On the design of effective learning materials for supporting self-directed learning of programming

Ville Tirronen
Department of Mathematical Information Technology, University of Jyväskylä, Finland
ville.tirronen@jyu.fi

Ville Isomöttönen
Department of Mathematical Information Technology, University of Jyväskylä, Finland
ville.isomottonen@jyu.fi

ABSTRACT

This paper reports on the action research that studies how to implement self-directed learning of programming in the academic context. Based on our findings from the previous steps with this research agenda, we focus on the design of learning materials. That is, we aim to facilitate the students' self-directed learning by developing illustrative and concise materials that the students could use to efficiently develop theoretical understanding of the learning topics. In designing the materials, we will rely on the cognitive load theory as the guiding theoretical framework. The paper demonstrates the planning stage of our second action research cycle.

Categories and Subject Descriptors

K.3.2. [**Computers and education**]: Computers and Information Science Education—*Computer Science Education*

General Terms

Human factors, Theory

Keywords

Programming education, functional programming, self-direction, cognitive load theory

1. INTRODUCTION

A recent trend in programming education has been to give students an active role in contrast to the traditional view of the students as passive information consumers. Perhaps the best known of these approaches is the "flipped classroom" [15] or "inverted classroom" [17] where the class hours are spent on practical learning activities and the off-hours on studying theory. While the flipped classroom appears to often link with the ubiquity of video lectures and other technology, we are guided by the notion of self-directed learning and study how to best support students in managing their own learning processes. We are interested in how to promote a self-directed way of studying in the formal academic context.

We study a 6-credit master's level functional programming course by means of action research. We have completed our first action research cycle where we studied in an explorative manner what issues emerged when a very active and self-directed role was required of students. We did away with the traditional lectures, which we found to contradict with our goal of promoting self-directed learning, and the exams and grading that can impose an external motivation on students, see e.g. [27]. The course was focused on programming tasks and discussional program review.

The first research cycle gave rise to several informed research questions and required a new design of certain aspects of the course. This paper focuses on the design of learning materials. That is, for the purposes of the second action research cycle, we apply Sweller's et al. [39] cognitive load theory to the design of effective materials, by which we aim to further facilitate the students' self-directed learning.

2. THE CURRENT STATE OF OUR ACTION RESEARCH

As stated in the introduction, we study how we can support self-directed learning in the academic context. As the ability to be self-directed can vary with the content [20], it must be noted that our research concerns functional programming which has been found to be a 'level field' for the students [23, 11]. We started the research based on the observations that understanding the difficult topics of functional programming would require more hands-on practice than we could introduce with a lectured course. In addition, we experienced that learning of syntax and basic structures with the Haskell FP language would not require lecturing. We wanted to avoid this kind of waste of the students' and the teacher's resources and became informed by the notion of self-directed learning. Taking an action research approach, we search for such course arrangements that would prompt and support self-directed study where time is expended on learning rather than on 'waste'.

Action research is typically a cyclic process where each cycle involves planning, action taking, and reflection [30]. The initial research plan is refined and understanding developed over the cycles conducted. We have implemented our first action research cycle, where we

1. planned our research, which we discuss in [40] and summarize with Figure 1 (*planning*),

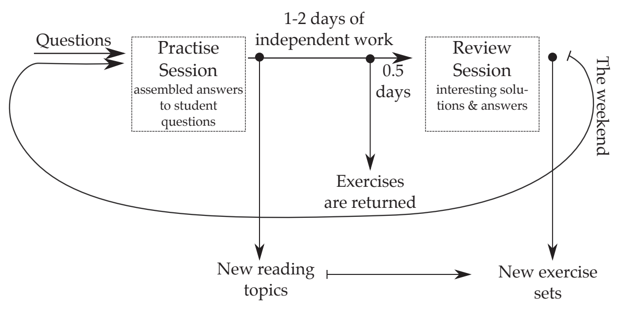

Figure 1: The course model during the first action research cycle

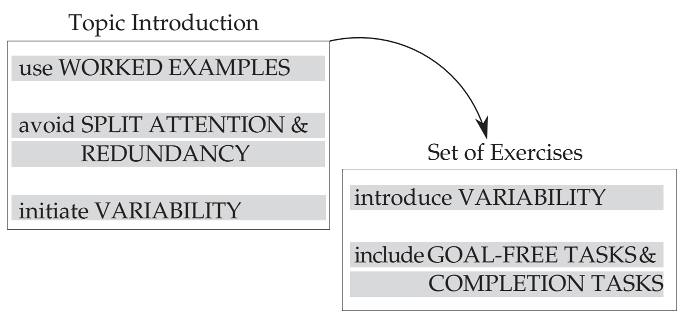

Figure 2: Our material design and the cognitive load theory

2. implemented the course according to our plans, when we also collected research data (*action taking*), and
3. analyzed the data collected by the first action taking phase (*reflection*).

