (int n) {
    if(n == 1) {
        return 1;
    }
    else {
        return 4 * f(n/2) + 3;
    }
}
```

Listing 5: The recursive function used by Götschi et al. to identify mental models (translated to Java)

A correct calculation of the call f(8) is 4 * (4 * (4 * 1 + 3) + 3) + 3 = 127. An activation diagram illustrating the two phases of the copies model is shown in Figure 10.

Götschi et al. also identified the copies and looping model from the students' solutions. Because the looping model can lead to correct results, but fails in some cases, Götschi et al. dub this a "risky" viable model.

Götschi did find other mental models that share some similarities with those of Kahney. Listed below is a summary of the mental models that were identified.

I. Active model

The active model only takes into account the active phase and disregards the passive phase. This model can also lead to correct results when the passive phase does not pass data back to the calling functions. In the function provided by Götschi et al. the order and precedence of operations requires the results to be passed back to the calling functions to get a correct result. Hence, in this case the active model fails.

II. Magic model

Götschi et al. categorized the submissions that showed some insight into the recursive nature of the assignment, but disregarded important details as the magic model. Götschi et al. gave some examples of faulty solutions with the magic model, see Figure 11.

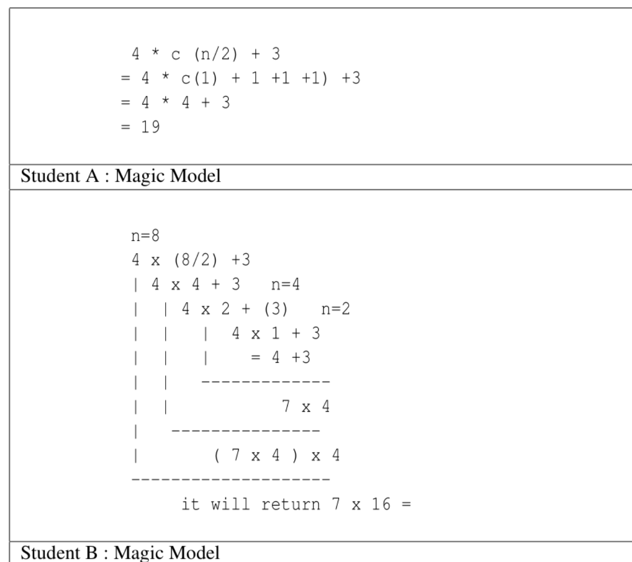


Figure 11: Two solutions categorized as magic models (Götschi et al., 2003).

```
4(1)(8/2) + 3
=4(4) + 3
= 19
```

Figure 12: Student solution categorized as step model. (Götschi et al., 2003).

III. Return value model

This non-viable model is the result of students' misconceptions about parameter passing, function calling, return values and program stack.

IV. Step model

In this non-viable model the function is only called once and thus not honoring the repeating embedded function calls. Only one branch of the if else statement was executed. An example of a student solution categorized as step model is shown in Figure 12.

V. Algebraic model

In this rare model students tried algebraic techniques to derive a formula from the function. An example of a student solution categorized as algebraic model is shown in Figure 13.

VI. Odd models

Traces that were incomprehensible or showed several aspects of the above models indicating lack of insight were categorized as odd models.

4.3. Limitations of the classification of mental models

The classification of mental models of individuals does have its limitations. A major problem is that one cannot simply peek inside the head of a person and draw detailed conclu-

