JFP **25**, e23, 16 pages, 2015. © Cambridge University Press 2015
doi:10.1017/S0956796814000021

EDUCATIONAL PEARL

Teaching types with a cognitively effective worked example format

VILLE TIRRONEN and VILLE ISOMÖTTÖNEN
University of Jyväskylä, Department of Mathematical Information Technology
(*e-mail:* ville.e.t.tirronen@jyu.fi; ville.isomottonen@jyu.fi)

Abstract

Teaching functional programming as a second programming paradigm is often difficult as students can have strong preconceptions about programming. When most of these preconceived ideas fail to be confirmed, functional programming may be seen as an unnecessarily difficult topic. A typical topic that causes such difficulties is the language of types employed by many modern functional languages. In this paper, we focus on addressing this difficulty through the use of step-by-step calculations of type expressions. The outcome of the study is an elaboration of a worked example format and a methodical approach for teaching types to beginner functional programmers.

1 Introduction

Student difficulties with type systems are often reported in the literature (see, for example, Joosten *et al.*, 1993; Clack *et al.*, 1995) and these include aspects ranging from difficult syntax to misunderstandings with higher order functions. Similar difficulties have prompted us to reconsider the way types are taught in our courses. In the present paper, we study how types can be presented to students using a calculational approach, and can document results of an experiment that studies the viability of our approach.

Our approach to dealing with the pedagogical difficulties on the topic of type systems is complementary to the more technical studies on how to best present type errors for the programmer. For example, to help a novice programmer, error messages could provide the programmer with reasoning as to why the program fails to compile (Chitil, 2001), or even to describe how the program could be modified so that the compiler accepts it (Lerner *et al.*, 2007). A large effort for improving type error messages is undertaken by Heeren (2005). We find that better error reporting is necessary, and also efficient ways for demonstrating the semantics of types are needed.

On the theoretical level, our work has been inspired by the Cognitive Load Theory (CLT), which is based on the model of human cognitive architechture. Drawing on the limitations of the human working memory, CLT has given rise to documented guidelines that intend to assist the presentation of information in educational contexts (Sweller *et al.*, 1998). The primary of these guidelines is the *worked-example effect*, which states that replacing conventional exercises by study of equivalent examples has a positive impact on learning (Zhu & Simon, 1987; Jelsma & van Merriënboer, 1990). Our specific interest in CLT is to apply the worked-example effect to the design of instructional material concerning type systems. We find that CLT has been previously applied to the field of programming (see, for example, van Merriënboer *et al.*, 1990a, 1990b).

On the practical level, our work can be associated with formative evaluation (Scriven, 1967), where we employ tests and experiments to guide the development of our course. In the empirical part of the present work, we employ a single case research design. Single case research experiments, also known as n-of-1 experiments, have been employed in psychology and behavioural sciences (Nock *et al.*, 2007), and are applicable to situations such as ours, where it is both hard to obtain many experimental subjects and control external factors. In these studies, each subject acts as both experimental control and experimental subject. Such n-of 1 experiments are interpreted *within subject* by drawing comparisons between measures of a single subject in different situations. This experimental setting has been promoted for empirical computer science studies by Harrison (2000).

The context of the present study is a functional programming (FP) course sequence consisting of two master's level courses. The first of these is an introductory course that covers the basics of the Haskell language, while the latter is an advanced course with more varying topics. During three iterations with the first course, we identified several learning difficulties and concluded that students' learning could be facilitated by careful design of exercises and other study materials (Tirronen & Isomöttönen, 2012; Isomöttönen & Tirronen, 2013). The present study is a continuation of this previous work.

2 Challenges in learning functional programming

The role of FP in the computer science curriculum varies from being the main instrument for teaching programming (Joosten *et al.*, 1993; Keravnou, 1995; Thompson & Hill, 1995; Blanco *et al.*, 2009) to being taught as an elective and second programming language (Chakravarty & Keller, 2004). The use of FP as a primary programming paradigm has been seen as beneficial for both student and curriculum as a whole (Joosten *et al.*, 1993; Thompson & Hill, 1995). These benefits are said to include allowing students to better focus on the essential parts of algorithms as well as giving students a more comprehensive view of programming languages in general.

Along with the relatively wide adoption of functional languages as a primary teaching instrument,[1] many reports on teaching experiences gained thereby have been published. These reports present a clear picture of the kinds of difficulties that a novice functional programmer can be expected to encounter. For example, Joosten *et al.* (1993) conclude, after several iterations of the first year FP course, that the syntax of the Miranda language causes difficulties, especially when the two levels of FP, the level of types and the level of values, are mixed. Further, the authors emphasize initial difficulties with the type system and learning higher order functions. Here the issue of learning higher order functions is also linked to the ability to properly define their types. Similarly, Clack *et al.* (1995) documented several difficulties in teaching modern type systems and imparting a proper

[1] The FP language scheme was the fifth most used CS1 language in US colleges in 2012 according to a list by Siegfried *et al.* (2012).

understanding of functions as values. For example, a student might identify the type of a function with the type of its return value. This is a reasonably logical model, since, for example, a function that returns a number is often used in expressions in which numbers are manipulated. This model will nevertheless become non-viable as soon as either partial application or higher order functions are discussed. Further problems with teaching types are examined by Ruehr (2008), who constructs a new graphical language devoted to teaching this topic.