For the first cycle we adopted an explorative research approach, which we associate with the 'practical action research' within the usual three-part action research taxonomy: technical, practical, emancipatory; see, for example, [9, 21, 12]. We surveyed the students' subjective course experiences three times during the course and considered the resultant student view of the course in relation to our teaching experiences. We wanted to develop an understanding of the key issues with our course model to form informed research questions for the subsequent research cycles.

During the first cycle (autumn 2011) we had no lectures, exams, or grades, and we relied on group work (peer learning). The course was driven by weekly exercise sets and student work was supported by two weekly contact sessions, as depicted in Figure 1. The first session was meant to give students support on the basis of the questions they would raise. Thus we expected students to drive the course by sending questions to us at a certain time during the week. The second session was a discussional program review that summarized the weekly learning topics in a reflectional manner.

Our analyses of the data indicated that the students preferred the course arrangements they were provided with when contrasted with more traditional ones. We also saw an increase in the pass rate: previously these were 61 % (2008), 43 % (2009), and now 81 % (2011). However, considerable difficulties were also encountered. We found three areas that would require careful attention to better prompt self-directed learning: group work, learning materials, and how to better support individual learning processes (the different paces of learning).

With regard to learning materials, our analyses indicated that the students found it difficult to form a synthesis of the learning topics when we provided them with the weekly exercise sets and they had to collect necessary information from the multiple information sources. Making a proper synthesis of the learning topics with such self-study was perceived to be difficult and too time-consuming. This raised our interest to support self-directed learning with integrated and concise learning materials that would make the difficult concepts of the Haskell language approachable in sufficiently short study time intervals. In this paper, we apply the cognitive load theory to the design of effective learning materials, by which we aim to improve the students' possibilities for self-directed learning. This demonstrates the planning stage of our second action research cycle.

Before proceeding, we must note that while we here refer to action research our subjective experience is that we are also heavily involved with design activity in which we engineer our learning environment (the FP course). For this reason our work is likely to link with design-based research in an educational setting as discussed for example by Brown [7]. We leave these methodological questions for a future study.

3. DESIGN OF MATERIALS

During the first action research cycle, our course consisted of week-long cycles where the students' work was driven by weekly exercise sets as described in Section 2. In the next implementation of the course, we link each week with a set of learning modules. The materials in each module consist of a *topic introduction* and a *set of exercises*, which we illustrate in Figure 2. In this section, we discuss the design of the materials from the viewpoint of the cognitive load theory [39].

3.1 Cognitive load theory

The Cognitive Load Theory (CLT) is a major theory employed to support instructional design [36]. The CLT, discussed for example by Sweller et al. [39, 36], is based on the human memory model that consists of working memory and long-term memory (see e.g. [4]), and the assumption that the working memory can only hold a limited number of items [34]. Sweller et al. rely on the hypothesis that the size, complexity, and sophistication of the items held in the working memory are not constrained. For example, the learning occurring over a long period of time can result in schemas that can incorporate a huge amount of information which would otherwise overload the working memory.

The cognitive load refers to how our cognitive system, and especially our working memory, is challenged with various tasks. The CLT differentiates between intrinsic, extraneous, and germane cognitive load. The first is characterized by the subject itself. For example, mathematics has a high intrinsic cognitive load because mathematical formulas have a lot of interdependencies between their elements, which must be learned as a whole. The extraneous cognitive load refers to the effort required to process poorly designed instructional designs, and is thus an unwanted effect in instruction. The germane cognitive load refers to such cognitive tasks that can contribute to the construction of useful and transferable knowledge (schemas).

A central aspect in the CLT is that managing certain cognitive tasks may require automation of certain other lower level tasks. As a result of automation working memory resources are not expended on low-level tasks (e.g. interpreting individual letters while reading a text) but can be focused on higher-level processes (e.g. interpreting meanings in a text). A more recently discussed aspect is that when considering the cognitive load in a particular situation it is necessary to take account of the learners' degree of expertise with the domain in question, and consider how learning can be scaffolded [24].

3.2 Applying the CLT

In designing the learning materials we are interested in replacing the extraneous cognitive load with the germane cognitive load, the latter of which should help students to comprehend the challenging theoretical topics of functional programming. To this end, we study several of the approaches to instructional design discussed by Sweller et al. [39]: the goal-free effect, the worked-example effect, the completion-problem effect, the split-attention effect, the redundancy effect, and the variability effect. In addition, we make use of the idea of the automation of schemata and scaffolding in our learning materials.

3.2.1 Split-attention effect

The split-attention effect refers to the learner's need to mentally integrate between multiple sources of information, which can increase the unwanted extraneous cognitive load.

During our first action research cycle the students wanted to see more examples although examples were available in multiple information sources that we suggested for them. This indicated that the examples available in various forms and in various places did not sufficiently benefit the students' learning. We conjecture that this issue arises from the split-attention effect. We now build our own integrated topic introductions, which should independently suffice as learning material.