```
if n= 1  then 1
    4*c(n/2) + 3 = 1
    4*c(1/2) + 3 = 1
    c/2 = -1/2
    c = -1
    Then if n = 8
    4(-1)(8/2) + 3 = -13
```

Figure 13: Student solution categorized as algebraic model. (Götschi et al., 2003).

sions about her internal representations. Mental models are not always identifiable from the limited notes from the students, the models are not directly accessible and some theorists even argue to discard them as irrelevant (Uttal, 2000).

Another issue is that Götschi's classification of the models is fine-grained, the active model and magic model seem quite similar and from the examples given it is not always clear what the differences are. Also, the concept of a mental model breaks down when students do not have a clue and start to guess answers.

We do, however, not take such an extreme stance as Uttal and do believe Kahney's and Götschi's work provides some interesting insights. The looping and active model reveal misconceptions about the base case and the passive flow. These models are popular faulty models and are almost correct, the students almost got the crux of the matter. If the students were exposed to examples and exercises where these misconceptions were challenged, the misconceptions could be mitigated.

Both Kahney and Götschi point out that programmers holding non-viable mental models can come to correct working solutions in some situations, but the model will fail in other situations. According to Kahney the holding of a non-viable mental model can allow the person to debug the model when confronted with a counter-example.

4.4. Difficulties and misconceptions found by Hamouda et al.

Hamouda et al. (2018) also studied students' responses to exam questions. RecurTutor was specifically designed to remedy the identified difficulties and misconceptions. RecurTutor is looked upon in more detail in Section 3. Listed below are the misconceptions Hamouda et al. found that are relevant to the functional paradigm.

- No statements that appear after the recursive call will execute.
- Statements that appear after the recursive call will execute before the recursive call is executed.
- If there is a base case, then it will always execute. If the recursive call does not reduce the problem to the base case, then the base case will return and that will terminate the recursive method.

Other identified misconceptions relate to mutable variables in imperative languages. Some difficulties identified by Hamouda et al. are:

- Cannot formulate a recursive call that eventually reaches the base case.
- Cannot write a correct base case.
- Cannot properly evaluate the base case such that the student believes that the recursive method executes one more or one less time than it should.

The found difficulties all revolve around the base case. The two first misconceptions seem to stem from a weak understanding of function invocation and the function return phase.

4.5. Phenomenographic perspective by Booth

Booth looked at novices learning recursion in the functional language SML from a phenomenographic perspective. She interviewed the students, analyzed and categorized the students' responses and examined exercise solutions. From the interview she identified three increasing levels of understanding pertaining to recursion that the starting students expressed.

- Recursion as a program construct
- Recursion as a means to accomplish iteration
- Recursion as being self referential

The levels of understanding are listed in an increasing hierarchy of insight. When a student understands recursion as a means to accomplish iteration he also understands recursion as a program construct. When a student understands recursion as being self referential he also understands the other two lower levels.

Surprisingly Booth did not find a strong connection between the deeper understanding of the subject and the solving of an exercise. Some students who expressed deep understanding of recursion during the interview had difficulties to provide correct answers to recursion exercises. She also found that students who provided correct solution did not always express deep understanding during the interview. In one case a student called Philip provided a correct solution through using a teacher's example as a starting template that was correctly filled in. Philip expressed the weakest level of understanding, namely recursion as a program construct only.

Booth concludes that solutions provided by students do not necessarily reflect the students' understanding of the subject matter. It does not always make sense to try to deduce the students' understanding of a program technique by looking at students' solutions.

4.6. Pedagogical suggestions

Based on the identified models Götschi makes some suggestions about best practices in the classroom. According to Götschi it is important for the students to have a good grasp

of function invocation, parameter passing and return values before introducing recursion. Also embedded function calls should be well understood.

Götschi argues that detecting a student's non-viable mental model can help to target the specific misconceptions and could lead to greater learning gains. It is therefore important that the examples and exercises are well constructed to challenge the weaknesses of risky mental models.

Similar insights are found by Velázquez-Iturbide (2000). He proposes a gradual introduction when novices are introduced to recursion. Since novices typically do not have a good grasp of function invocation, parameter passing etc. Velázquez-Iturbide proposes to teach recursive grammars first, recursion in functional languages next and later introduce recursion in imperative languages. Velázquez-Iturbide argues recursion in imperative languages is more challenging because of the interaction with the imperative mechanics.

Hamouda et al. note that it takes different skills to trace than to write recursive functions. In the case of recursive functions it is the tracing that forces the student to focus on the details of how the computer executes the recursive function calls. While writing a recursive function seasoned programmers do not focus on how the computer executes the function. Hamouda illustrates this with the factorial function where one just writes fact(n) = n * fact(n - 1) and consider this as a mathematical fact without worrying about how fact(n - 1) is calculated. According to Hamouda this is similar to a call to a library function, one simply trusts the call to fact(n - 1) will succeed.

This is similar to how experienced programmers trace programs. Detienne and Soloway (1990) found experienced programmers usually use symbolic tracing instead of concrete tracing. In symbolic tracing details are ignored and no concrete values are filled in as long as the program behaves as expected. As an example, the behaviour of a looping or recursive construct is assessed without a full concrete tracing of the code. When encountering unexpected behaviour the skill of concrete tracing is used to fill in concrete values in variables and go over every statement step by step.

Novices do not have the skill to perform symbolic nor concrete tracing yet and are required to look into the details of code execution and learn concrete tracing first. Once they mastered the details they can look at it in a different light and use symbolic tracing. As Charlie Parker once said: "Master the instrument. Master the music. And then forget about all that and just play."

4.7. SUMMARY

To conclude this section we will summarize the findings in a few take-home messages.

1. The copies model is the conceptual model held by experts about recursion. In this model every recursive call spawns a new function with its own scope, when the base case is reached data can be returned to the previous functions in the so-called passive flow.
2. Several non-viable mental models seem to exist when looking at students' solutions. It is common that novices are unaware of the passive flow of recursive evaluation.

Mental models, such as the looping and active model, are risky and can lead to correct results when the passive flow does not alter the result. Exercises should be well constructed to challenge the weaknesses of these risky mental models.

3. Identifying a correct base case or recursive step is often challenging for novices.

4. Recursion is challenging due to its reliance on other basic programming concepts such as function invocation and parameter passing. These should be well understood first.

5. Recursion is more challenging in imperative languages. The introduction of recursion via grammars and functional programming could ease the understanding.

6. The ability to provide a correct solution does not always reflect a deep understanding of recursion.

7. Once the evaluation of recursive functions is mastered through tracing, the writing of recursive functions can be looked upon form a higher level of abstraction and becomes less challenging.

Points four to six are rather general pedagogical remarks and are out of scope of the current research. The first three points are relevant for the tutor prototype: visualisations of the copies model, exercises that actively involve the passive flow and exercises asking for identification of a correct base case or recursive case should be offered.

5. HOW DO CURRENT EDUCATIONAL TOOLS ILLUSTRATE RECURSION?

In this section we compare tools that try to help novices to understand the execution of programming code. We examine eight tools: three visualizations tools and five tutors. Some tools are focused on recursive code while others have a broader focus.

The visualization tools and tutors have similar educational aims and have overlapping features. Visualizers are focused on generating animations, but this distinction is blurry. Some tutors also provide animated code execution and some visualizers do provide the ability to input code. The distinctive feature to be called a tutor is for a system to be able to offer exercises and to be able to judge students' input to be correct or incorrect.

Name	Content ownership	Representation	Paradigm	Language
WinHIPE	Own content	Animated expression steps and trees	Functional	Hope
SREC	Given content	Code highlighting, activation tree, control stack, function trace	OO	Java
JSVEE	Given content	Code highlighting, animated stack frame, text console	OO	Language agnostic

Table 1: The examined visualization tools (Pareja-Flores et al., 2007; Fernández-Muñoz, Pérez-Carrasco, Velázquez-Iturbide, & Urquiza-Fuentes, 2007; Sirkia, 2016)

Name	Content ownership	Representation	Task	Paradigm
Ask-Elle	Modified content	Partial function definition with holes	Provide step-wise function definition	Functional
RecurTutor	Given content	To be completed code examples and questions	Provide base case, recursive step, tracing questions	OO
ChiQat	Given content	Animated evaluation, code highlighting and animated recursion graph	Draw recursion graph, answer multiple choice questions	OO
WHAT	Modified content	To be completed code questions	Evaluate expression, provide function type or definition	Functional
HEE	Own content	Rewrite steps	Provide next rewrite step	Functional

Table 2: The examined intelligent tutoring systems (Olmer et al., 2014; Gerdes, Heeren, Jeuring, & van Binsbergen, 2016; Hamouda et al., 2018; AlZoubi et al., 2014)

5.1. VISUALIZATIONS TOOLS

WINHIPE

WinHIPE is an IDE with an educational focus (Pareja-Flores et al., 2007). It is one of the few visualization tools in the functional paradigm. The user, be it a student or educator, is able to author code and let the tool generate animations with little effort. A custom interpreter was built to enable the generating of evaluation steps. The tool uses term rewriting to generate different expression trees showing the rewrite steps. The tool supports graphical representations for lists and trees, other expressions are displayed textually. The rewrite steps can be compiled into animations. The user can choose to omit steps in the animation before generating the animation. Figure 14 shows the rebuilding of a tree from its pre- and inorder traversals.

The animations can be saved as files, viewed in the WinHIPE environment or can be exported as animations for viewing on the web. For educational purposes descriptions and source code can be added to explain the animations, see Figure 26 near the end of this thesis for an example.

The viewer of the animation is able to play and pause the animation as well as perform

other actions such as step forward or backward, jump to start and jump to end.

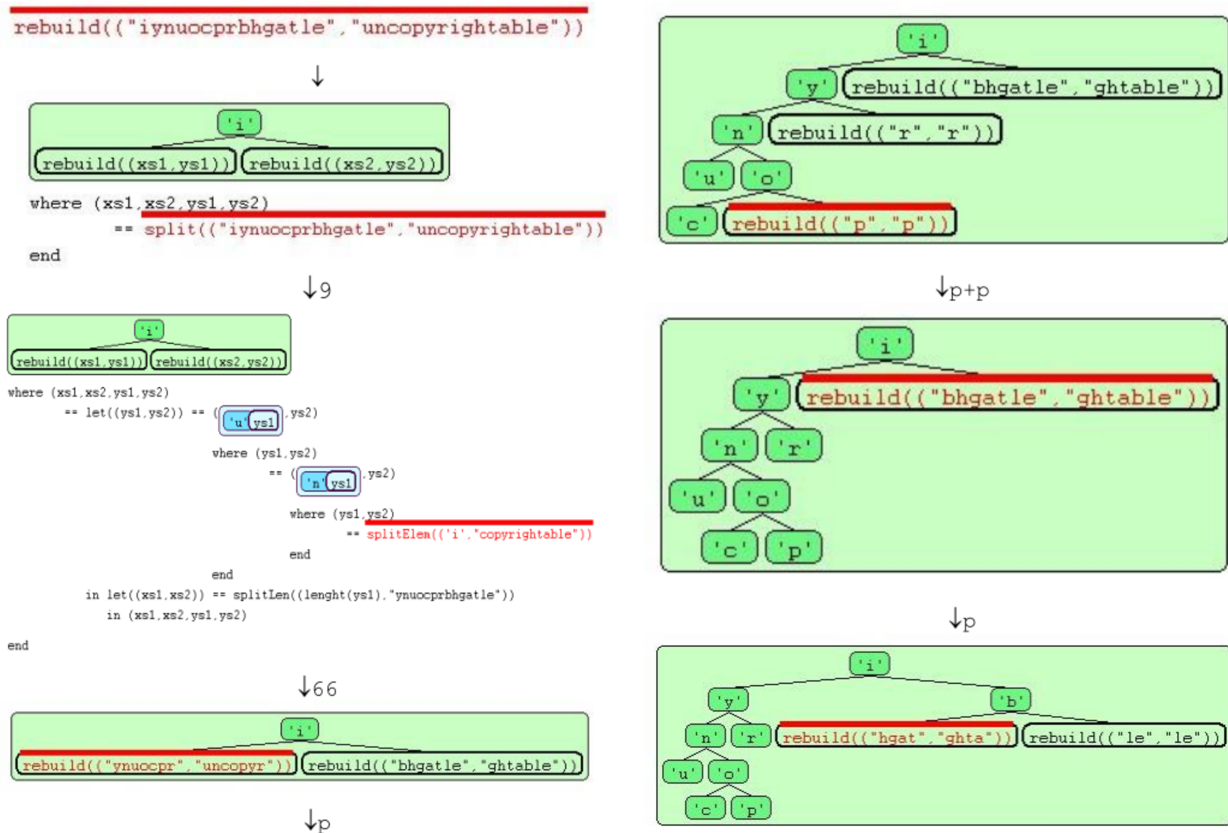


Figure 14: Visualisations generated by WinHIPE.

SREC

SREC is a visualization tool aimed at illustrating recursion in Java (Fernández-Muñoz et al., 2007). It features different views of the executed code: an activation tree, a control stack and a trace of the function calls, see Figure 15. To create an animation the user has to preprocess the code manually. The input and return variables are fed into the SREC framework to enable it to detect the program execution and output the animation. The downside of this approach is that changing the code, which is a trivial action for an educator, might be a task too complex for novices who might have a hard job understanding the code as it is.

The animations can be saved as files for future retrieval. During execution of the animation the preprocess edits of the code are hidden for demonstration purposes. As with WinHIPE the tool provides animation controls.

JSVEE

Another tool in the crowded world of the object-oriented code visualizers is JSVEE (Sirkia, 2016). This open-source browser-based library enables educators to generate animations with little effort. JSVEE is aimed to be language agnostic and currently supports a subset of Python and C# with the help of external translation files. To generate an animation the

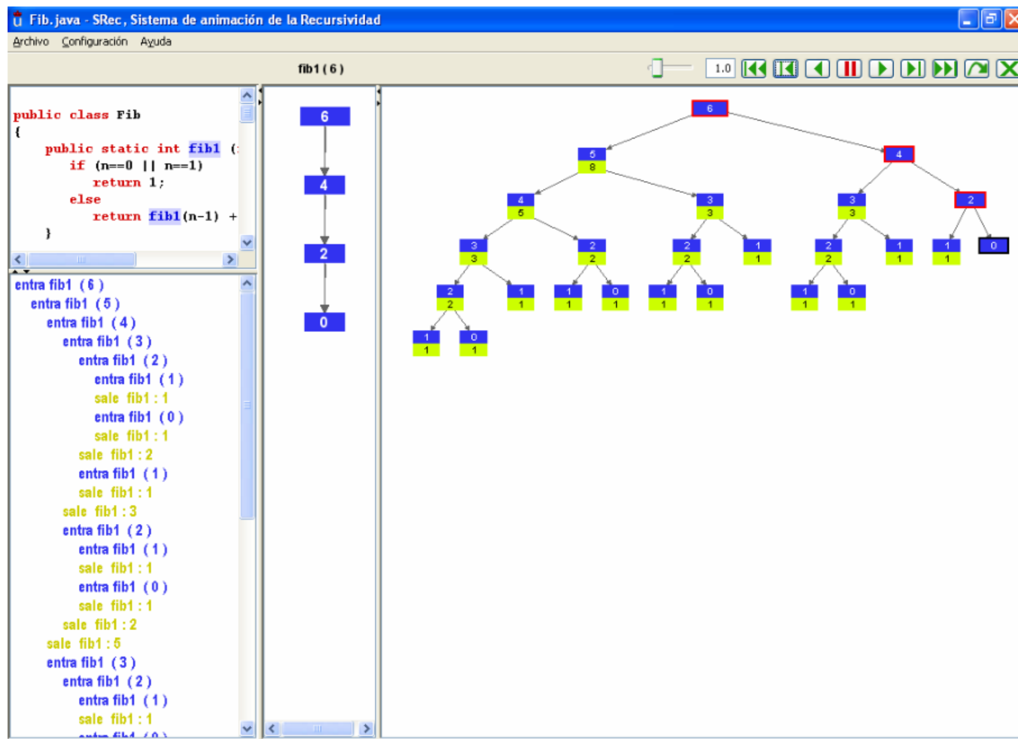


Figure 15: An animation of the Fibonacci algorithm in SREC

source code of a program is automatically converted into a Javascript object containing all the animation information.

There is no environment to save or open files. An animation can however easily be saved as a small static stand-alone website that thus can be used for demonstration purposes. Explanatory annotations can be added to steps with the optional Kelmu library.

The animations are displayed in-browser and show the execution of expressions in their own stack frame. The controls are limited to the essential controls: play, one step forward, one step backward and jump back to start.

5.2. Tutoring Systems

Ask-Elle

Ask-Elle is an intelligent tutoring system that supports the step-wise development of a given set of Haskell exercises (Gerdes et al., 2016). It is not aimed at introducing recursion but is included in this overview because it is one of the few tutoring systems for the Haskell language.

The tool can evaluate incomplete program solutions through the use of "holes". The user can fill in a question mark to indicate a hole where the program needs to be extended. Ask-Elle provides hints on request for the next steps to take. To accomplish this Ask-Elle uses strategy-based model tracing and property-based testing to assess a student's solution. Like HEE, Ask-Elle also uses the IDEAS framework.

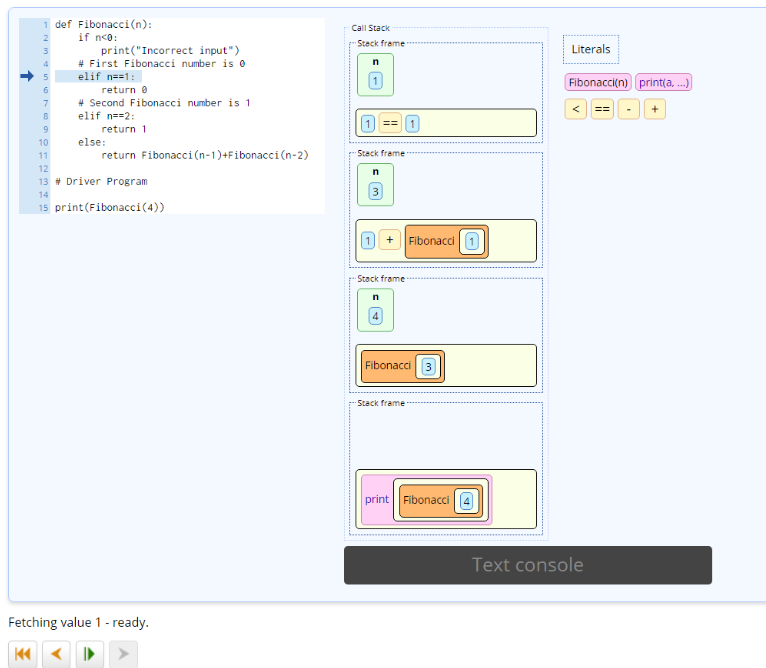


Figure 16: An animation in JSVEE showing a recursive process stack trace

Practice with the evaluation of a Haskell Expression

Haskell Expression

Start length [1,2,3,4,5] Select

Options

Outermost evaluation strategy

Innermost evaluation strategy

Next step

Diagnose Next evaluation step

Hints

Show number of steps left | Show all rules that can be applied | Show next rule

Show next step | Do next step

Output

Rules that can be applied independent of strategy:
Calculate the length of a list

Derivation

length [1,2,3,4,5]
= { Calculate the length of a list }
foldl (\n -> \x -> n + 1) 0 [1,2,3,4,5]

Figure 17: The Haskell Expression Evaluator (Olmer et al., 2014)

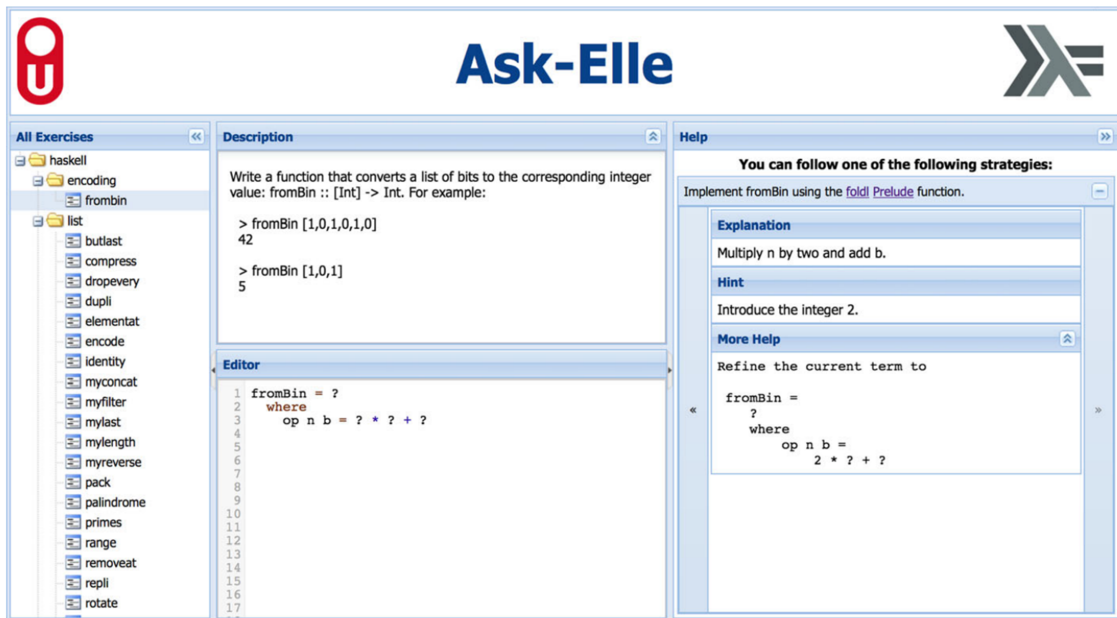


Figure 18: An exercise in the Ask-Elle tutor, the question marks are the holes that are to be filled in (Gerdes et al., 2016).

RECURTUTOR

RecurTutor is a tutor providing a fixed set of exercises in the Java language (Hamouda et al., 2018). As the name implies it is specifically aimed at teaching recursion. The tutor presents code completion questions and tracing questions. The code completion questions focus on filling in the base case or the recursive case. The student's submission is sent to the server where the code is validated through several test cases that should succeed in order to complete the exercise successfully. The results of the function calls are offered as feedback in the RecurTutor interface.

RecurTutor is the result of a research into the difficulties and misconceptions students have when encountering recursive questions during exams at Virginia Tech. RecurTutor was created to remedy the found misconceptions and difficulties. Hamouda et al. found that students using RecurTutor outperformed students that received typical instructions.

The exercises do not provide a visualization, but the accompanying introduction does. The introduction consists of a body of text and step-through slides. Here the different instantiations of a recursive function are shown with their code and variables. A screenshot of the explanation is shown in 19, a screenshot of an exercise in 20.

Hamouda, Edwards, Elmongui, Ernst, and Shaffer (2020) contributed another tutor focusing on recursion algorithms in binary trees. This tool is not covered in this thesis since it is not aimed at novices.

CHIQAT

The ChiQat tutor is an interactive tutor in the object-oriented paradigm (AlZoubi et al., 2014). It consists of a fixed set of exercises for both Java and Python of which a set is focused on recursion.

11 / 11

The n=5 copy will multiply the return value of the n=4 by 24. This last copy will return the result of the required factorial.