Recursion, which is an even more fundamental concept for FP than for many mainstream languages, can also cause difficulties. For example, Segal (1994) found misconceptions related to identifying the base cases with termination conditions, whereas Bhuiyan *et al.* (1994) argue that the concept of suspend computation and lack of teaching recursion as a problem-solving tool aggravate the difficulties. After identifying and empirically verifying the existence of such misconceptions, Segal (1994) proposes measures to counteract them during teaching, while Bhuiyan *et al.* (1994) develop an entire computer-aided learning environment for teaching the topic.

As reported by Segal (1994), some difficulties seem to be linked with prior exposure to other programming paradigms. Many of the studies reviewed identify specific difficulties that arise during transition from one programming paradigm, nowadays usually that of object-oriented programming, to the paradigm of FP (Joosten *et al.*, 1993; Clack & Myers, 1995). Students with prior experience in imperative or object-oriented programming seem to have more difficulties and exhibit greater resistance in learning FP and the formal tools that are associated with it. The study by Keravnou (1995) confirms these observations and posits that these arise from the confusion between two different computational models.

Part of the difficulties can also be rooted in student perceptions and issues of motivation. Wallingford (2002) proposes that due to lack of proficient teachers and sufficient emphasis on the topic, students only learn the surface features of FP, with the result that the reasons behind various FP abstractions are not understood. In other words, students understand neither *why* the topic is taught nor why *they* should learn it. This resonates with Chakravarty *et al.* (2004), who observed that students' preconceptions against FP, for example the observed lack of practical use of the language being taught, can cause significant resistance to making an effort to learn it.

From the above difficulties, the present paper is concerned with the beginner difficulties with modern type systems, which are unique to modern functional languages. As can be observed in literature, these difficulties are significant, but unlike topics such as recursion, studies that offer to solve problems with learning types seem scarce. In this paper, we focus on applying deductive reasoning for remedying the difficulty of learning the language of the Haskell type system. We approach this problem by adapting the commonly used step-by-step evaluation of expressions to demonstrate type deduction.

3 On Dijkstra's style notation for worked examples

We, like many other FP teachers, promote the use of step-by-step calculations of program fragments as a primary way to present the behaviour of short programs. For presentation of these calculations, we use the notation originating from the notes of Dijkstra (2000). The original purpose of this notation was to communicate a specific train of thought to

(a) Definition of `sum`

```
sum (x:xs) = x + sum xs {- (1) -}
sum []     = 0          {- (2) -}
```

(b) Example calculation with `sum`

```
  sum [1,2,3]
≡ {- (1), with x := 1 and xs := [2,3] -}
  1 + sum [2,3]
≡ {- (1), with x := 2 and xs := [3] -}
  1 + (2 + sum [3])
≡ {- (1), with x := 3 and xs := [] -}
  1 + (2 + (3 + sum []))
≡ {- (2) -}
  1 + (2 + (3 + 0))
≡ 6
```

Fig. 1. Worked example on evaluation of a (recursive) function.

the reader when presenting mathematical proofs, but we find that the presentational issues with communicating proofs and reasoning with programs are similar, and that a similar notation is helpful for both. The Dijkstra's style notation appears in various forms in many educational works, such as Bird (1998) and Broda *et al.* (1994), to name a few.

The widespread use of step-by-step calculations in educational literature makes it evident that it is well suited for manipulating FP constructs. Moreover, using the Dijkstra's style notation makes it natural to progress from simple step-by-step calculations to proof constructions, which are often introduced in FP courses that emphasize formal method concepts. By means of a similar notation, the students can free themselves to concentrate on learning the proof techniques instead of learning how such proofs should be presented.

An example of the Dijkstra's style notation is presented in Figure 1. With this notation each intermediate step is written on a separate line and these intermediate steps are connected with a suitable relation symbol, as well as textual hint, that informs the reader why the step was taken. Such calculations can be used to demonstrate the behaviour of a program or a language construct (see Figures 1 and 2), or can even be used to give rational derivation of a small program (Bird, 1998).

Inspired by CLT, we conjecture that step-by-step calculations offer several benefits in teaching. Firstly, we find that these calculations are good examples of the worked-example effect, which is highly recommended in the CLT literature. Further, many of the step-by-step calculations can be made self-contained by providing appropriate hints between transformations. Self-contained examples are also promoted by CLT, which posits that they lessen the mental load of students in comparison to cases where students have to refer to external resources (see the 'split-attention effect' in Sweller *et al.*, 1998). Similarly, each step in these calculations contains the entire program state and the entire program trace is visible at once. Secondly, we find that step-by-step calculations have been used outside presentation of examples. For example, we have instructed the students to use them as a tool for exploring the behaviour of Haskell expressions. Such an approach has been demonstrated to be beneficial for the students by Fung *et al.* (1996), and, drawing from the CLT principles, we conjecture that it allows students to study difficult program fragments while still leaving cognitive resources for the construction of useful mental models.