3.2.2 Redundancy effect

The redundancy effect refers to the use of several illustrations each of which can independently prompt the development of schemas about the learning topic in question. With redundancy the students can be confused as they must consider the relationship between the overlapping illustrations. There was a high risk of this happening in our previous research cycle and we are now informed to build learning materials with no redundancy.

3.2.3 Worked-example and completion-problem effects

The worked-example effect indicates that the unwanted cognitive load can be smaller with worked examples than with solving equivalent problems. Reducing unwanted load can leave more room for tasks that introduce the germane cognitive load.

In the previous research cycle, we noticed that the students wanted to have such examples from which they could imitate the way of approaching the problems and simple examples that would clarify difficult FP/Haskell concepts. We concluded that as we had provided students only with conventional programming exercises it was difficult for them to develop a sufficient theoretical synthesis of the learning topics in the time they could allocate for studying the topic. This again made it very difficult for them to make proper questions of the topics and their self-directed way of learning became constrained. To alleviate this problem we now make use of the worked-example effect.

In our topic introductions (see Figure 2) we will use concise worked examples complemented with annotations that point to the certain aspects in the examples, which together should help the students to draw a viable mental model of the topic under study. We will discuss our use of the worked-example effect more in the Section 4.1.

A worked example can be turned into a completion problem by omitting certain parts that the learner must fill in. The learner is required to carefully study the example in order to develop the schemas needed to finish it. This approach has been found to introduce less extraneous cognitive load than traditional exercises, see e.g. [42]. We rely on the worked examples in our topic introductions and continue with completion problems in the exercise sets.

3.2.4 Goal-free effect

The goal-free effect states that the problems with no explicitly set goals have been found to better contribute to the schema construction in comparison with the goal-oriented tasks. This matches well our idea of student-driven learning where we encourage an explorative way of studying.

3.2.5 Variability effect

We assume that the guidelines above reduce the extraneous cognitive load, which in turn allows us to increase the germane cognitive load. To this end, we can rely on the concept of variability in the CLT, which means that variation in the problem solving situations leads to better transferability of knowledge (schemas). The improved transferability results from the fact that the learner is prompted to identify the relevant and irrelevant elements with the topic studied. We can introduce the variability effect in the end of the topic introductions and continue with the same idea when designing the exercises; see Figure 2.

3.2.6 Automation of schemas

Haskell is a new approach to programming for most of our students. For this reason, we want to induce automatization of certain rote tasks such as reading type declarations and basic syntax conventions. For example, we always show type declarations in conjunction with the function implementations and prepare exercises where type declarations are practiced by means of symbol manipulation. The sooner the students can fluently manage these low level tasks, the more resources become available for the primary topics of the course.

3.2.7 Scaffolding

As it is suggested in the CLT literature [2], we can introduce scaffolding by starting with instructions and worked examples and progressing through completion problems to authentic programming tasks. This progression will take place across our topic introductions and exercise sets so that towards the end of each exercise set we introduce traditional programming tasks. In our view, the traditional programming tasks are needed to reinforce occupational skills.

3.2.8 Summary

During the first action research cycle we provided the students only with traditional exercises, which required the students to search for appropriate examples and information from multiple sources. With our new approach our topic introductions already consist of examples and related illustrative annotations. The exercises then introduce more variability effect to the learning material, which should further reinforce the schemas the students develop from the example-based topic introductions. We will give an example of the materials in one course module in the next section.

4. EXAMPLE

4.1 The topic introduction

In this section, we illustrate the construction of a CLT-informed topic introduction (see Figure 2). This topic introduction covers the concept of 'functors' in the Haskell language and is presented here with the same structure of five steps as it is presented to students. In this topic introduction, we identify the crucial aspects that characterize this concept and express them step by step, aligning with the suggestion to explicate problem stages with worked examples [39]. Further, we develop the material by use of illustrative code segments complemented with annotations (text) emphasizing the key features in the code to the students, which is also is supported by work on cognitive modelling [1].

It is important to note that the material in this section can be seen as one large worked example of developing higher-order abstractions in the Haskell language. The study of such abstractions is an important topic as effective functional programming typically requires an understanding of a variety of abstract concepts such as categories, monads, and combinator libraries.

The first four steps presented below attempt to prompt a proper schema in the mind of the student, and the connection with CLT therein is the attempt to minimize the redundancy by taking a narrow focus on the essential concepts. Furthermore, the essential prerequisite knowledge is repeated at the beginning of the topic introduction, which reduces the learners need to split their attention to other parts of the course material.

In the final step, we aim to prompt learning by comparing several small examples. This draws on the variability effect in the CLT where the key idea was that variation in problem situations helps a learner to identify relevant aspects in the learning topic. Thus, the last step in this introductory material prompts students to observe very different instances of the functor concept and to see in what kind of situations they can be used. This extends the 'what constitutes as a functor' schema developed by the previous steps. We give further notes on the connections between our material (topic introduction and exercises) and CLT along with the presentation of our functor example below.