```
int factorial(int n) {
 if (n <= 1)
  return 1;
 return n * factorial(n-1);
}
```

```
int factorial(n = 2) {
 if (n <= 1)
  return 1;
 return 2 * 1;
}
```

```
int factorial(n = 3) {
 if (n <= 1)
  return 1;
 return 3 * 2;
}
```

```
int factorial( n = 4 ) {
 if (n <= 1)
  return 1;
 return 4 * 6;
}
```

```
int factorial( n = 5) {
 if (n <= 1)
  return 1;
 return 5 * 24;
}
```

Figure 19: A screenshot of the slides accompanying the RecurTutor by Hamouda et al.

X264: Recursion Programming Exercise: Multiply

For function `multiply`, write the missing base case condition and action. This function will multiply two numbers `x` and `y`. You can assume that both `x` and `y` are positive.

Examples:

```
multiply(2, 3) -> 6
```

```
1 public int multiply(int x, int y) {
2   if <<Missing base case condition>> {
3     <<Missing base case action>>
4   } else {
5     return multiply(x - 1, y) + y;
6   }
7 }
8
```

Check my answer! Reset

Figure 20: A screenshot of an exercise of the RecurTutor by Hamouda et al.

The ChiQat tutor visualises the copies model through a so-called "recursion graph". A recursion graph is a directed graph where ovals stand for function invocations and rounded rectangles stand for function returns. Before a student can attempt an assignment an animation is played showing an animated recursion graph while the corresponding line of code is highlighted (see Figure 21). After the animation several assignments are given such as identifying the base case in a multiple choice question and drawing a recursion graph. Feedback is provided after submission of a solution. The exercises and animations are not editable.

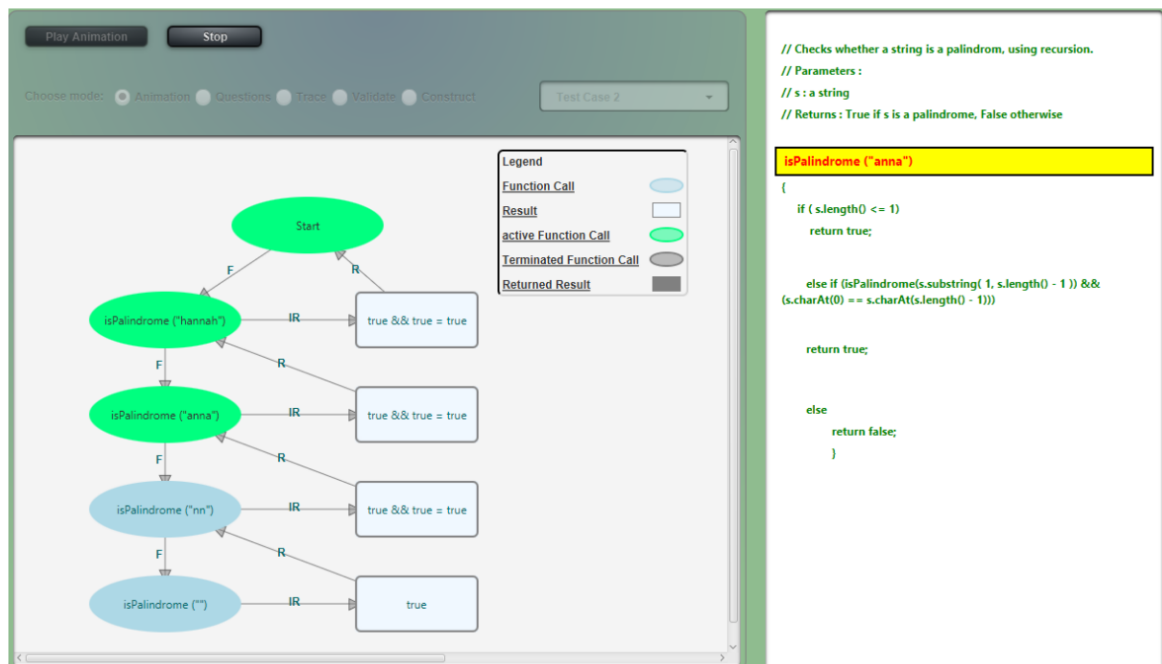


Figure 21: A screenshot of the Chiqat tutor by (AlZoubi et al., 2014)

WEB-BASED HASKELL ADAPTIVE TUTOR

The Web-based Haskell Adaptive Tutor, WHAT for short, is an ITS for learning Haskell (López, Núñez, Rodríguez, & Rubio, 2002). It offers different levels of exercises in a friendly and easy-to-use interface. Exercises range from simple numerical calculations, over typing and building simple function definitions, to more complex custom higher-order functions. Assignments on recursive functions are also included.

WHAT employs a student model and is able to adapt the level of exercises to the student's performance. To offer assignments at the appropriate level the student's learning speed and memory performance are assessed.

Each student belongs to a group of students called a class. Next to recording a student's performance on an individual level, WHAT records the performances of all the students in a class and learns what to expect from a typical student in a certain class. Additionally, a human teacher can use this system to see which topics students are struggling with.

Assignments are taken from a predefined set. Random variations are generated from this set and are offered to the students when they have reached the appropriate level. When

a student submits a solution the solution is checked with a set of input data in a Haskell environment. The set of input data is carefully selected to be able to offer sensible feedback and hints.

HASKELL EXPRESSION EVALUATOR

The Haskell Expression Evaluator (HEE) is both a tutor and a code visualizer (see Figure 17 for a screenshot). It is explained in more detail in Section 2.

5.3. LEVEL OF ENGAGEMENT

We use the 2DET engagement taxonomy proposed by Sorva et al. (2013) to assess the level of engagement the tools offer. 2DET stands for *two dimensional engagement taxonomy* and is an extension of the taxonomies proposed by Naps et al. (2003) and Myller, Bednarik, Sutinen, and Ben-Ari (2009). The 2DET taxonomy lays out the engagement that a tool provides across two dimensions: the content ownership and the direct engagement dimension. Increasing one of these properties increases the total engagement. The tools are layed out in Figure 22.

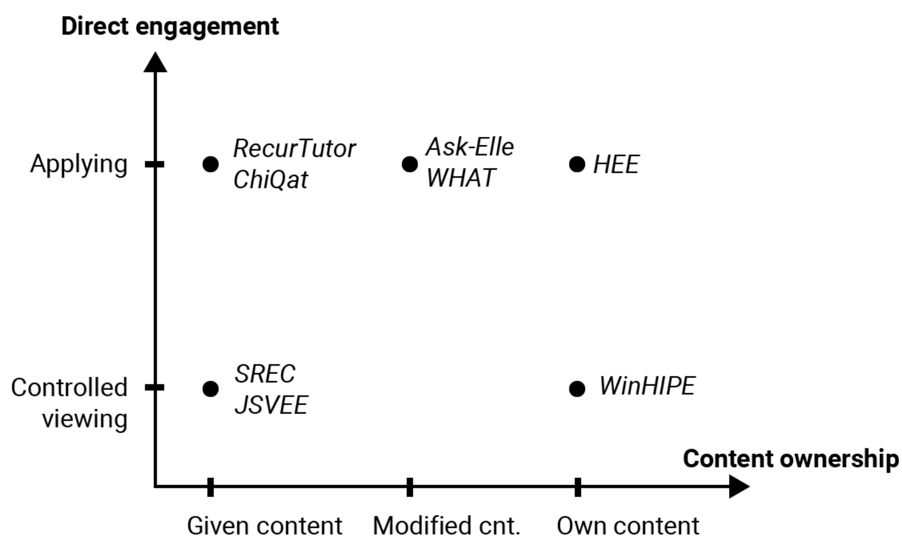


Figure 22: The tools layed out in the 2DET taxonomy.

The content ownership dimension represents how the user is able to change the content. To change the content increases the interactivity and thus makes the learning more active which, according to the constructivist viewpoint, is a good thing. When the student provides its own code to be visualised instead of code that he did not produce is arguably more engaging, for one, the student will probably be more interested in his own code.

Sorva et al. (2013) distinguish four levels of content ownership: given content, own cases, modified content and own content. A tool that offers content that can not be altered by the user falls in the given content category. When a user can enter its own content, to visualise or to practices with, the tool is categorized as such. Most of the tools in this

document fall in the given content or own content category. Ask-Elle and WHAT occupy a special spot. To fulfill an assignment the tutors ask to input code for a certain task. Different solutions, and in the case of Ask-Elle different routes on how a student develops the function, are supported. We therefore place the tutors in the modified content category.

The direct engagement dimension expresses the level of interactivity with the content. Sorva et al. (2013) identified seven levels of direct engagement. The tools fall in two categories, for brevity we will clarify just those two.

The visualizers all fall in the "controlled viewing" category which means the user can play and pause an animation, skip back etc, but no other input is provided. It is no surprise the tutors fall in a higher category of direct engagement since they expect more complex input. The tutors fall in the "applying" category: the user should modify components to perform a task.

The 2DET taxonomy places two important properties on the foreground. There are also other properties that influence the engagement of a tool such as the level of complexity of a task, the form of the animation or task or what idea a visualization or tutor communicates. As an example, the ChiQat tutor asks the user to draw a recursion diagram with ovals, rounded rectangles and arrows. This way of illustrating might appeal to certain students and could lead to greater learning gains. Ask-Elle on the other hand introduces the notion of incremental development and hole-driven development where partially finished solutions can be evaluated. This approach could appeal too, increasing engagement and learning gains.

5.4. EFFECTIVENESS

Most of the tools were tried out in classroom settings, but often the test settings were not documented and feedback was qualitative or informal. We will discuss two more rigorous experiments including control groups performed with RecurTutor and WinHIPE.

Hamouda et al. (2018) recorded the time spent with RecurTutor and found that students using the tutor spent more time practicing than students using traditional methods. The students that had used the tutor had better scores on the subsequent recursion exam questions.

It is difficult to make strong claims about the effectiveness of the tutor because the amount of time spent using the tutor versus traditional methods is not equal. RecurTutor was, however, conceived to encourage students to spend more time with the material and challenge the students' misconceptions. In that sense the tutor was effective in this particular student group.

An interesting experiment was done by Urquiza-Fuentes and Ángel Velázquez-Iturbide (2007). They asked two similar groups of students to study the tree breadth traversal algorithm. One group received animations created with WinHIPE, the other group got instructions to build animations with WinHIPE. Both groups had ample time to study the algorithm and were asked to answer questions when they thought they had enough knowledge. Interestingly the group that had to create the animations did spend more time with the material and scored better in the subsequent questions.

The study with WinHIPE indicates that exercises that require more interactivity and challenge the students at an appropriate level encourage students to spend more time which leads to greater learning gains. Both studies do suggest that the tutors are engaging and encourage the students to spend more time studying. These findings are in line with constructivists' claim that learning should be an active endeavour.

Both studies suggest that a higher level of meaningful interaction leads to more time spent studying and greater learning gains. This could be because the interaction offered by the tools challenges the misconceptions of the students, possibly making the students more aware of their weaknesses and thus motivating the students to overcome these weaknesses and increasing the time spent with the learning material. Another effect at play could be that the students were more entertained by the tools that showed self-created animations in the case of WinHIPE and direct feedback and a quiz-like setup in the case of RecurTutor.

To make claims about the effectiveness of the other tools more studies are needed.

5.5. SUMMARY

The examined tools display different approaches on how to teach and illustrate recursion. WinHIPE and SREC have a more graphical approach using tree structures. Jsvee provides elaborate animations illustrating imperative mechanics and displays an expanding stack frame when evaluating recursive calls. The examined tutors are more text based and usually require text input to complete tasks. ChiQat is the only exception in this category, requiring the student to draw a recursion graph.

RecurTutor is embedded in a large volume of reference material and is accompanied by introductory lessons with text and static visualizations. As argued by Nesbit et al. (2014), including reference material is an advantage. Most of the other tools were also used in a classroom settings, also accompanied by lessons and reference material, but these were not shared.

Both HEE and Ask-Elle provide the possibility to ask for hints. Ask-Elle has an advanced feedback system offering immediate and detailed feedback and offers links to reference material in the provided hints. The other three tutors offer less scaffolding.

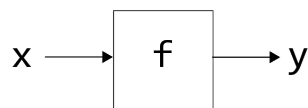
The handling of input is an important property as it raises the level of engagement of a tool. The study with WinHIPE and RecurTutor suggest that students who are more actively engaged with a tool show greater learning gains.

6. A TUTOR FOR RECURSION IN HASKELL

In this section a tutor is presented that tries to mitigate the misconceptions and difficulties identified in Section 4. The tool illustrates the copies model by means of function tables, these are first briefly touched upon. Subsequently the architecture and the inner workings of the tools are described and illustrated with some selected code snippets and a walk-through.

6.1. USING FUNCTION TABLES TO ILLUSTRATE REWRITE STEPS

The basic building block of a functional language such as Haskell is the function. The concept of a function in Haskell is analogous to a function in mathematics in that it transforms, through function application, one or several input values to an output value.



When transforming integer values the notation in Haskell stays close to the algebraic notation of a function.

$$f(x) = 2x + 2$$