Regardless of potential benefits, all program comprehension tools (for an overview, see Sorva 2012) are likely to present *some* extraneous cognitive load for the student. In the

(a) Demonstrating case statements

```
    case 2 of
      1 → "1st"
      2 → "2nd"
      3 → "3rd"
      a → show a++"th"
≡ {- Since 2 ≠ 1, the first case is dropped -}
    case 2 of
      2 → "2nd"
      3 → "3rd"
      a → show a++"th"
≡ {- Since 2 = 2, we picked that case -}
    "2nd"
```

(b) Demonstrating let expressions

```
    let x = 15
        y = x * x
    in x + y
≡ {- replace x's with 15-}
    let y = 15 * 15
    in 15 + y
≡ {- replace y's with 15*15-}
    15 + (15 * 15)
≡ {- Arithmetic -}
    240
```

Fig. 2. Worked examples on the behaviour of let and case expressions.

case of step-by-step calculation, effort must be spent in learning to read the notation, while in the case of program visualization tools, some effort is likely to be spent for understanding the connection between visualization and textual code. The potential difference between these tools is the set of problems for which they offer benefits surpassing the extra cognitive load. For example, when demonstrating behaviour of a simple, declarative program fragment, such as a type expression, we find that a visualization tool cannot tell much more than a simple derivation, but still tasks the student to connect visualization and program code. We speculate that the situation is reversed in cases where a complex data structure, such as a binary tree or a graph, is being manipulated. Here the behaviour of the program contains unnecessary information, which can be effectively removed with visualization tools, but can add extraneous cognitive load when step-by-step calculations are used. In other words, we argue that step-by-step calculations can be a complementary tool to promote programming learning.

3.1 Notation for demonstrating type deduction

We have observed that lack of good syntax for manipulating types makes it difficult for beginners to approach types. For example, the current Haskell syntax offers no clear way to illustrate intermediate steps of type deduction, which leads to difficulties in teaching: Without a way to present the intermediate steps, it is hard to describe why certain expressions have the types that they do. The general assumption that the lack of a proper notation for manipulating types is in part responsible for the difficulties in learning them is also reinforced by our observation that students who are fluent with the types have usually devised some *ad hoc* notation for manipulating them. The need for a notation with which to demonstrate types is also observed by Ruehr (2008), whose solution is to construct an entire graphical language for this task.

In contrast to Ruehr's (2008) graphical notation for types, we attempt to lessen the difficulty of teaching types in our courses with the use of step-by-step calculations. This approach allows us to demonstrate functioning of types in the same way as working with values is commonly demonstrated. That is, we hope to be able to apply the worked-example effect and to divide type-related problems into small steps that allow our students

Table 1. *Semi-formal rules for type deduction*

Example expressions	Reduction
`λa → b` *Type of function is the type of input → type of output*	`«(:t a) → (:t b)»`

`«a → b» «a»`
`«Int → (Int → Bool)» «Int»`
Type of application is the return type of the function if *the parameter type matches the argument type*
	`«b»`
	`«Int → Bool»`

`«a»`
`«(a → b) → [a] → [b]»`
Type variable can be replaced by any *other type* if *all instances of the variable are replaced*
	`«b»`
	`«(Int → b) → [Int] → [b]»`

`42`
`map f «[a]»`
Symbols can be replaced by their type in «»-brackets if *this does not introduce conflicting type variables*
	`«Num a ⇒ a»`
	`«(a'→b) → [a'] → [b]» «a»`

`:t «a → b» «Eq a ⇒ a»` *Constraints must be moved to the left of the equation*	`:t Eq a ⇒ «a → b» «a»`

to explore complex types without overbearing cognitive load. The explicit learning objective considered here is learning to decide types for basic Haskell expressions.

Demonstrating type deductions with step-by-step calculations requires an extension to the Haskell syntax. Our proposed syntactical extension consists of two new notations. The first of these is the operator '`:t x`', which is read as 'the type of `x`'. The operator `:t` is named according to the commands for obtaining the type of an equation in the Haskell interpreter used in our course. The second notational extension is '«x»', which is used to denote an arbitrary value of type 'x'.[2] These syntactic extensions allow us to manipulate type expressions much the same way as we do with value-level expressions.

The rules by which we instructed our students to manipulate the types are given in Table 1. These rules are given in the form of natural language statements and examples as our students are not yet adept in working with formal systems. The rules are intentionally simplified and while these allow deducing the types of many basic expressions, these do

[2] Even though it would be possible to make use of the existing Haskell syntax by writing '«x»' as 'undefined :: x', we find this to be too verbose in practice.

not form a complete set. The focus of these rules is function application and they do not cover pairs, lists and type constructors. These topics can be covered after the basics have been learnt.

Next, we give an example of how these rules can be used to express the calculation of a type. As an example, we use the expression ‘fmap fmap’. Since equations such as these are often complex, we use underline to indicate the modified part of the equation for the reader and hint what after the ≡-sign refers to. The process begins by writing the expression we want to compute:

$$\begin{aligned}
&\quad \mathbf{:t}\ \underline{\text{fmap}}\ \text{fmap}\\
\equiv\ &\textit{\{- type of fmap -\}}\\
&\quad \mathbf{:t}\ \text{«}\mathbf{Functor}\ \text{f} \Rightarrow (\text{a} \to \text{b}) \to (\text{f a} \to \text{f b})\text{»}\ \underline{\text{fmap}}\\
\equiv\ &\textit{\{- type of fmap, notice the renamed variables -\}}\\
&\quad \mathbf{:t}\ \text{«}\mathbf{Functor}\ \text{f} \Rightarrow (\text{a} \to \text{b}) \to (\text{f a} \to \text{f b})\text{»}\\
&\qquad \text{«}\mathbf{Functor}\ \text{g} \Rightarrow (\text{u} \to \text{v}) \to (\text{g u} \to \text{g v})\text{»}\\
\equiv\ &\textit{\{- Float constraints outside of the expressions -\}}\\
&\quad (\mathbf{Functor}\ \text{f}, \mathbf{Functor}\ \text{g}) \Rightarrow\\
&\qquad \mathbf{:t}\ \text{«}(\text{a} \to \text{b}) \to (\text{f a} \to \text{f b})\text{»}\\
&\qquad\ \ \text{«}\ (\text{u} \to \text{v}) \to (\text{g u} \to \text{g v})\text{»}
\end{aligned}$$