Before the functor learning module students have been taught the basic constructs of the language and they should be familiar with all the data types and definitions used here. For the benefit of the reader, the essential syntax used in this section consists of

1) ordinary definitions denoted by "=" with name on the left and the expression on the right,
2) type definitions denoted by symbol "::" with the name on the left and the type on the right, and
3) function type declarations, denoted by "a -> b" that describe functions from set a to set b.

Further, readers unfamiliar with the ML family of languages should take note that the function call (and the definition of) is denoted by juxtaposition instead of parenthesizing the function argument. Notice also that we typeset the quotations from the material with the italic typeface and the code samples with a fixed width font.

4.1.1 Step 1

The topic introduction to functors starts in an instructive manner, by referring to lists and map function, which capture the most basic aspect of a functor, and hopefully initiate the proper schema in the minds of students.

In this module we study a broad class of data types that are called functors. In short, functors are types that can be "mapped over" much in the same way as lists are mapped over by the function map. Let us remind ourselves of how this function is used:

```
map reverse ["chopin", "shubert", "handel"]
 == ["nipohc","trebuhs","lednah"]
```

Notice that we begin with a simple and instructive example that is familiar to students and can initiate the process of scaffolding.

The students are then shown type declarations for the mapping operations defined for lists, streams of values, and binary trees. This demonstrates the general form of the map function.

```
map       :: (a -> b) -> [a] -> [b]
mapStream :: (a -> b) -> Stream a -> Stream b
mapTree   :: (a -> b) -> BinaryTree key a
                      -> BinaryTree key b
```

To emphasize the similarity of the above operations, we show how to abstract over the concrete data type names:

```
type X a = [a]
type Y a = Stream a
type Z a = BinaryTree String a
```

We then repeat the previous map definitions to give the students the general pattern of a functor:

```
map       :: (a -> b) -> X a -> X b
mapStream :: (a -> b) -> Y a -> Y b
mapTree   :: (a -> b) -> Z a -> Z b
```

Presenting these declarations relies on the CLT variability effect as the students are able to observe the shared pattern between the different operations. We finally sum up the essence of the functor with the following annotation:

Notice that the only difference in the above definitions relates to X, Y, and Z: the first is a list, the second a binary tree with string keys, and the third a stream. Compare here the fact that we can have integers and floats that have operations for summing them with the fact that we can have lists, binary trees, and streams that have operations for mapping *them. We call the first constructs 'numbers' and the latter 'functors'. It is this existence of mapping operation that makes a data type a functor.*

Within this annotation we use a simple metaphor related to numbers, which we argue to loosely link with the variability effect.

Finally, we give the formal Haskell definition for the functor concept, on which the students can base their work:

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

4.1.2 Step 2

In the previous step, all of our examples were, in a sense, containers. At this point we prompt the students to notice that according to the previous definitions, not all containers are functors. We demonstrate this by showing how a simple mapping operation that uses a function "intToString", cannot be implemented for a data structure that can only contain numbers:

Observe that with the fmap operation it must be possible to change the data type of the elements inside a container, but the type of the container cannot be changed. For example, we can produce a list of strings from a list of integers, but we cannot produce a binary tree from a list by using fmap.

In the following we demonstrate how this aspect limits functors to data types that are parametric. Let us consider a non-parametric data type that is used for representing 3D vectors in a linear algebra library:

```
newtype Vector = Vec (Double,Double,Double)
```

We cannot define a general mapping operation for our 3D vector. For example, if we consider mapping the vector with the function "intToString", the result would have to contain strings instead of doubles, and it wouldn't be our 3D vector anymore.

```
intToString :: Int -> String
intToString = ..

mapVector intToString (Vec (1,0,1))
 == ??
```

Thus our 3D vector is not a functor.

The step 2 continues our main worked example, adding a new aspect to the functor concept. Additionally, this step can be seen as a small worked example where annotation is used to highlight the point under study.

4.1.3 Step 3

Next we prompt the students to notice that the functor is a more general pattern than mapping over the elements in a container, e.g. in a binary tree. Now we need to show students that a functor is a more abstract concept than indicated with the previous steps:

Yet another aspect of functors is that they might not be containers at all, which is demonstrated by the fact that even ordinary functions fit in the pattern of functors:

```
instance Functor (x ->) where
-- fmap :: (a -> b) -> (x -> a) -> (x -> b)
   fmap f g = f . g
```

That is, the function composition (f . g) can also be thought of as a mapping operation in which the change is applied to the result value of the function being mapped over. For example, we can define an operation that changes the type of the result of a function into a string (which is handy for printing things on the screen):

```
intToString :: Int -> String
intToString = ..

makePrintable :: (x -> Int) -> (x -> String)
makePrintable = fmap intToString
```

In the following we apply such an operation to a function that calculates the length of a list:

```
length :: [a] -> Int
length = ..

lenghtAsString :: [a] -> String
lenghtAsString = makePrintable length
```

Similar to Step 2, our functor worked-example is continued by adding a new aspect to it. We do this by developing a small worked example with annotations. We would also connect this step with the variability effect as we give an example that is useful in practice.