```
f x = 2 * x + 2
```

Listing 6: A function that takes an integer and outputs another

When evaluating a recursive function the given expression is repeatedly rewritten. The given redex, short for reducible expression, is evaluated step-by-step until it cannot be reduced anymore. Below the evaluation of a call to `fac 4` with the different rewrite steps is shown. The last four multiplication steps are combined into one step. The redexes are underlined.

```
fac 4
↓
4 * fac 3
↓
4 * (3 * fac 2)
↓
4 * (3 * (2 * fac 1))
↓
4 * (3 * (2 * (1 * fac 0)))
↓
```

```
4 * (3 * (2 * (1 * 1)))
↓
24
```

Function tables are a widely used notation for introducing algebraic thinking in mathematics education (Martinez & Brizuela, 2006). Because function tables are so widely used it is an interesting notation to leverage to illustrate function application in Haskell.

WinHIPE and SREC show the function calls in a tree diagram to illustrate the evaluation. In contrast we will show the calls to the fac function in a function table. The rows in the table make the application of the function definition explicit. The table illustrates the copies model as the expert conceptual model. Every line in the table corresponds to one function instantiation. Table 3 illustrates such a function table for `fac 4`.

The proposed tutor automatically calculates the tables. A screenshot of the tutor evaluating `squares [3, 5, 7, 3]` is shown in Figure 23.

FAC	
Input	**Output**
4	4 * fac 3
3	3 * fac 2
2	2 * fac 1
1	1 * fac 0
0	1

Table 3: A function table with the input and output of the fac function.

6.2. Architecture of the prototype

The proposed prototype is a web-based tool that consists of two parts: a Haskell program that utilizes the Haskell cgi package and a browser front-end. The Haskell part is currently one program that contains the function definitions and the function calls to be evaluated. When receiving an http-request it calculates the rewrite steps and function tables, renders the html code and responds by sending back the html page. The http-request includes two arguments: the exercise id and the step number.

6.3. Using algebraic data types to define and evaluate expressions

In this subsection the techniques used to build the prototype are explained. First a data type is introduced to enable the encoding of expressions. Next a simple evaluator that supports nested additions is introduced. The evaluator is then expanded in two steps to make it more general and to enable the function tables to grow interactively.

To define and rewrite expressions a recursive data type is declared in Haskell.

```haskell
data Expr = Var String | Con Int | App Expr Expr
```

Haskell Recursion Tutor

Choose an exercise:

squares [3, 5, 7, 3]

Given are following function definitions:

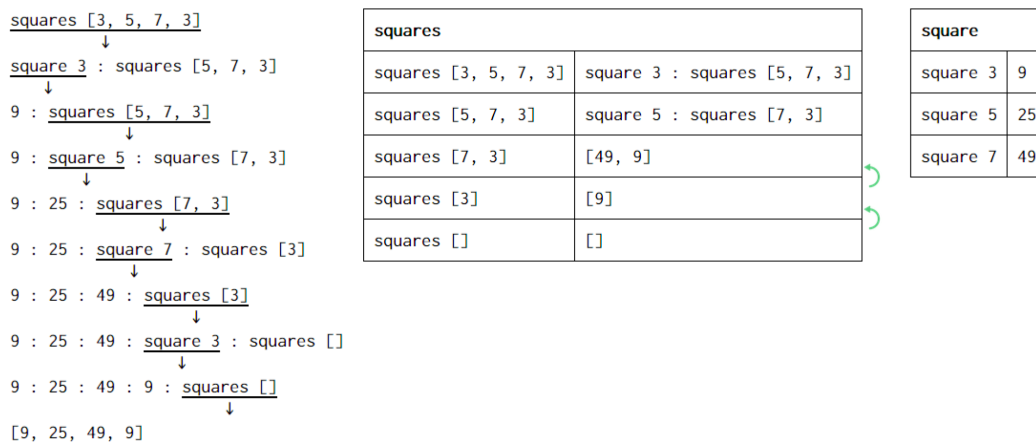
```
square x = x * x

squares [] = []
squares (x:xs) = square x : squares xs
```

View the rewrite steps below:

12/14

Evaluation done. Green arrows indicate data passed back through passive flow.



```
squares [3, 5, 7, 3]
↓
square 3 : squares [5, 7, 3]
↓
9 : squares [5, 7, 3]
↓
9 : square 5 : squares [7, 3]
↓
9 : 25 : squares [7, 3]
↓
9 : 25 : square 7 : squares [3]
↓
9 : 25 : 49 : squares [3]
↓
9 : 25 : 49 : square 3 : squares []
↓
9 : 25 : 49 : 9 : squares []
↓
[9, 25, 49, 9]
```

squares	
squares [3, 5, 7, 3]	square 3 : squares [5, 7, 3]
squares [5, 7, 3]	square 5 : squares [7, 3]
squares [7, 3]	[49, 9]
squares [3]	[9]
squares []	[]

square	
square 3	9
square 5	25
square 7	49

Figure 23: A screenshot of the proposed tutor. The expressions on the left are already fully rewritten. At the end of the rewrite process the table rows are normalized from bottom to top. Data flowing back through the passive flow is indicated with green arrows.

The idea behind this data type is that an expression could be a variable, an integer or an application of a function on an expression (be it a string, an integer value or a function application). Note that an expression encoded with this data type forms a binary tree where an App is a node. The leaves of the tree are populated by Var and Con instantiations. Some examples of expressions encoded with the data type are shown in listing 7.

```
Var "Hello" -- literal string
Con 23      -- literal integer
App (Var "square") (Con 4) -- application of function
   square on 4
```

Listing 7: Expressions encoded with the data type

The function in listing 7 is a unary function. Binary functions are also possible to define through the function application of a function application. Thus a unary function taking one parameter is applied to a second parameter through currying.

```
App (App (Var "+") (Con 4)) (Con 3)
```

Lists can be encoded too by introducing a list constructor that we will call "cons", in line with the functional tradition, and an empty list constructor. The list [5], that is the list containing the integer value 5, is displayed in Listing 8.

```
App (App (Var "cons") (Con 5)) (Var "[]")
```

Listing 8: A list encoded with the data type

To calculate the rewrite steps rewrite rules are defined. The rewrite steps of the binary function addition are demonstrated in a small evaluator in listing 9. If the evaluator encounters an application of the addition function that does not match the addition of two integers it will look at the first parameter. If this is already a fully evaluated integer it will recursively evaluate the second parameter otherwise the first. Through the use of recursion nested addition operations are supported.