As the first step, we replaced ‘fmaps’ with the elements that describe respective types. Since this introduces constraints to the expression, the rules mandate that they are floated out of the expression. To continue with the calculation, we must next unify the types of the parameters with the type of the arguments for the application rule to be applicable. This can be done by substituting a type variable:

$$\begin{aligned}
&\quad (\mathbf{Functor}\ \text{f}, \mathbf{Functor}\ \text{g}) \Rightarrow\\
&\qquad \mathbf{:t}\ \text{«}(\underline{\text{a}} \to \text{b}) \to (\text{f}\ \underline{\text{a}} \to \text{f b})\text{»}\\
&\qquad\ \ \text{«}(\text{u} \to \text{v}) \to (\text{g u} \to \text{g v})\text{»}\\
\equiv\ &\textit{\{- unify a with (u} \to \textit{v) -\}}\\
&\quad (\mathbf{Functor}\ \text{f}, \mathbf{Functor}\ \text{g}) \Rightarrow\\
&\qquad \mathbf{:t}\ \text{«}((\text{u} \to \text{v}) \to \underline{\text{b}}) \to (\text{f}\ (\text{u} \to \text{v}) \to \text{f}\ \underline{\text{b}})\text{»}\\
&\qquad\ \ \text{«}(\text{u} \to \text{v}) \to (\text{g u} \to \text{g v})\text{»}\\
\equiv\ &\textit{\{- unify b with (g u} \to \textit{g v) -\}}\\
&\quad (\mathbf{Functor}\ \text{f}, \mathbf{Functor}\ \text{g}) \Rightarrow\\
&\qquad \mathbf{:t}\ \text{«}\underline{((\text{u} \to \text{v}) \to (\text{g u} \to \text{g v}))} \to (\text{f}\ (\text{u} \to \text{v}) \to \text{f}\ (\text{g u} \to \text{g v}))\text{»}\\
&\qquad\ \ \text{«}\underline{(\text{u} \to \text{v}) \to (\text{g u} \to \text{g v})}\text{»}\\
\equiv\ &\textit{\{- function application eliminates the parameter and the argument -\}}\\
&\quad \mathbf{Functor}\ \text{f}, \mathbf{Functor}\ \text{g} \Rightarrow\\
&\qquad \mathbf{:t}\ \underline{\text{«}}\text{f}\ (\text{u} \to \text{v}) \to \text{f}\ (\text{g u} \to \text{g v})\underline{\text{»}}\\
\equiv\ &\textit{\{- :t eliminates the brackets -\}}\\
&\quad \mathbf{Functor}\ \text{f}, \mathbf{Functor}\ \text{g} \Rightarrow\\
&\qquad \text{f}\ (\text{u} \to \text{v}) \to \text{f}\ (\text{g u} \to \text{g v})
\end{aligned}$$

which completes the example.

Table 2. *Survey results for question 'Did you, during the start of the course identify type signatures such as* `myFunc :: Int → Bool` *with Java/C# style function prototypes such as* `bool myFunc(int:x)`*'*

Yes	No	Yes + explanation
7	9	2

4 The experiment

To study the effect of teaching students to decide types in the proposed step-by-step manner, we performed an experiment during an early phase of the introductory FP course. By this experiment we attempted to study whether worked examples using the proposed notation, combined with the active student use of the deductive system, would have an effect on student performance in connecting value-level expressions to their types. We are also interested in studying student acceptance of the proposed approach.

4.1 Problem background

In our teaching endeavour, we found many student difficulties that were related to types. In our earlier courses, where we had not placed a special emphasis on teaching types, we saw that a large fraction of the student cohort failed to form a usable mental model of the type system in a reasonable time and relied on incorrect models for the whole course. We experienced both syntactic problems, such as not being able to identify the type signatures from value-level expressions, and semantic difficulties, such as difficulties in understanding type variables.

As a particular example of a misconception occurring with types, we found that many students explored the concept of function types through the prototype notation used in the C# and Java languages. Apparently, these students identified Haskell types such as `f:: String → Int` with the Java prototypes such as `int f(x:string)`. Unfortunately, such identification is *not* a viable learning strategy. Consider, for example, learning about partial application by studying the type signatures `f :: Int → Int → Bool` and `f :: Int → (Int → Bool)`. While the first can be identified with `bool f(x:int,y:int)`, the latter has no corresponding notation in Java. In a *post facto* survey conducted on this question (see Table 2), we saw that half of the students had indeed used this pattern, while some commented that they had found it inaccurate later on.