4.1.4 Step 4

We continue our on-going worked example by attempting to capture the essence of a functor with laws that define its behavior. We give students an example of a misleading functor definition that should motivate them to consider these laws.

Observe that there are often many well-typed definitions for fmap. Semantically, however, not all of them make sense. For example, let us consider a somewhat suspicious functor definition for lists:

```
instance Functor [] where
  fmap f xs = reverse (map f xs)
```

This definition is valid Haskell code, but it does not fulfill the intuition of what mapping should do. We do not want the mapping operation to reverse the order of the list as this would make it quite confusing to use. To avoid such confusing definitions, we propose that there are two laws that each functor definition should satisfy:

```
fmap id       == id                -- fmap 1
fmap (f . g)  == (fmap f . fmap g) -- fmap 2
```

Together, these laws say that the functor does not change the structure *of the results, but only the contents of it. In the following we use these laws to show that our previous confusing example is not a valid functor definition:*

```
fmap id [1,2,3]
= {- Definition of the 'confusing fmap' -}
reverse (map id [1,2,3])
= {- fmap 1. applied to list and map -}
reverse [1,2,3]
= {- what reverse does -}
[3,2,1]
```

which breaks the first functor law, since [3,2,1] ≠ id [1,2,3].

Notice that it is difficult to instruct the compiler to verify the functor laws, which is why the task of following and verifying them rests on the programmer.

This step relies on symbol manipulation in illustrating the functor laws to the students. We carefully annotate each line using the proof notation style of Dijkstra to avoid introducing extraneous cognitive load. We also aim to scaffold students towards exercises where they need to complete certain points in annotated proof tasks.

By knowing the functor laws the students can use and reason about any functor without a need to know the implementation. This is the essence of modular computer programs.

4.1.5 *Step 5*

We conclude the topic introduction by giving students a couple of examples of functors in action. To further distill the essence of what is a functor, we attempt to provide several very different examples of functors. These include containers, like two-dimensional arrays:

```
intToDouble :: Int -> Double
intToDouble = ..

toDoubles :: Array (Int, Int) Int
          -> Array (Int,Int) Double
toDoubles = fmap fromIntegral
```

and different types of IO computations:

```
getContents :: IO String
getContents = ..

countWordsFromStdin :: IO Int
countWordsFromStdin = fmap (length . words)
                           getContents
```

and computations that can fail, which occur, for example, when doing lookups from a key-value container:

```
lookup :: Name -> Phonebook -> Maybe Number
lookup = ..

takeAreaCode :: Number -> AreaCode
takeAreaCode = ..

lookupAreaCode :: Name -> Phonebook
                  -> Maybe AreaCode

lookupAreaCode n pb
  = fmap takeAreaCode (lookup n pb)
```

Together with previous steps above, these should give students basic examples about the actual use of functors, while utilizing the variability effect to attain more germane cognitive load in the form of identifying similarities between vastly different things such as computations and containers. We might advance this step later by asking the student to explicate the common part of the items presented here.

4.2 Exercises

This sections gives examples of the exercises and their progression in the functor module.

4.2.1 *Exercise 1*

In the first exercise we rely on the goal-free effect by asking the students to consider an overly verbose bit of a code that can be improved by, among other things, using functors:

How would you improve the following code? What kind of benefits can you achieve by your changes?

```
display :: Maybe Email -> String
display = ..

getEmail :: Account -> Email
getEmail = ..

displayUserEmail :: BinaryTree User Account
                 -> User -> String
displayUserEmail db u
    = let account = lookup db u
      in display (case account of
                    Just e  -> Just (getEmail e)
                    Nothing -> Nothing)
```

This kind of exercise is also supported by a student difficulty we observed in the previous instances of the course, namely, the lack of skill to refine the programs by using abstract FP/Haskell concepts. Observations related to the student difficulty of doing assignments well have been made in the context of other languages as well [28].

4.2.2 *Exercise 2*

A more explorative case of the goal-free task is presented below. We assume this kind of exercise to be applicable for group work and to promote self-directed mood in the students' study.

Try to discover as many types as you can that allow a proper definition of a functor. I.e. One that satisfies the functor laws:

```
fmap id      == id                -- fmap 1
fmap (f . g) == fmap f . fmap g   -- fmap 2
```

Discuss how 'fmap' would behave for each of the types and how you could use that behaviour in your programs.

4.2.3 *Exercise 3*

This exercise relates to the functor laws given in Section 4.1. It is designed to train the student in symbolic manipulation of Haskell programs, which is a skill that should become automatic. We hope that doing symbolic manipulation by hand efficiently supports the process of automatization.

This exercise is directly informed by the completion-problem effect. This form of problem statement is chosen due to the observation that symbolic manipulation and proof problems are usually the exercises least preferred by the students. Further, we have observed that lengthy examples that are heavily based on such manipulations are often studied carelessly, which contributes to difficulties in absorbing this topic. Here, the completion problem hopefully avoids such an effect as doing the exercise includes studying the embedded example.