```
step :: Expr -> Expr
step (App (App (Var "+") (Con n)) (Con m)) = Con (n+m)
step (App (App (Var "+") x) y) =
   case x of
      Con n -> add (Con n) (step y)
      _     -> add (step x) y
```

Listing 9: Rewrite rules for addition

The maybe type is used to indicate if a rewrite step was successful or not. An integer or string cannot be rewritten and thus returns Nothing. This technique is illustrated in the refined evaluator in listing 10 where the recursive step is written in general terms. Other

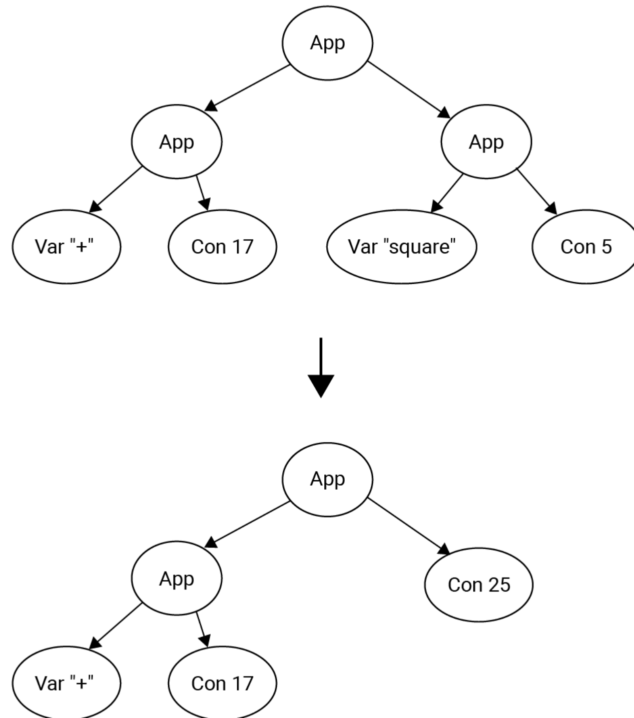


Figure 24: Rewriting of the expression 17 + square 5 to 17 + 25

functions like square and multiply can now easily be added without other changes to the code.

```
step :: Expr -> Maybe Expr
step (Con _) = Nothing
step (Var _) = Nothing
step (App (App (Var "+") (Con n)) (Con m)) = Just (Con (n+m))
step (App (App (Var "*") (Con n)) (Con m)) = Just (Con (n*m))
step (App (Var "square") (Con n)) = Just (Con (n*n))
step (App f a) =
   case step f of
      Just f' -> Just (App f' a)
      Nothing ->
         case step a of
            Just a' -> Just (App f a')
            Nothing -> Nothing
```

Listing 10: Generalized evaluator supporting addition multiply and square function

Thanks to the recursive nature of the last rewrite rule the evaluator works on nested functions. Figure 24 shows how the expression `17 + square 5` is rewritten to `17 + 25`.

The right branch in the expression tree in Figure 24 is rewritten. The function's input and result at the location of the rewrite should get added to the corresponding function

table. In this case a row in the function table "square" should be added. To identify the rewritten function the location of the rewrite is stored in a path. A path is a list of directions where each direction corresponds to the left or right branch of a node, see Listing 11. In Figure 24 the path of the rewrite is simply [R].

```
type Path = [Direction]

data Direction = L | R
```

Listing 11: A path is a list of directions and direction is either L or R

It can be tedious to have to define and read functions using the App constructor, especially for binary constructors. To make this more manageable smart constructors are defined, see listing 12.

```
bin :: String -> Expr -> Expr -> Expr
bin s x y = App (App (Var s) x) y

cons :: Expr -> Expr -> Expr
cons = bin "cons"

nil :: Expr
nil = Var "[]"

mult :: Expr -> Expr -> Expr
mult = bin "*"

add :: Expr -> Expr -> Expr
add = bin "+"

square :: Expr -> Expr
square = App (Var "square")

squares :: Expr -> Expr
squares = App (Var "squares")
```

Listing 12: Smart constructors

Listing 13 shows a further refined evaluator incorporating paths and the definition of the squares function as shown in Listing 1, enabling the mapping on a list of the square function. The recursive step builds the path when a branch can be rewritten to keep track of the location of the rewrite. The path is used to tag the expression that is about to be rewritten in the front-end.

```
1  step :: Expr -> Maybe (Expr, Path)
2  step (Con _) = Nothing
3  step (Var _) = Nothing
4  step (App (App (Var "+") (Con n)) (Con m))
5        = Just (Con (n+m), [])
6  step (App (App (Var "*") (Con n)) (Con m))
7        = Just (Con (n*m), [])
8  step (App (Var "square") (Con n)) = Just (Con (n*n), [])
9  step (App (Var "squares") (App (App (Var "cons") x) xs)) =
10    Just (cons (square x) (squares xs), [])
11 step (App f a) =
12    case step f of
13       Just (f', p) -> Just (App f' a, L:p)
14       Nothing ->
15          case step a of
16             Just (a', p) -> Just (App f a', R:p)
17             Nothing -> Nothing
```

Listing 13: Evaluator with path

The step function provides the main rewrite mechanics. It is now fairly easy to define a recursive function that produces all the rewrite steps and the location of the rewrite of an expression.

```
stepsPath :: Expr -> [(Expr, Path)]
stepsPath e =
   case step e of
      Just (e', p) -> (e, p) : stepsPath e'
      Nothing      -> (e, []) : []
```

Listing 14: Function that produces all rewrite steps and the locations of the rewrite

6.4. WALK-THROUGH

To demonstrate the evaluator we will walk through the first two rewrite steps of an example expression: the application of the squares function to the list [7, 5]. The expression `squares [7, 5]` is encoded below. The expression tree is the first tree in Figure 25.

```
App (Var "squares")
    (App (App (Var "cons")
              (Con 7))
         (App (App (Var "cons")
                   (Con 5))
              (Var "[]")))
```

The first call to the step function will match the squares rule, the expression is rewritten

to the expression below. This corresponds to the rewriting of expression tree 1 to 2 in Figure 25.

```
App (App (Var "cons")
         (App (Var "square")
              (Con 7)))
    (App (Var "squares")
         (App (App (Var "cons")
                   (Con 5))
              (Var "[]")))
```

The second rewrite, from tree 2 to tree 3, does not match any of the first rules so the last catch all recursive step on line 11 is applied. The recursive step checks if f, that is the left branch of the tree, can be rewritten by calling the step function again, starting the process again on line 12. Again none of the first rules apply so the recursive case is applied again. The left branch of the left subtree is now checked, since this is a leaf containing Var "cons" Nothing is returned indicating a rewrite is not possible and the step function is applied on the right branch instead. Finally one of the first step rules match: the square rule on line 8. Here the unfolding or unwinding of the nested recursive calls starts and the location is set to the empty list. Since step did return a Just value (the application of the square function) the subtree is rebuilt with the newly returned value and direction R is prepended on the empty list. Step f from the previous recursive call prepends L. The path to the rewrite location is thus [L, R].

The rewrite steps and their paths as produced by the stepsPath function are displayed below.

```
(squares [7, 5],[])
(square 7 : squares [5],[L,R])
(49 : squares [5],[R])
(49 : square 5 : squares [],[R,L,R])
(49 : 25 : squares [],[R,R])
([49, 25],[])
```

6.5. Getting the sub-expression form a path

Another key element of the tutor is to list the rewritten parts of the expression in the corresponding table. We will highlight some essential functions.

To get to the sub-expression the process to build the path is reversed. Instead of prepending the path, the first element is popped of at the head of the path list and the function is reapplied on the left or right branch until the empty list is reached. This idea is captured in the code in listing 15.

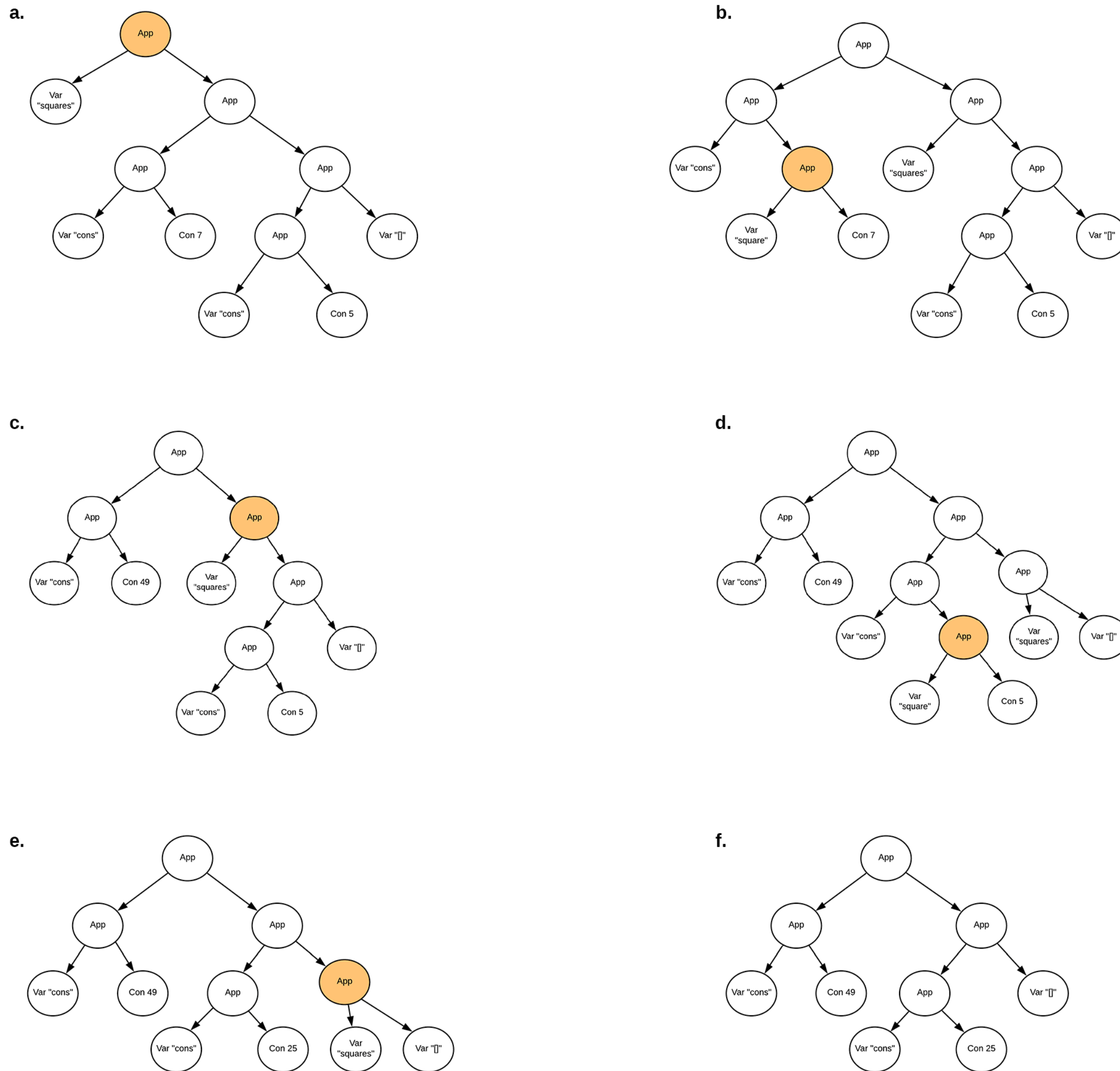


Figure 25: Rewrite trees of squares [7, 5] with highlighted rewrite locations.
The expressions and paths to the rewrite locations are listed below:

```
a. expression: square [7, 5], path: []
b. expression: square 7 : squares [5], path: [L,R]
c. expression: 49 : squares [5], path: [R]
d. expression: 49 : square 5 : squares [], path: [R,L,R]
e. expression: 49 : 25 : squares [], path: [R,R]
f. expression: 49 : 25 : []
```

```
subexpr :: Path -> Expr -> Expr
subexpr [] e = e
subexpr (L:p) (App f a) = subexpr p f
subexpr (R:p) (App f a) = subexpr p a
subexpr _ _ = error "subexpr:␣invalid␣path"
```

Listing 15: Getting the sub-expression from a path and an expression

The subexpr function is used by inOutStep to produce a list of expressions containing the name of the rewritten function and the parameters. The function is listed in Listing 16, the helper function fname is omitted for brevity. The inOutSteps function iterates the inOutStep function a given number of times or until the rewriting is finished.