Observations of this kind led us to conclude that we could not communicate the proper mental model of the type system to the students without a more reasoned approach of teaching it. The approach utilizing step-by-step calculations was initially developed during the first iteration of our advanced FP course, where it was designed according to Design-Based Research (DBR) principles by utilizing theory and involving participants. Students who participated in developing this approach generally expressed a positive opinion about the result, but found using it in practice to be taxing. After this tentative trial, we included also the step-by-step worked examples in our introductory course material, which allowed us to conduct a more formal experiment, described in the following section, during the introductory course.

4.2 Experiment setup

The number of students available for the experiment was small. The test subjects needed just the right amount of Haskell experience to be able to understand the material presented but still be sufficiently challenged by types. This limitation ruled out the usual quantitative experiments with control groups and we opted to perform repeated single subject qualitative study over the available student population instead.

Our experimental setup follows the basic three-phase reversal experiment where each individual test subject undergoes three experiment phases. In the experiment the first and last phases provide experimental control for the experiments, and the middle phase introduces the variable being studied. Differing from the basic reversal experiment, where the performance of the subjects is expected to return to the baseline level after the intervention, we observed a learning effect, which should not demonstrate significant reversal after the intervention has been subtracted. In our experiment, we attempted to establish baseline performance by requiring the students to follow Lipovaca's (2012) popular book while completing the exercises in the control phases. In the middle phase, we required the students to use the stepwise deductions and the exact notation for the type deduction presented in this paper. The material presented to the students comprised short explanations of each type level construction used in the experiment, the deductive rules presented in Section 3.1 and five worked examples of type deductions. Before this experiment, worked examples in the Dijkstra's style notation had been introduced to the students in the context of working with value-level expressions.

The test questions that were used in the experiment are given in Figure 3. The questions form two similar sets denoted by letters A and C. The A set of questions was repeated in both first and second phases of the trial. This was done to assess the intervention and the baseline performance with the same measure. Questions A1 to A3 and C1 to C3 were made abstract in an attempt to force students to work without hints such as suggestive names from which the answer could be guessed. Questions A4 and C4 are a part of an exercise given in the introductory course and provide helpful naming to gauge the effect of having less abstract code. Finally, questions A5 and C5 represent tasks that might arise from working with truly abstract code. These questions are not presented in the results as none of the students could make a meaningful headway with these.

From our previous experience we had formed a conclusion that the students' mathematical background and programming avocation were significant factors for performance in the introductory FP course. The background of test subjects was studied with the following questions:

- How well do you remember secondary level mathematics? (not at all – clearly)
- How much have you studied university level mathematics? (not at all – completed the advanced level studies)
- How much do you practise programming? (not at all – almost daily)

After completing the given exercises, we presented the following questions to evaluate the student perception of the method used:

- How fast was the taught method to use? (fast to use – very arduous to use)

(a) Experiment questions

A1)
```
-- Given
f :: [a] → [[a]]
g :: [a] → [a]
-- What is the type of
λx → g (f x)
```

A2)
```
-- What is the type of
λf → (map f) [1,2,3]
```

A3)
```
-- Given
(++) :: [a] → [a] → [a]
"cat" :: [Char]
f = λx → λy → x++y
-- What is the type of
f "cat"
```

A4)
```
-- Given
translate :: Int → Int
          → Picture → Picture
color     :: Color → Picture
          → Picture
circle    :: Int → Picture
-- What is the type of
λg x → translate 10 10
      (color g (circle x))
```

A5)
```
-- Given
f :: (b → c) → (a → b) → a → c
-- What is the type of
f f
```

(b) Control questions

C1)
```
-- Given
f :: (a,b) → b
g :: b → b
-- What is the type of
λx → f (g x)
```

C2)
```
-- Given
succ :: Int → Int
-- What is the type of
λf → succ (f 1)
```

C3)
```
-- Given
f = λx → λy → x
-- What is the type of
f 42
```

C4)
```
-- Given
translate :: Int → Int
          → Picture → Picture
color     :: Color → Picture
          → Picture
circle    :: Int → Picture
-- What is the type of
λg y → translate g 10
       (color red (circle y))
```

C5)
```
-- Given
f :: (b → c) → (a → b) → a → c
-- What is the type of
f f f
```

Fig. 3. Questions about types in the experiment.

- How well did the taught method help you understand types? (it did – it made types more difficult to understand)
- How hard was this exercise? (very easy – very hard), asked after each exercise.

The preceding questions were answered by checking suitable answer from a 5-level scale provided with the question. The scale questions were augmented with a free form question that prompted the students to describe their learning experiences during the test:

- Describe your learning experience.

This experiment was piloted by observing the performance of a CS 1 instructor (later on referred to as subject #1), who had taken the introductory course during the previous year. After the pilot phase, the experiment was taken by seven volunteers from the introductory FP course, who were motivated to participate with a small reward. Three of the participants could be considered beginners in programming in general, while the rest had studied for more than three years and demonstrated a solid programming background. Other than subject #1, no participant had an earlier FP experience.