To make this exercise as easy to approach as possible, we take into account the split-attention effect and include all the necessary equations in the start of the exercise, whereas in the previous course instances we might have considered finding the appropriate definitions to be a learning task by itself.

Complete the proof of the following theorem:
Theorem 1. *The definition of the map function,*

```
map _ []     = []              -- map 1
map f (x:xs) = f x : map f xs -- map 2
```

satisfies the second functor law, i.e.

```
map (f . g) == (map f . map g)
```

Proof. *We prove this by induction over the structure of the list. First of all, we show that the theorem holds for an empty list and two arbitrary functions f and g:*

```
 map (f . g) []
== {- Map 1 -}
 []
== {- Inverse of Map 1 -}
 map f []
== {- EXPLAIN WHY -}
 map f (map g [])
== {- Inverse of definition of (.) -}
 (map f . map g) []
```

Secondly, we show that if the theorem holds for a list of a particular length, it must also hold for a list that is one element longer, and thus, by the induction principle, for all lists. This follows from showing that

```
map (f . g) xs == (map f . map g) xs
```

implies

```
map (f . g) (x:xs) == (map f . map g) (x:xs)
```

This can be proved by a simple calculation:

```
 map (f . g) (x:xs)
== {- Map 2 -}
 (f . g) x : map (f . g) xs
== {- The inductive assumption -}
 (f . g) x : (map f . map g) xs
== {- EXPLAIN WHY -}
 f (g  x) : (map f (map g xs))
== {- EXPLAIN WHY -}
 map f (g x:map g xs)
== {- EXPLAIN WHY -}
 map f (map g (x:xs))
== {- Definition of (.)-}
 (map f . map g) (x:xs)
```

This proves the theorem.

The style of proof notation is the same as the one used in Step 4 of our topic introduction. This notation gives an explanation for each manipulation step, which we find to align well with CLT as no additional material is needed to understand the proof and the split attention is avoided.

4.2.4 Exercise 4

The final exercise continues the theme of the previous proof exercise so that the students now prove properties of their own code. This is a traditional exercise that we hope to have scaffolded with the examples in our topic introduction and the previous exercises.

Define a functor instance for the following binary tree data type and show by induction that it obeys the functor laws.

```
data BinaryTree key value
 = Tip |
   Branch key value
           (BinaryTree key value)
           (BinaryTree key value)
```

We have progressed from a simple map function for lists, given at the beginning of our topic introduction, to a traditional exercise that the students should complete independently. With earlier instances of the course, we would have started weekly assignments with such an exercise. Now, we hope to have scaffolded the learning sufficiently so that the completion of this exercise is not too difficult.

4.3 Evaluation of our material design

In order to support our action research we must ascertain that the topic introductions and exercises such as the ones given above can control the cognitive load. The cognitive load can be measured either by physiological instruments, performance based evaluations, or subjective estimates; see [39]. Of these only the subjective estimation is arguably practical in an authentic course context and has been found to be comparable in accuracy to the other measurement schemes [18, 3]. We will adopt the simple subjective estimation scheme devised by Ayres [3] where the cognitive load of individual tasks in each exercise is measured using a seven-point scale ranging from 'extremely easy' to 'extremely difficult'. We include such a subjective question after each step in our topic introductions and after each exercise in the learning module.

In addition, we repeat the surveys of the first action research cycle. These surveys included general questions such as "Is there enough learning material/information available to support your learning? What has been the biggest help during the course? Which of the exercises have been the most useful...least useful ones?" This will inform us of whether the anxiety about the use of materials has diminished and how the materials fit the students' general impression of the course.

5. RELATED WORK

Design of instructional materials for programming courses is an active topic of study from where we can identify five themes. Firstly, there is the theme best characterized by the name "nifty assignments", the representative example of which is the collection of successful CS1/CS2 assignments published by Parlante [37]. These assignments are exercises that have been polished enough to be shared and have often been through considerably many iterations of courses and students. A good example of a "nifty assignment" is the Boggle word-game utilized by Feldman and Zelenski [13] to teach recursion in the context of CS1/2 courses. This exercise has been continually improved for over a decade.

Secondly, we find a theme categorized by the idea of enlivening the course assignments by the inclusion of an interesting topic, such as scientific experimentation [5] or advanced CS topics [22]. One subclass of this type of assignments are the various approaches dealing with image processing and machine vision [33, 32, 41, 8], or, as in one extreme attempt of grabbing the attention of the students, processing images from Mars [14].

For the third theme, we have instructional material design that incorporates special topics, such as security awareness [31], discrete mathematics [29], or algorithm efficiency [16] into the curriculum. This is usually done in order to emphasize the additional topic instead of enlivening the learning of the original course content.

As the fourth theme, there is a class of papers that study how to support social interplay and cooperative learning through the design of instructional materials, exemplified by Leland et. al. [6], who devise their exercises for student groups. In the exercises each member has a distinct role, such as a "manager of variables". This line of research is supported by psychological studies such as observing the students' emotional experiences of programming [26]. experiences of programming students are investigated.