```
inOutStep :: Expr -> Maybe (String, [Expr])
inOutStep e =
   case (e, step e) of
      (e, Just (e', p)) -> Just (fname $ subexpr p e, [subexpr
         p e, subexpr p e'])
      _                 -> Nothing
```


```
inOutStep :: Expr -> Maybe (String, [Expr])
inOutStep e =
   case (e, step e) of
      (e, Just (e', p)) -> Just (fname $ subexpr p e, [subexpr
         p e, subexpr p e'])
      _                 -> Nothing
```


```
inOutSteps :: Expr -> Int -> [(String, [Expr])]
inOutSteps e n
   | n <= 0    = []
   | otherwise = case (inOutStep e) of
      Just x -> x : inOutSteps (step' e) (n-1)
      Nothing -> []
```

Listing 16: Generating the function name and parameters from a rewrite.

The last function we will highlight is the generation of the tables. This function uses the inOutStep function. The input and output parameters are inserted into the tables by the insert function, double values are not inserted. The function receives an integer to stop after a certain amount of steps, this integer is received from the front-end.

```haskell
type Table = (String, [[Expr]])

exprToTables :: Expr -> Int -> [Table]
exprToTables e n = exprToTables' (inOutSteps e n) []
  where
    exprToTables' :: [(String, [Expr])] -> [Table] -> [Table]
    exprToTables' (r:rows) ts
      | rows == [] = insert r ts
      | otherwise  = insert r (exprToTables' rows ts)

    insert :: (String, [Expr]) -> [Table] -> [Table]
    insert (s,exprs) [] = [(s, [exprs])]
    insert (s,exprs) (t:ts)
      | s == fst t = if exprs `elem` snd t then (t:ts)
                     else (s, exprs:snd t):ts
      | otherwise  = t : (insert (s,exprs) ts)
```

Listing 17: Generating the function name and parameters from a rewrite

The result of an evaluation of `exprToTables` with `squares [7, 5]` as the first parameter is shown in Listing 18.

```
[("squares",
     [[squares [7, 5],square 7 : squares [5]],
      [squares [5],square 5 : squares []],
      [squares [],[]]]),
 ("square",
     [[square 7,49],
      [square 5,25]])]
```

Listing 18: The generated tables with function call and output of the functions.

The data generated by `exprToTables` constitutes the content of the function tables. When the function is fully evaluated the rows containing unevaluated expressions are normalized step by step from bottom to top to illustrate the passive flow. Green arrows are shown indicating data flowing back to the previous function call.

6.6. SUMMARY

In this section a prototype is presented that has a unique way of illustrating recursive calls by using function tables to illustrate the evaluation of recursive functions. Research question 1 showed that it is common that novices are unaware of the passive flow of recursive evaluations. The function tables make this passive flow explicit and illustrate the copies model.

To encode expressions a recursive data type is introduced. An expression can be viewed as a tree structure. A custom-made evaluator calculates the evaluation steps and saves

the rewrite location in a path, enabling the insertion of the rewritten sub-expressions in the function tables. After evaluating the function call the table will contain unevaluated expressions, the rows are normalized from bottom to top to illustrate the passive flow.

At its current state the prototype is not able to handle input and thus is not able to offer exercises nor assess input as being correct or incorrect. The handling of input enabling students to solve an exercise is a key feature of a tutor (VanLehn, 2006). At the moment the prototype is more an evaluation visualization tool than a tutor. The next section explores some possible additional features to make the prototype into a real tutor.

7. Discussion

In the future the prototype could be verified in a classroom setting to see if it holds up to its promises: does it ease the learning of recursion? As an alternative to such a verification we reflect on the prototype by comparing it to two other tools. In the first part of this section the prototype is compared to WinHIPE and HEE when evaluationing a similar function. A discussion about the limitations and possible future work concludes this section.

7.1. Comparison with WinHIPE and HEE

WinHIPE and the Haskell Expression Evaluator are tools that offer similar features as the proposed prototype. How does the prototype compare to these tools? In this part of the thesis we will look at how the three tools illustrate the evaluation of a call to the function sum or sumlist, a function that calculates the sum of a list of integers. Note that the sum function is defined in the Haskell prelude with the higher-order function fold, but such functions are often introduced in an explicit recursive manner when introducing recursion to novices (Urquiza-Fuentes & Ángel Velázquez-Iturbide, 2007) (Hutton, 2017).

Figure 26 shows WinHIPE illustrating the sumlist function in the functional programming language HOPE. WinHIPE displays descriptions, the source code of how this function can be implemented recursively and an animation of the evaluation. The Haskell Expression Evaluator shows the evaluation through the application of the definition in the Haskell prelude and is able to show the difference between inner- and outermost evaluation. Figure 27 shows an outermost evaluation of the sum function by HEE. Figure 28 shows how the prototype illustrates an evaluation.

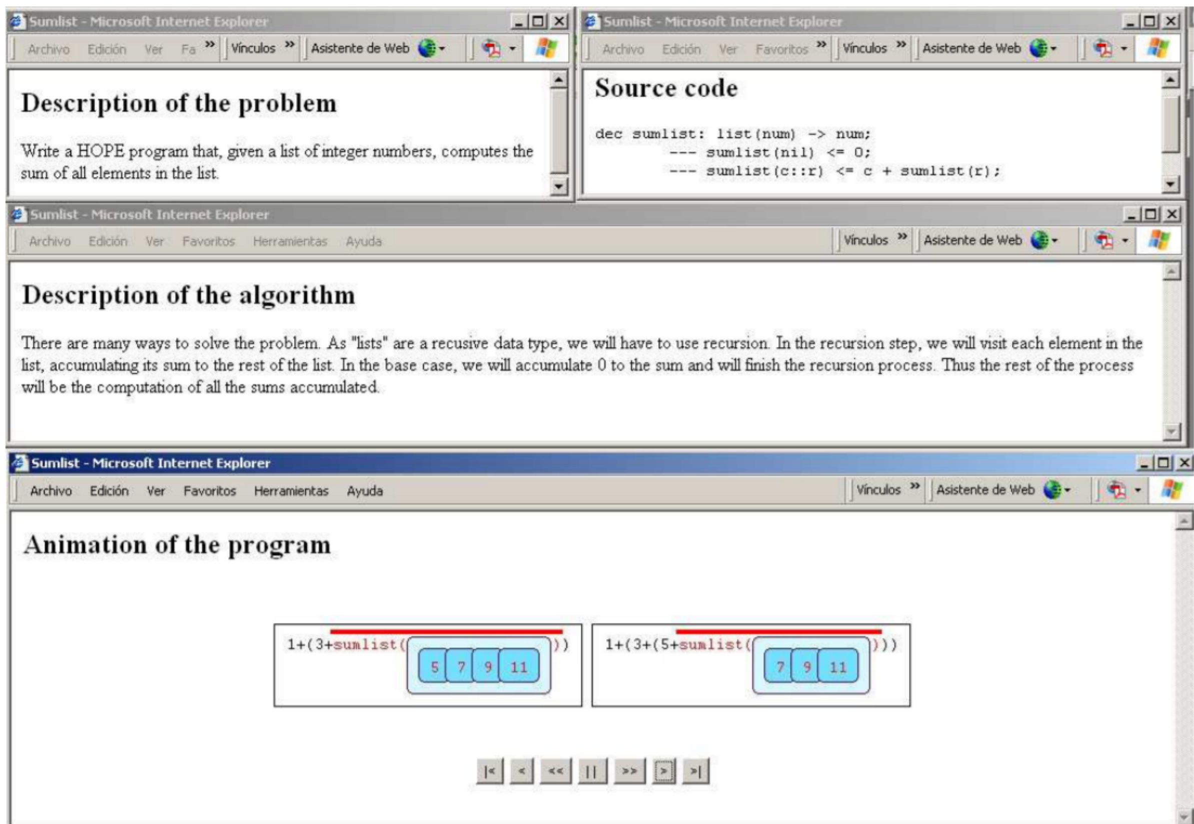


Figure 26: An example of how WinHIPE is able to offer descriptions, source code and algorithm animation. (Urquiza-Fuentes & Ángel Velázquez-Iturbide, 2007)

Show the derivation of a Haskell Expression

Evaluate! sum [3, 5, 7, 3]

Options

- (●) Show derivation with text
- ( ) Show derivation with rules
- (●) Outermost evaluation strategy
- ( ) Innermost evaluation strategy

Derivation

```
  sum [3, 5, 7, 3]
=   { Calculate the sum of a list of numbers }
  foldl (+) 0 [3,5,7,3]
=   { Process a list using an operator that associates to the left }
  foldl (+) (0 + 3) [5,7,3]
=   { Process a list using an operator that associates to the left }
  foldl (+) ((0 + 3) + 5) [7,3]
=   { Process a list using an operator that associates to the left }
  foldl (+) (((0 + 3) + 5) + 7) [3]
=   { Process a list using an operator that associates to the left }
  foldl (+) ((((0 + 3) + 5) + 7) + 3) []
=   { Process a list using an operator that associates to the left }
  (((0 + 3) + 5) + 7) + 3
=   { Add two numbers }
  ((3 + 5) + 7) + 3
=   { Add two numbers }
  (8 + 7) + 3
=   { Add two numbers }
  15 + 3
=   { Add two numbers }
  18
```

Figure 27: Evaluation of sum [3, 5, 7, 3] by HEE

Haskell Recursion Tutor

Choose an exercise:

sum [3, 5, 7, 3]

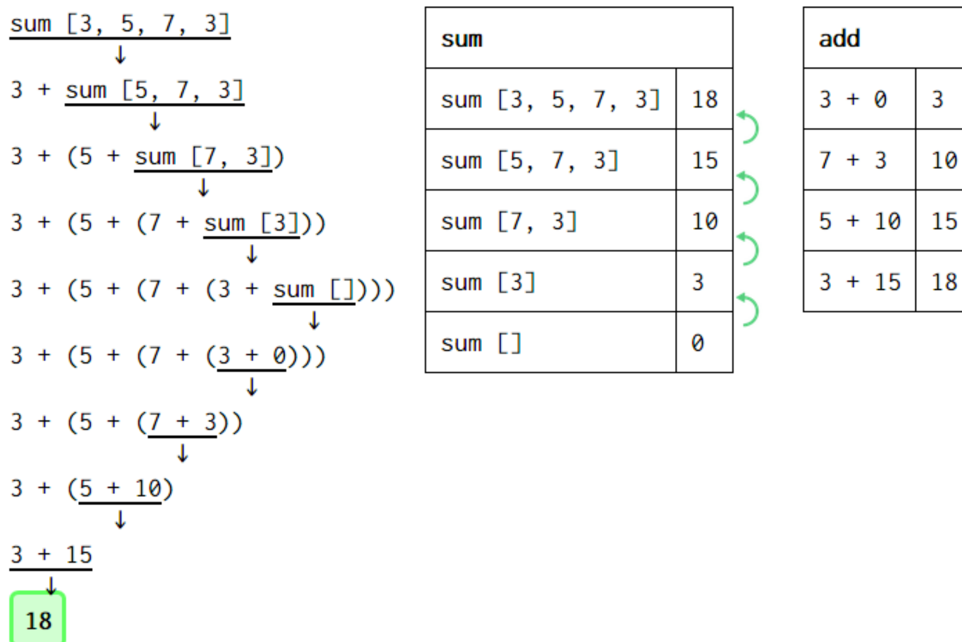
Given are following function definitions:

```
sum []     = 0
sum (x:xs) = x + sum xs
```

View the rewrite steps below:

14/14

Evaluation done. Green arrows indicate data passed back through the passive flow.



sum	
sum [3, 5, 7, 3]	18
sum [5, 7, 3]	15
sum [7, 3]	10
sum [3]	3
sum []	0

add	
3 + 0	3
7 + 3	10
5 + 10	15
3 + 15	18

Figure 28: Evaluation of sum [3, 5, 7, 3] by the prototype. After the function is fully evaluated the data passed back through the passive flow is illustrated through the usage of green arrows.

WinHIPE and HEE both have its strengths. WinHIPE is able to produce algorithm animations with little effort. The aim of this tool is to provide insight into a broad range of algorithms and to enable students to consume and create algorithm visualizations. HEE on the other hand is able to showcase rewrite steps and the difference between innnermost and outermost evaluation. But when looking at the visualizations it is clear that both tools are not designed to specifically target the weaknesses of novices about recursion. The visualization of the sumlist function by WinHIPE does not show the passive flow, except for the brackets indicating the order of operations. The copies model is not shown explicitly. HEE evaluates the sum function through the higher-order function foldl which is a step up in abstraction level than the explicit recursive functions of the proposed prototype.

We would argue that the prototype is the most apt to illustrate the copies model and the passive flow of recursive evaluation. The different function calls or the "copies" of the copies model are shown as rows in the function tables. The passive flow is illustrated through the normalization of the rows in the function tables and by employing arrows indicating the data flowing back. The exercise devised by Götschi et al. to specifically target the passive flow is also present in the prototype.

The answer to research question 1 in Section 4 showed that students struggle with identifying a correct base case and recursive step. This is not yet addressed by the tool in its current state since the tool does not offer code input. Since the input is limited, the level of engagement is also limited.

7.2. Limitations and Future Work

Input and feedback

The inability to handle input is one of the main limitations of the prototype. The prototype is only able to provide worked-out examples. In the 2DET taxonomy in Figure 22 it would share places with the least engaging tools: SREC and JSVEE. An exercise mode with input fields that checks the input against the calculated next step would be valuable. One sensible place to add input fields is in the tables or part of the next rewrite step. The exercises created in this manner would only have one possible solution, so feedback solutions might be simple. The support for student input would greatly increase the level of engagement. A mock-up of how this could look like is shown in Figure 29.

Haskell Recursion Tutor

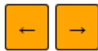
Choose an exercise:

sum [3, 5, 7, 3]

Given are following function definitions:

```
sum []     = 0
sum (x:xs) = x + sum xs
```

Complete the rewrite steps below:



4/14

```
sum [3, 5, 7, 3]
        ↓
3 + sum [5, 7, 3]
          ↓
3 + (5 + sum [7, 3])
             ↓
3 + ( ? )
```

sum	
sum [3, 5, 7, 3]	3 + sum [5, 7, 3]
sum [5, 7, 3]	5 + sum [7, 3]
sum [7, 3]	?

Figure 29: Input could be asked for in the tables or in the next rewrite step.

Another sensible way to ask for input is to let the tutor give the evaluation steps and function tables and ask the students for a function definition. Here the possible answers are multiple, instead of giving a recursive definition some users might give an answer applying the prelude function map. To support multiple solutions the proposed function could be tested on the server to check if produces the correct output, or strategy-based model tracing and property checking could be used, similar to Ask-Elle's approach.

Another option is to ask for a part of the function definition such as the base case or the recursive step. This narrows the possible input space and is thus easier to check for correctness. Such questions would be more focused on the identified difficulties by Hamouda et al. A mock-up of how this could look like is shown in Figure 30.

Haskell Recursion Tutor

Choose an exercise:

sum [3, 5, 7, 3]

Given are the evaluation steps of the function call `sum [3, 5, 7, 3]`