Our experiment does not offer validation for the general use of the method due to small number of participants and so it should be read as an explorative study. Further threats to

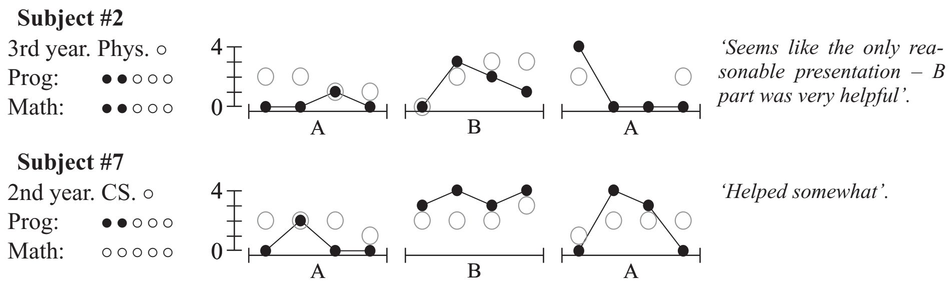

Fig. 4. Performance of beginners.

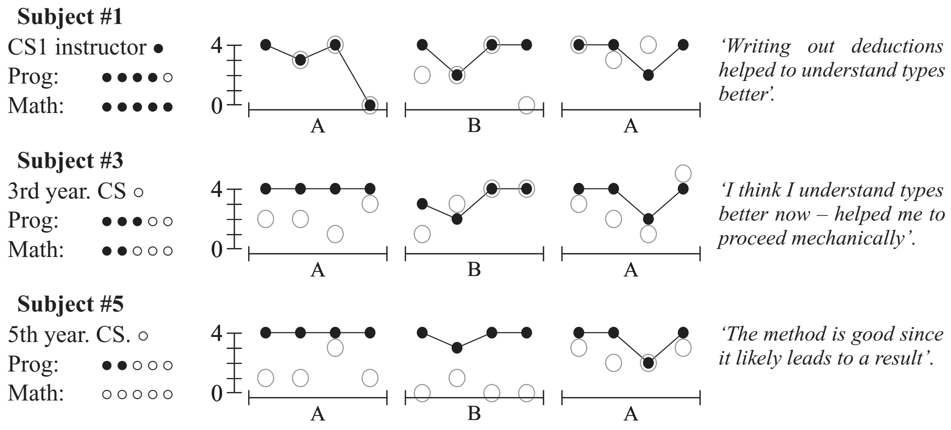

Fig. 5. Performance of advanced students.

generalizability of this experiment arise due to the selecting of volunteers as experiment subjects. Choosing persons with the highest interest towards the topic might have adverse effect on this study as these students are less likely to have difficulties with the topic taught, but can also be more open to trying novel ways of studying. In addition, our decision of using the same exercises for both first and second phases made it more difficult to interpret the results obtained: the mistakes that were made during the first phase of the experiment tended to carry over to the second phase, with the result that some students 'corrected' properly done deductions in the second phase to match the wrong result in the first phase.

4.3 Results

In this section, we use a graphical presentation to give an overview of the student performance (see Figures 4–6). In the figures, the leftmost column shows a short student background and their answers to survey questions about their mathematical background and their familiarity with programming in general. The line graphs in the centre of the figures indicate the level of success for each of the exercises in each of the three experiment phases. The empty circles are used to indicate subject's estimate of the difficulty

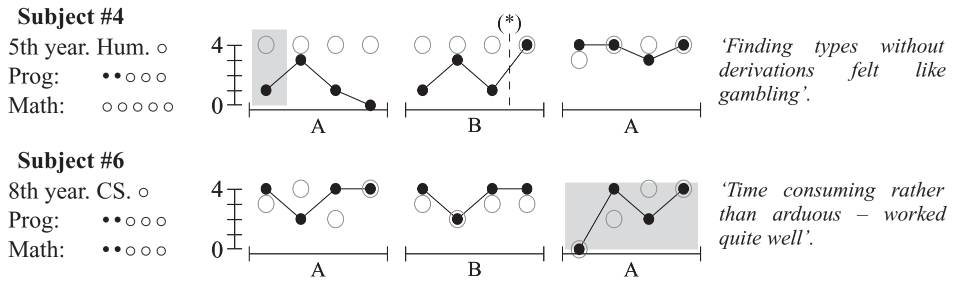

Fig. 6. Performance of other students

of exercises. Missing circles indicate that the students did not estimate the difficulty, and the grey squares indicate where the subject deviated from the study protocol and used the taught method when instructed not to do so. The rightmost column contains a quote of the subject's free form evaluation of the taught method. When observing the figure, note that each plot represents only four different exercises and the visual impact of single mistake can be large.

4.3.1 Effect on beginners

Subjects #2 and #7 can be best described as beginners, and they are arguably the most relevant subjects for this study. Their performance is shown in Figure 4. These students also show the highest measured performance difference between the control and intervention phases, and their comments suggest that they had benefitted from the exercise.

The performance of subject #2 is dismal in both first and last experiment phases. The comments made by subject #2 hint to an unwillingness to consider the problem without a formal tool in the first phase, and this hypothesis is further reinforced by the fact that the subject spent little time on the first portion of exercises: 12 min in the first phase in contrast to 1 h 23 min in the second phase and 27 min in the third phase. Regardless, the subject shows a marked improvement in the middle phase of the experiment, and most of the mistakes done in this phase are result of simple errors in applying the rules and misconceptions about type classes. This same difficulty could be observed, although in lesser scale, in answers of all the subjects, and this most likely resulted from the fact that type classes had not been used in the early part of the introductory course and thus had insufficient examples. As is evident from the positive comment of the student, the mechanical way of deducing types had a positive effect. Unfortunately, in the last phase, and without the given notation, the subject's performance regresses back to the baseline level. We assume that this happened due to the subject following our instructions to the letter, and actively avoiding the use of the presented notation and instead trying to find an alternative.