The articles that most resemble our current work form a theme characterized by the application of a learning theory to the design of instruction. This theme is well characterized by works that apply problem-based learning on computer science curriculum [25], presenting larger problems such as AI programming to the students [35]. The cognitive load theory also appears in this theme. Gray et al. [19] focus merely on this theory by decomposing programming procedures into subtasks and using worked examples with scaffolding. Caspersen and Bennedsen [10] use the CLT in conjunction with other compatible learning theories.

Our approach in this paper differs from the above themes by specifically aiming to enable self-directed learning on the part of students by the use of the CLT in material design. We find the CLT to be a proper framework that allows us to measure our success with other than pure performance based metrics. Further, as observed in the survey by Pears et. al. [38], the abundance of studies concerning individual learning settings is no longer sufficient in advancing the field, which is a fact that we find to encourage studies based on theory such as CLT.

6. CONCLUDING REMARKS

In this paper we detailed our ongoing action research that aims to enable a self-directed way of studying in the formal academic context. The paper illustrated the planning stage of our second action research cycle where we also identified with the design-based research. We focused on the design of learning materials using the cognitive load theory as our theoretical framework. We hope to promote self-direction by removing unwanted cognitive load in the use of the materials. To this end, we demonstrated a re-design of a single problematic module in the previous course instance in accordance with the cognitive load theory. Through this kind of design efforts we are prepared to continue our investigation on self-directed study in the academic setting.

We noticed that utilizing CLT in a specific context was difficult. In our experience, teachers have to first carefully consider what kind of topic they are teaching in order to develop an understanding of how to benefit from CLT in their instructional design. For example, when teaching abstract concepts such as our functor example, it was not originally very clear what could constitute as worked examples. We concluded that we should associate the development of each abstract concept to one large worked example, where problem stages are clearly delineated and explained with smaller annotated worked examples.

7. REFERENCES

[1] J. Anderson, C. Boyle, A. Corbell, and M. Lewis. Cognitive modelling and intelligent tutoring. *Artif. Intell.*, 42:7–49, 1990.

[2] R. Atkinson, A. Renkl, and M. Merrill. Transitioning from studying examples to solving problems: Effects of self-explanation prompts and fading worked-out steps. *Journal of Educational Psychology*, 95(4):774, 2003.

[3] P. Ayres. Using subjective measures to detect variations of intrinsic cognitive load within problems. *Learning and Instruction*, 16(5):389–400, 2006.

[4] A. D. Baddeley. *Human Memory: Theory and Practice*. Lawrence Erlbaum Associates, Hove, UK, 1990.

[5] D. Baldwin and J. Koomen. Using scientific experiments in early computer science laboratories. In *ACM SIGCSE Bulletin*, volume 24, pages 102–106. ACM, 1992.

[6] L. Beck, A. Chizhik, and A. McElroy. Cooperative learning techniques in CS1: design and experimental evaluation. *ACM SIGCSE Bulletin*, 37(1):470–474, 2005.

[7] A. L. Brown. Design experiments: Theoretical and methodological challenges in creating complex interventions in classroom settings. *The Journal of the Learning Sciences*, 2(2):141–178, 1992.

[8] K. Burger. Teaching two-dimensional array concepts in Java with image processing examples. In *ACM SIGCSE Bulletin*, volume 35, pages 205–209. ACM, 2003.

[9] W. Carr and S. Kemmis. *Becomming Critical: Education, Knowledge and Action Research*. The Falmer Press, London, 1986.

[10] M. Caspersen and J. Bennedsen. Instructional design of a programming course: a learning theoretic approach. In *Proceedings of the third international workshop on Computing education research*, pages 111–122. ACM, 2007.

[11] M. Chakravarty and G. Keller. The risks and benefits of teaching purely functional programming in first year. *Journal of Functional Programming*, 14(01):113–123, 2004.

[12] T. Clear. *Critical Enquiry in Computer Science Education*, chapter 2. Routledge Falmer, 2004.

[13] T. Feldman and J. Zelenski. The quest for excellence in designing CS1/CS2 assignments. In *ACM SIGCSE Bulletin*, volume 28, pages 319–323. ACM, 1996.

[14] H. Fell and V. Proulx. Exploring martian planetary images: C++ exercises for CS1. *ACM SIGCSE Bulletin*, 29(1):30–34, 1997.

[15] J. Foertsch, G. Moses, J. Strikwerda, and M. Litzkow. Reversing the lecture/homework paradigm using eTEACH web-based streaming video software. *Journal of Engineering Education*, 91(3):267–274, 2002.

[16] J. Gal-Ezer, T. Vilner, and E. Zur. Teaching algorithm efficiency at CS1 level: A different approach. *Computer Science Education*, 13(3):235–248, 2004.

[17] G. Gannod, J. Burge, and M. Helmick. Using the inverted classroom to teach software engineering. In *ICSE '08. ACM/IEEE 30th International Conference on*, pages 777–786, May 2008.