```
sum [3, 5, 7, 3]
        ↓
3 + sum [5, 7, 3]
          ↓
3 + (5 + sum [7, 3])
             ↓
3 + (5 + (7 + sum [3]))
                ↓
3 + (5 + (7 + (3 + sum [])))
                    ↓
3 + (5 + (7 + (3 + 0)))
```

sum	
sum [3, 5, 7, 3]	3 + sum [5, 7, 3]
sum [5, 7, 3]	5 + sum [7, 3]
sum [7, 3]	7 + sum [3]
sum [3]	3 + sum []
sum []	0

Can you provide the correct function definition?

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

Figure 30: The base case or recursive step could be asked as input.

Extension of the functions or exercises

Currently only a limited set of custom functions are supported such as the length, sum, squares, factorial and Fibonacci functions. The supported function set could be expanded. In contrast, the Haskell Expression Evaluator demonstrates evaluation steps of user provided Haskell expressions and supports the functions from the standard prelude such as

map, foldl, head, maximum, etc. The functions in the prelude are more advanced than the exercises supported in the proposed prototype. In the future these two tools could be merged into one, blending the features of the two tools: supporting custom functions such as squares and displaying function tables while supporting the prelude functions and a practice mode. Simple and more advanced exercises are then possible.

The exercises are currently taken from a predefined set. There is no interface where a teacher can add functions or exercises. Currently, to add an exercise knowledge about the structure of the internally used data type is needed. To support the management of exercises in a user-friendly manner a parser should be created that parses standard Haskell function definitions into definitions applying the data type. A content management system and GUI enabling the input of exercises would be a useful addition.

If a parser would exist this could allow students to put in their own functions to produce the evaluation steps and function tables, increasing the level of engagement. The study held with WinHIPE showed that a more active, creative interaction can lead to greater learning gains. If the input of students' functions is allowed care should be taken only supported functions are accepted and infinite recursive calls are denied.

Offering the exercises in increasing level of difficulties while storing the student's progress is another possible feature. Storing the progress requires a student model and a tracking system.

Animating the building of the function tables

Currently the rows in the function tables are normalized after the function call is fully rewritten. This choice was made to reduce the complexity of the prototype. Another option is to show the normalization during the actual passive flow when the base case is reached. This would be more accurate and could potentially offer more insight, but would also make the prototype more complex. Ideally the numbers would animate to their corresponding place as shown in Figure 31. During step 1 the number zero would animate to the row above, to the location of "sum []" and replace it.

An option would be to discard the basic arithmetic functions such as add and multiply and do the addition in one step to remove some unimportant details.

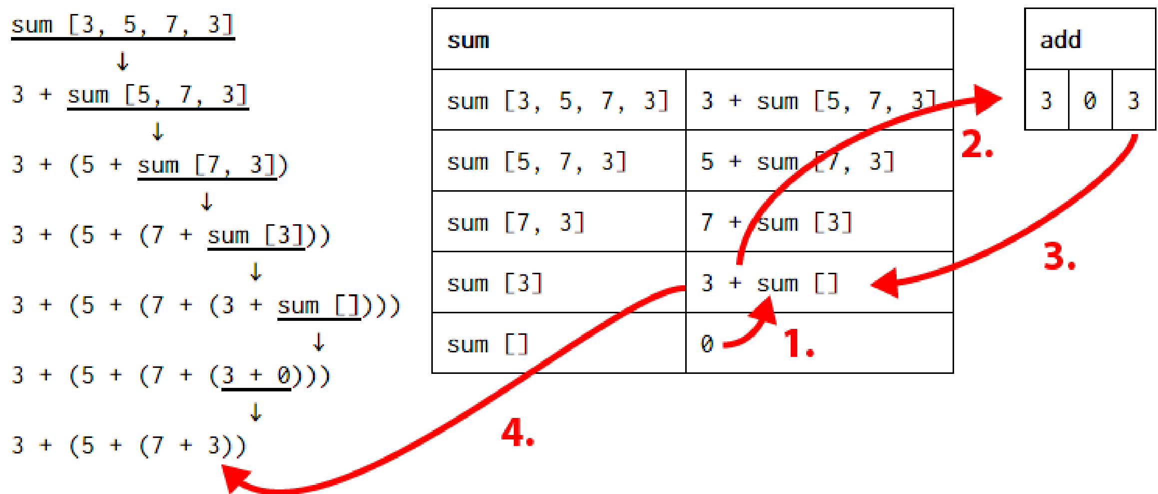


Figure 31: Suggested timing of animations for illustrating the passive flow more accurately.

Graphical Representation

WinHIPE, SREC and ChiQat tutor illustrate the evaluation of a recursive function in a diagram. This could also be a way to illustrate the recursive calls in the recursion tutor. Since the tutor is displayed in a web browser one of the many Javascript libraries could be used to display a tree representation. A library such as https://github.com/magjac/d3-graphviz supporting the dot graphics language could possibly be used for this (Gansner, Koutsofios, & North, 2006). An example image generated with dot is shown in Figure 32.

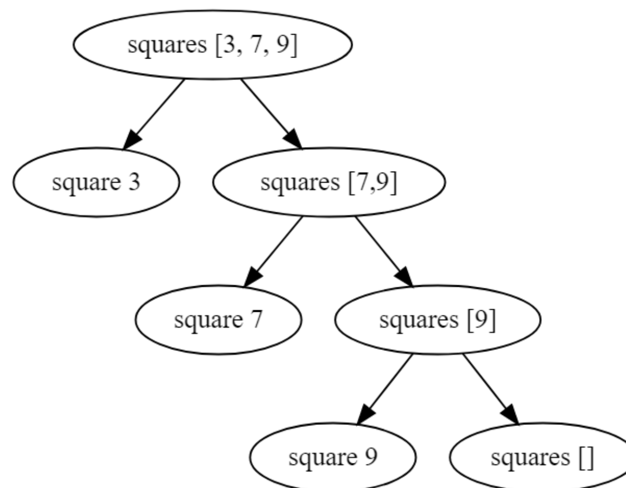


Figure 32: Graphical representation of an expression tree.

Qualitative and Effectiveness Studies

The prototype is not yet used in the wild. Some feedback on the appreciation of the tool and on certain design choices could be collected from novices through a qualitative study.

The effectiveness of the proposed tutor could be examined in a classroom setting. The

investigation of different features, with and without interaction or tree diagram, and a control group that receives the learning material in a traditional manner could be an interesting setup for an experiment.

8. CONCLUSIONS

The main research question of this document is: how can a tutoring environment support students in understanding the evaluation of recursive functions in Haskell? The main research question is subdivided in three sub-questions of which the answers are summarized below.

WHAT ARE DIFFICULTIES AND MISCONCEPTIONS ENCOUNTERED BY STUDENTS WHEN TRYING TO UNDERSTAND RECURSIVE FUNCTIONS?

In research question 1 the current literature was investigated and some of the already identified difficulties and misconceptions commonly held by novices were documented. Kahney (1983) and Götschi et al. (2003) found that experts possess the copies model as the conceptual model of recursion. Several non-viable mental models were identified. Some models, such as the looping model and the active model, are close to the copies model but show that novices often hold misconceptions about the passive flow. Hamouda et al. (2018) found novices often have problems finding a correct base case or recursive step. Ideally exercises should be offered that challenge these potential weaknesses.

HOW DO CURRENT TUTORING ENVIRONMENTS ILLUSTRATE RECURSIVE FUNCTIONS?

The literature study in research question 2 revealed several different approaches on how existing tools visualise recursive functions. Some tools employed graphical representations, such as tree diagrams, function call diagrams and stack frames, while others employed a textual approach.

The tools' abilities testified of a sizeable amount of time and effort invested by the researchers. However, the studies into the effectiveness of the tools did not reflect the same rigour, as they were often lacking.

Two studies held with control groups were examined, the studies suggested that the tools were engaging and encouraged the students to spend more time with the learning material leading to greater learning gains and higher test scores. The study with WinHIPE showed that a more active, creative interaction with the tool did lead to higher test scores than a passive, consuming one.

HOW CAN A TUTORING ENVIRONMENT BE DESIGNED TO ILLUSTRATE RECURSION IN HASKELL?

Research question 3 is the main contribution of this thesis: the proposed prototype that calculates and displays the rewrite steps and shows the rewritten sub-expressions in function tables. The function tables illustrate the copies model, every row in the table of the recursive function shows a new instantiation of the function. To our knowledge this is a unique way of illustrating the copies model. Also the passive flow is explicitly shown: the data is passed back to the previous rows to calculate the final result.

The difficulties to provide a correct base case or recursive step, identified by Hamouda et al., are not tackled by the prototype in its current stage. The handling of students' input, which could increase the level of engagement and could provide a more creative interaction, and the testing of the tool in a classroom setting remains future work.

REFERENCES

AlZoubi, O., Fossati, D., Di Eugenio, B., & Green, N. E. (2014). Chiqat-tutor : An integrated environment for learning recursion. *Proceedings of the Second Workshop on AI-supported Education for Computer Science (AIEDCS)*, pages 53 - 62.

Belland, B. (2017). *Instructional scaffolding in stem education.* Springer-Verlag.

Berghammer, R., & Milanese, U. (2001). Kiel - a computer system for visualizing the execution of functional programs. *WFPL 2001*, pages 365 - 368.

Booth, S. (1992). *Learning to program, a phenomenographic perspective.* Acta Unversitatis Gothoburgensis.

Carter, J., & Jenkins, T. (1999). Gender and programming: What's going on? *Poceedings of the Conference on Integrating Technology into Computer Science Education, ITiCSE, 31*, pages 1 - 4.

Crow, T., Luxton-Reilly, A., & Wuensche, B. (2018). Intelligent tutoring systems for programming education: A systematic review. *Proceedings of the 20th Australasian Computing Education Conference*, pages 53 – 62.

Du Boulay, B., O'Shea, T., & Monk, J. (1999). The black box inside the glass box: presenting computing concepts to novices. *International Journal of Human-Computer Studies, 51*(2), pages 265 - 277.

Fernández-Muñoz, L., Pérez-Carrasco, A., Velázquez-Iturbide, J. Á., & Urquiza-Fuentes, J. (2007). A framework for the automatic generation of algorithm animations based on design techniques. *European Conference on Technology Enhanced Learning*, pages 475 - 480.

Forrester, J. W. (1971). Counterintuitive behavior of social systems. *Theory and Decision, 2*(2), pages 109 - 140.

Gal-Ezer, J., & Harel, D. (1998). What (else) should cs educators know? *Communications of the ACM*, *41*, pages 77 - 84.

Gansner, E., Koutsofios, E., & North, S. (2006). *Drawing graphs with dot.* Technical report, AT&T Research.

Gentner, D., & Stevens, A. L. (2014). *Mental models.* Psychology Press.

Gerdes, A., Heeren, B., Jeuring, J., & van Binsbergen, L. T. (2016). Ask-Elle: an adaptable programming tutor for Haskell giving automated feedback. *International Journal of Artificial Intelligence in Education, 27*, pages 56 - 100.

Götschi, T., Sanders, I., & Galpin, V. (2003). Mental models of recursion. *SIGCSE Bull.*, *35*(1), pages 346 - 350.

Guzdial, M. (2011). From science to engineering – exploring the dual nature of computing education research. *Communications of the ACM*, *54*(2), pages 37 - 39.

Hamouda, S., Edwards, S. H., Elmongui, H. G., Ernst, J. V., & Shaffer, C. A. (2018, November). Recurtutor: An interactive tutorial for learning recursion. *ACM Transactions on*

Computing Education, 19(1).

Hamouda, S., Edwards, S. H., Elmongui, H. G., Ernst, J. V., & Shaffer, C. A. (2020). Btrecurtutor: a tutorial for practicing recursion in binary trees. *Computer Science Education, 30*(2), 216-248.

Heeren, B., & Jeuring, J. (2014). Feedback services for stepwise exercises. *Science of Computer Programming, 88*, pages 110 - 129.

Hutton, G. (2017). *Programming in haskell.* Cambridge University Press.

Kahney, H. (1983). What do novice programmers know about recursion. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '83*, pages 235 – 239.

Keuning, H., Jeuring, J., & Heeren, B. (2018, September). A systematic literature review of automated feedback generation for programming exercises. *ACM Transactions on Computing Education, 19*(1).

Kölling, M., Quig, B., Patterson, A., & Rosenberg, J. (2003). The bluej system and its pedagogy. *Computer Science Education, 13*(4), pages 249 - 268.

Lahtinen, E., Ala-Mutka, K., & Järvinen, H.-M. (2005). A study of the difficulties of novice programmers. , pages 14 – 18.

Lipovaca, M. (2011). *Learn you a Haskell for great good! A beginner's guide.* No Starch Press.

López, N., Núñez, M., Rodríguez, I., & Rubio, F. (2002). What: Web-based haskell adaptive tutor. *Artificial Intelligence: Methodology, Systems, and Applications*, pages 71 - 80.

Martinez, M., & Brizuela, B. M. (2006). A third grader's way of thinking about linear function tables. *The Journal of Mathematical Behavior*, *25*(4), pages 285 - 298.

Mayer, R. (1981, 03). The psychology of how novices learn computer programming. *ACM Computing Surveys, 13*, pages 121 - 141.

McCracken, D. D. (1987). Ruminations on computer science curricula. *Communications of the ACM*, *30*(1), pages 3-5.

Moons, J., & De Backer, C. (2013). The design and pilot evaluation of an interactive learning environment for introductory programming influenced by cognitive load theory and constructivism. *Computers & Education, 60*(1), pages 368 - 384.

Moreno, A., Myller, N., Sutinen, E., & Ben-Ari, M. (2004). Visualizing programs with jeliot 3. *Proceedings of the working conference on Advanced visual interfaces*, pages 373 - 376.

Mory, E. H. (2004). Feedback research revisited. *Handbook of research on educational communications and technology, 2nd edition)*, pages 745 – 784.

Myller, N., Bednarik, R., Sutinen, E., & Ben-Ari, M. (2009, March). Extending the engagement taxonomy: Software visualization and collaborative learning. *ACM Transactions on Computing Education, 9*(1).

Naps, T., Rößling, G., Almstrum, V., Dann, W., Fleischer, R., Hundhausen, C., ... Velázquez-Iturbide, J. A. (2003, 05). Exploring the role of visualization and engagement in com-

puter science education. *SIGCSE Bulletin, 35*, pages 131 - 152.

Nesbit, J., Adesope, O. O., Liu, Q., & Ma, W. (2014). How effective are intelligent tutoring systems in computer science education? *2014 IEEE 14th International Conference on Advanced Learning Technologies*, pages 99 - 103.

Nwana, H. S. (1990). Intelligent tutoring systems: an overview. *Artificial Intelligence Review, 4*, pages 251 - 277.

Odekirk-Hash, E., & Zachary, J. L. (2001). Automated feedback on programs means students need less help from teachers. In *Proceedings of the thirty-second sigcse technical symposium on computer science education* (p. 55–59). New York, NY, USA: Association for Computing Machinery. Retrieved from `https://doi.org/10.1145/364447.364537` doi: 10.1145/364447.364537

Olmer, T., Heeren, B., & Jeuring, J. (2014). Evaluating Haskell expressions in a tutoring environment. *Electronic Proceedings in Theoretical Computer Science, 170*, pages 50 - 66.

Pareja-Flores, C., Urquiza-Fuentes, J., & Velázquez-Iturbide, J. A. (2007, March). Winhipe: An ide for functional programming based on rewriting and visualization. *SIGPLAN Not., 42*(3), pages 14 – 23.

Phillips, D. C. (1995). The good, the bad, and the ugly: The many faces of constructivism. *Educational Researcher, 24*(7), pages 5 - 12.

Roberts, E. (1986). *Thinking recursively.* John Wiley and Sons.

Sho-Huan, T., Ching-Tao, C., Wing-Kwong, W., & Jihn-Chang, J. (2001). Visual representations for recursion. *International Journal of Human-Computer Studies, 54*(3), pages 285 - 300.

Sirkia, T. (2016, 10). Jsvee & Kelmu: Creating and tailoring program animations for computing education. *IEEE Working Conference on Software Visualization*, pages 36 - 45.

Sorva, J. (2012). *Visual program simulation in introductory programming education.* Doctoral dissertation.

Sorva, J., Karavirta, V., & Malmi, L. (2013, 10). A review of generic program visualization systems for introductory programming education. *ACM Transactions on Computing Education, 13*, pages 1 - 64.

Turner, J., & Belanger, F. (1996). Escaping from babel: Improving the terminology of mental models in the literature of human-computer interaction. *Canadian Journal of Information and Library Science, 21*, pages 35-58.

Urquiza-Fuentes, J., & Ángel Velázquez-Iturbide, J. (2007). An evaluation of the effortless approach to build algorithm animations with Winhipe. *Electronic Notes in Theoretical Computer Science, 178*, pages 3 - 13.

Uttal, W. R. (2000). *The war between mentalism and behaviorism: On the accessibility of mental processes.* Lawrence Erlbaum.

van de Pol, J., Volman, M., & Beishuizen, J. (2010). Scaffolding in teacher–student interac-

tion: A decade of research. *Educational Psychology Review, 22,* pages 271-296.

VanLehn, K. (2006). The behavior of tutoring systems. *I. J. Artificial Intelligence in Education, 16,* pages 227 - 265.

VanLehn, K. (2011). The relative effectiveness of human tutoring, intelligent tutoring systems, and other tutoring systems. *Educational Psychologist, 46*(4), pages 197 - 221.

Velázquez-Iturbide, J. A. (2000, 03). Recursion in gradual steps (is recursion really that difficult?). *ACM Sigcse Bulletin, 32,* pages 310 - 314.

Venables, A., Tan, G., & Lister, R. (2009). A closer look at tracing, explaining and code writing skills in the novice programmer. *Proceedings of the Fifth International Workshop on Computing Education Research Workshop,* pages 117 – 128.

Wilcocks, D., & Sanders, I. (1994). Animating recursion as an aid to instruction. *Computers & Education, 23*(3), pages 221 - 226.

Wu, C.-C., Dale, N., & Bethel, L. (1998, 03). Conceptual models and cognitive learning styles in teaching recursion. *ACM Sigcse Bulletin, 30,* page 292 - 296.