In contrast to subject #2, the poor performance of subject #7 can be explained through a single misunderstanding: The subject systematically confused application with abstraction and when asked the type of expression `\x->g (f x)` with given types `g::[a]→[a]` and `f::[a]→[[a]]`, the subject answers `([a]→[[a]])→[a]→[a]`. The same pattern is observed several times in the first phase, but it does not appear after the second phase.

Subject #7 performs almost perfectly during the middle phase of the experiment, which is remarkable considering that the subject had spent little more than an hour studying the topic. Unlike subject #2, subject #7 does exhibit a learning effect in the third phase.

4.3.2 Effect on advanced students

As shown in Figure 5, subjects #1, #3 and #5 exhibit almost identical performance throughout the experiments and all of them perform either very well or perfectly in the first phase of the test. In contrast to beginners, the performance of these subjects degrades in the second phase. This loss of performance was expected, since the exercises in the middle phase are strictly more discriminating than those in other phases: these exercises require a correct deduction leading to the correct answer as well as the correct answer. Regardless of the degraded performance, the students comment favourably about the method and indicated that they felt that they had learned something new. The common mistakes that these students made concern the use of type classes.

Although these students do not strictly belong to the target audience of this method, it is possible that they might have benefitted from this exercise as more misconceptions could be brought to light and eliminated. However, though only one of the subjects indicate this in comments, demanding such heavy method of students who are already proficient with types might lead to motivational issues.

4.3.3 Effect on other students

As shown on Figure 6, subjects #4 and #6 exhibit more variability in their performance as well as their background than other students. These subjects also confused the experimental protocol so that subject #4 used the formal deduction in the first question of the first phase and subject #6 did the same for all questions in the last phase.

The initially weak performance of student #4 could be explained by two persistent misconceptions: refusal to accept the concept of curried functions, and difficulties in understanding scoping of type variables. The exercises consistently used groups of type definitions that shared the same variable names which the subject took to mean the same variable. After failing to conclude proper types for the majority of the exercises in the first two phases, the subject was instructed to fix the latter misconception, which shows a high improvement in performance in the last experiment phase (this intervention is marked with an asterisk in Figure 6). In the free form comments, this subject also indicated that the taught method was found to be helpful and clarifying.

Variability in the performance of subject #6 can also be observed to originate from a single root cause, which is again the improper understanding of type classes. The curious failure in the first question of the last phase seems to be an honest mistake, which the subject probably could have fixed if given another try. All in all, subjects #4 and #6 were positive in their comments, and their comments indicated open mindedness about new ways of teaching this topic.

5 Discussion

In this paper, we have described an approach to teaching types by using an algebraic, step-by-step approach. The proposed approach is inspired by CLT and has been developed with the participation of student cohort. The approach has been also studied experimentally and the results are reported in the present paper.

Our approach to teaching types was reported to be a positive experience by the students who participated in the experiment. Students felt that they had spent their time well and that they had learned much about types. From the actual answers, we can conjecture that the students with inferior initial performance benefitted most by using our approach, while those with adequate initial performance gained relatively little. From the teacher's point of view, we find that requiring stepwise deductions for the exercises can make it easier to discover and correct student misconceptions. We also believe that our approach to teaching types can make it easier to develop exercises and teaching materials that make the use of CLT effective.

As for the challenges in using our approach, we have observed that teaching formal deduction is costly, and cannot be done solely for the benefit of teaching types. Without introducing the concept of step-by-step deductions earlier in the course, it is likely that requiring it with types will require too much extra effort for it to be beneficial. Also, in our case, all of the test subjects reported that they would have liked to see more examples related to our approach and would have further benefitted from a more clear presentation of the material.

Our findings suggest that the use of this approach requires exact and well-designed course materials to minimize the effort of learning to use the deductive system. It appears necessary to introduce step-by-step deductions early during FP courses to make full use of the proposed approach. However, as is evident from many introductory books about FP, the step-by-step deductions are a good tool for teaching, and introducing these early in course materials should not be problematic. As a final note, the mechanical nature of this approach hints that learning this process could be benefitted from computer-aided learning environment in the same way as programming itself is benefitted from development environments that help to eliminate simple mistakes.

References

Bhuiyan, S., Greer, J. E. & McCalla, G. I. (1994) Supporting the learning of recursive problem solving. *Interact. Learn. Environ.* **4**(2), 115–139.

Bird, R. (1998) *Introduction to Functional Programming Using Haskell*, 2nd ed. Upper Saddle River, NJ: Prentice Hall.

Blanco, J., Losano, L., Aguirre, N., Novaira, M. M., Permigiani, S. & Scilingo, G. (2009) An introductory course on programming based on formal specification and program calculation. *ACM SIGCSE Bull.* **41**(2), 31–37.

Broda, K., Khoshnevisan, H. & Eisenbach, S. (1994) *Reasoned Programming*. Upper Saddle River, NJ: Prentice Hall.