[18] D. Gopher and R. Braune. On the psychophysics of workload: Why bother with subjective measures? *Human Factors: The Journal of the Human Factors and Ergonomics Society*, 26(5):519–532, 1984.

[19] S. Gray, C. St Clair, R. James, and J. Mead. Suggestions for graduated exposure to programming concepts using fading worked examples. In *Proceedings of the third international workshop on computing education research*, pages 99–110. ACM, 2007.

[20] G. O. Grow. Teaching learners to be self-directed. *Adult Education Quarterly*, 41(3):125–149, 1991.

[21] S. Grundy. Three models of action research. In S. Kemmis and R. McTaggart, editors, *The Action Research Reader*, pages 353–364. Deakin University Press, Geelong, third edition, 1990. Reprinted from Grundy, S. (1982), Three Models of Action Research. Curriculum Perspectives, 2(3), 23–34.

[22] G. Holmes and T. Smith. Adding some spice to CS1 curricula. In *ACM SIGCSE Bulletin*, volume 29, pages 204–208. ACM, 1997.

[23] J. Hughes, D. Spoonhower, G. Blelloch, R. Harper, P. Gibbons, C. Pareja-flores, J. Urquiza-fuentes, J. Velázquez-iturbide, R. Ennals, D. Gay, et al. Experiences from teaching functional programming at Chalmers. *ACM SIGPLAN Notices*, 43(11):0, 2008.

[24] S. Kalyuga, P. Ayres, P. Chandler, and J. Sweller. The expertise reversal effect. *Educational Psychologist*, 38(1):23–31, 2003.

[25] J. Kay, M. Barg, A. Fekete, T. Greening, O. Hollands, J. Kingston, and K. Crawford. Problem-based learning for foundation computer science courses. *Computer Science Education*, 10(2):109–128, 2000.

[26] P. Kinnunen and B. Simon. Experiencing programming assignments in CS1: the emotional toll. In *Proceedings of the Sixth international workshop on Computing education research*, pages 77–86. ACM, 2010.

[27] B. Klug. To grade, or not to grade: A dramatic discussion in eleven parts. *Studies in Higher Education*, 1(2):197–207, 1976.

[28] M. Kölling and D. Barnes. Enhancing apprentice-based learning of Java. In *ACM SIGCSE Bulletin*, volume 36, pages 286–290. ACM, 2004.

[29] J. Krone and T. Feil. Incorporating mathematics into the first year CS program: a new approach to CS2. *Journal of Computing Sciences in Colleges*, 17(1):44–51, 2001.

[30] K. Lewin. Action research and minority problems. *Journal of social Issues*, 2(4):34–46, Nov. 1946.

[31] S. Markham. Expanding security awareness in introductory computer science courses. In *2009 Information Security Curriculum Development Conference*, pages 27–31. ACM, 2009.

[32] S. Matzko and T. Davis. Teaching CS1 with graphics and C. In *ACM SIGCSE Bulletin*, volume 38, pages 168–172. ACM, 2006.

[33] S. Matzko and T. Davis. Using graphics research to teach freshman computer science. In *ACM SIGGRAPH 2006 Educators program*, page 9. ACM, 2006.

[34] G. A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psycological Review*, 63:81–97, 1956.

[35] J. O'Kelly and J. Gibson. Robocode & problem-based learning: a non-prescriptive approach to teaching programming. In *ACM SIGCSE Bulletin*, volume 38, pages 217–221. ACM, 2006.

[36] F. Paas, A. Renkl, and J. Sweller. Cognitive load theory and instructional design: Recent developments. *Educational Psychologist*, 38(1):1–4, 2003.

[37] N. Parlante, J. Zelenski, K. Schwarz, D. Feinberg, M. Craig, S. Hansen, M. Scott, and D. Malan. Nifty assignments. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, pages 491–492. ACM, 2011.

[38] A. Pears, S. Seidman, L. Malmi, L. Mannila, E. Adams, J. Bennedsen, M. Devlin, and J. Paterson. A survey of literature on the teaching of introductory programming. *ACM SIGCSE Bulletin*, 39(4):204–223, 2007.

[39] J. Sweller, J. J. G. van Merrienboer, and F. G. W. C. Paas. Cognitive architecture and instructional design. *Educational Psychology Review*, 10(3):251–296, 1998.

[40] V. Tirronen and V. Isomöttönen. Making teaching of programming learning-oriented and learner-directed. In *Proceedings of the 11th Koli Calling International Conference on Computing Education Research*, Koli Calling '11, pages 60–65, New York, NY, 2011. ACM.

[41] R. Wicentowski and T. Newhall. Using image processing projects to teach CS1 topics. In *ACM SIGCSE Bulletin*, volume 37, pages 287–291. ACM, 2005.

[42] X. Zhu and H. Simon. Learning mathematics from examples and by doing. *Cognition and instruction*, 4(3):137–166, 1987.