Chakravarty, M. M. T. & Keller, G. (2004) The risks and benefits of teaching purely functional programming in first year. *J. Funct. Program.* **14**(1), 113–123.

Chitil, O. (2001) Compositional explanation of types and algorithmic debugging of type errors. *ACM Sigplan Not.* **36**(10), 193–204 (ACM).

Clack, C. & Myers, C. (1995) The dys-functional student. In *Funtional Programming Languages in Education*, Hartel, P. & Plasmeijer, R. (eds), Lecture Notes in Computer Science, vol. 1022. Berlin, Germany: Springer, pp. 289–309.

Dijkstra, E. W. (2000, July) *The Notational Conventions I Adopted, and Why*. Available at: http://www.cs.utexas.edu/users/EWD/ewd13xx/EWD1300.PDF. Accessed December 8, 2013

Fung, P., O'Shea, T., Goldson, D., Reeves, S. & Bornat, R. (1996) Computer tools to teach formal reasoning. *Comput. Educ.* **27**(1), 59–69.

Harrison, W. (2000) N=1, an alternative for software engineering research? In *Beg, Borrow, or Steal: Using Multidisciplinary Approaches in Empirical Software Engineering Research, Workshop Report*, Limerick, Ireland, vol. 5. Citeseer, pp. 39–44.

Heeren, B. (2005) *Top Quality Type Error Messages*. PhD thesis, Universiteit Utrecht, The Netherlands;IPA Dissertation Series.

Isomöttönen, V. & Tirronen, V. (2013) Teaching programming by emphasizing self-direction: How did students react to active role required of them? *ACM Trans. Comput. Educ. Res* **13**(2), article No. 6.

Jelsma, O. & van Merriënboer, J. J. G. (1990) The adapt design model: Towards instructional control of transfer. *Instr. Sci.* **19**(2), 89–120.

Joosten, S., Berg, K. & Hoeven, G. V. D. (1993) Teaching functional programming to first-year students. *J. Funct. Program.* **3**(1), 49–65.

Keravnou, E. (1995) *Introducing Computer Science Undergraduates to Principles of Programming Through A Functional Language.* Functional Programming Languages in Education. LNCS, vol. 1022. Berlin, Germany: Springer; pp. 15–34. see http://www.springer.com/computer/swe/book/978-3-540-60675-8

Lerner, B. S., Flower, M., Grossman, D. & Chambers, C. (2007) Searching for type-error messages. *ACM SIGPLAN Not.*, **42**, 425–434. (ACM).

Lipovaca, M. (2012) *Learn You a Haskell for Great Good!: A Beginner's Guide*. San Francisco, CA: No Starch Press.

Nock, M., Michel, B. & Photos, V. (2007) Single-case research designs. In *Handbook of Research Methods in Abnormal and Clinical Psychology*, Mackay, D. (ed.). Thousand Oaks, CA: Sage, pp. 337–350.

Ruehr, F. (2008) Tips on teaching types and functions. In *Proceedings of the 2008 International Workshop on Functional and Declarative Programming in Education*. New York, NY: ACM, pp. 79–90.

Scriven, M. (1967) The methodology of evaluation. In *Perspectives of Curriculum Evaluation, Aera Monograph Series on Curriculum Evaluation*, Tyler, R., Gagné, R. & Scriven, M. (eds), vol. 1. Chicago, IL: Rand McNally, pp. 39–83.

Segal, J. (1994) Empirical studies of functional programming learners evaluating recursive functions. *Instr. Sci.* **22**(5), 385–411.

Siegfried, R. M., Greco, D., Miceli, N. & Siegfried, J. (2012) Whatever happened to Richard Reid's list of first programming languages? *Inf. Syst. Educ. J.* **10**(4), 24.

Sorva, J. (2012) *Visual Program Simulation in Introductory Programming Education*. Esbo, Finland: Aalto University.

Sweller, J., van Merrienboer, J. J. G. & Paas, F. G. W. C. (1998) Cognitive architecture and instructional design. *Educ. Psychol. Rev.* **10**(3), 251–296.

Thompson, S. & Hill, S. (1995) *Functional Programming Through the Curriculum.* Functional Programming Languages in Education. LNCS, vol. 1022. Berlin, pp. 85–102. Germany: Springer; see http://www.springer.com/computer/swe/book/978-3-540-60675-8

Tirronen, V. & Isomöttönen, V. (2012) On the design of effective learning materials for supporting self-directed learning of programming. In *Proceedings of the 12th Koli Calling International*

Conference on Computing Education Research (Koli Calling'12). New York, NY: ACM, pp. 74–82.

Van Merriënboer, J. J. G. (1990a) Strategies for programming instruction in high school: Program completion vs. program generation. *J. Educ. Comput. Res.* **6**(3), 265–285.

Van Merriënboer, J. J. G., & Paas, F. G. W. C. (1990b) Automation and schema acquisition in learning elementary computer programming: Implications for the design of practice. *Comput. Hum. Behav.* **6**(3), 273–289.

Wallingford, E. (2002) Functional programming patterns and their role in instruction. In *Proceedings of the International Conference on Functional Programming*, Pittsburgh, PA. New York, NY: ACM, 151–160.

Zhu, X. & Simon, H. A. (1987) Learning mathematics from examples and by doing. *Cogn. Instr.* **4**(3), 137–166.