# Visual Support for Learning Monads

Tim Steenvoorden Jurriën Stutterheim Erik Barendsen Rinus Plasmeijer

Institute for Computing and Information Sciences
Radboud University Nijmegen, P.O. Box 9010, 6500 GL Nijmegen, The Netherlands
{t.steenvoorden, j.stutterheim, e.barendsen, r.plasmeijer}@cs.ru.nl

Monads are an important topic in functional programming. In Haskell, for example, monadic I/O is the only way to perform I/O at all. However, newcomers to functional programming, such as bachelor students, often struggle with learning about monads. In other domains, graphical formalisms such as Venn diagrams or commutative diagrams are often used to support students with a visual learning preference in learning new concepts. Previously, we have developed a novel tool, called Tonic (Task-Oriented Notation Inferred from Code) that generates a graphical representation of the monadic structure of Clean programs, akin to flow diagrams. Tonic is integrated in the Clean compiler, allowing us to *automatically* generate these blueprints from code. In this paper we describe and evaluate how we have used blueprints to help a group of second year bachelor's students learn about monads. We have found that using blueprints in the lecture slides and in the assignments have a positive impact. Visually oriented learners tend to appreciate blueprints, but tend to look at given blueprints rather than constructing them themselves. Exam marks are on the same level or better than previous years, indicating students' performance is not negatively affected. We conclude that Tonic should be developed further, such that students can generate blueprints during code development.

## 1 Introduction

The works of Moggi [11], Wadler [24], and Peyton-Jones [14] in the late 1980's and early 1990's establish monads as a powerful abstraction to maintain state, model failing or non-deterministic computations, perform side-effecting operations, etcetera. Monadic I/O is even the principal, and indeed the only, way to perform I/O in Haskell [7]. Still, today, over 25 years after the introduction of the concept of monads to the world of functional programming, beginning functional programmers struggle to grasp the concept of monads. This struggle is exemplified by the numerous blog posts about the effort of trying to learn about monads. From our own experience we notice that even at university level, bachelor level students often struggle to comprehend monads and consistently score poorly on monad-related exam questions.

Considering that the concept of monads is not likely to disappear from the functional programming landscape any time soon, it is vital that we, as the functional programming community, somehow overcome the problems novices encounter when first studying monads. To do so, we have find a way to support students when they run into monads for the first time. This study aims to answer the question: *"Can visualizations of monadic programs support students learning the concept of monad, and if so, in what way?"*

### 1.1 Visualizing Monads

Previously, we have presented a novel tool called Tonic: Task-Oriented Notation Inferred from Code [23, 22]. Tonic automatically generates a visual representation, called *blueprints*, of a program's code in a manner akin to flow-diagrams and BPMN[1] [12]. These blueprints give a high-level graphical represen-

[1] BPMN: Business Process Model and Notation, a graphical representation for specifying business processes.

tation of a program on the abstraction level of monads. Figure 1 and Figure 2 in Section 2 both show an example of a blueprint. Tonic's original goal was to partially bridge the communication gap that exists between programmers and non-technical project stakeholders by giving non-technical people insight into the design of a program. Later, we saw that programmers may benefit from Tonic as well, by using it as a tracer or debugger.

Whether Tonic achieves these goals has not been formally studied yet, but we suspect that it may aid in teaching and learning about monads as well. Inexperienced students can initially be regarded as non-technical stakeholders, matching Tonic's target audience profile. Students studying other subjects, such as object-oriented programming, have traditionally benefited from the use of graphical languages like UML[2] [20]. Tonic may be able to fulfill a similar role for students studying monads. This study will investigate whether this is indeed the case. If so, this will support future efforts in developing Tonic into a stable and user-friendly product that is fit for the classroom.

Monads are a sweet-spot in the functional design space. They abstract from many bookkeeping details, such as passing around state, while at the same time being concrete enough to recognize individual steps in a program. Additionally, monads force a particular order of evaluation, something that is lost in lazy functional programming. This makes it possible for Tonic to track the progress of a program while it is being executed. It can highlight the part of a blueprint that is currently being executed in a *dynamic blueprint*. In a Tonic version specialized to the iTask system [15], Tonic can also inspect the values being passed around in a program, essentially turning it into a tracer or debugger. Tracers and debuggers are known to be hard to implement for lazy functional programming languages.

## 1.2 Aspects of Learning Monads

We suspect the difficulty in comprehending monads has several causes. First and foremost, monads are a technically advanced concept. The way monads are commonly implemented, using type classes, requires a student to have mastered several functional programming concepts first: higher-kinded types, higher-order functions, and type classes. To prove the monad laws for any given type, one needs to have mastered equational reasoning and proof by induction as well. Teachers and students cannot relate monads to *basic* mathematical concepts as they can do with functions or expressions.

Another challenge may stem from a lack of support for students with a visual learning preference. Many formalisms have some form of visual support to express ideas. For example, set theory has Venn diagrams, graph theory has visual graphs, category theory has commutative diagrams, and object-oriented programming has UML. No such ubiquitous visualization language exists for functional programming, however. Attempts at visual representations for functional programming have been made in the past [3, 16, 18], but none of these visual languages have become as ubiquitous as the aforementioned formalisms. In addition, the impact of visual representations of general-purpose functional programming languages on learning has, to our knowledge, never been formally studied.

In studying the way students learn, the concept of a *notional machine* is often employed. A notional machine is an abstraction of the computer that one can use for thinking about what a computer can and will do. The construction of a mental model for a notional machine is considered crucial to learners in programming [2]. Often hidden or unmarked actions in program code, like inheritance or instantiation in Object Orientation [6], cause problems for novices. A notional machine helps students to relate program actions to events in the model [21]. Monadic code gets a concrete meaning at the time it is applied on a concrete monad instance. Due to the high abstraction of monadic code, the construction of a notional

[2]UML: Unified Modeling Language, a general-purpose modeling language for object-oriented programming.

machine is even more important for this aspect of functional programing languages. The ability to trace code by hand to build such a model can help students immensely [13].

In this paper we aim to make a first step toward using and testing Tonic as visual aid during lecturing which can help students write and understand monadic programs. We study the impact of these visualizations in lecture material on monads, which is part of an eight week course taught at Radboud University in Nijmegen, the Netherlands. Students reception and perception are investigated using a multi-method approach. By analyzing quantitative and qualitative student data, we aim to get a better understanding of students' perception of blueprints and the way they think about monads. This way we can expand our learning tools and didactic methods concerning monads in particular and functional programming in general.

The remainder of this article is structured as follows. In Section 2 we take a look at two monadic functions and their visual representation. Next, Section 3 gives a high level overview of the implementation of Tonic and technical challenges met during implementation. Section 4 elaborates on the context of the in depth case study and methods used for data collection and analysis. Results and analysis of the case study can be found in Section 5. Section 6 discusses the results of our study and presents our conclusions. Next, Section 7 discusses future work, after which Section 8 wraps up with a discussion of related work.

## 2 Examples

In this section we illustrate the capabilities of Tonic based on two examples. The first example uses the `IO` monad. We show how a function which concatenates the contents of two files is visualized by Tonic. The second example is about random number generation. A recursive function generates a list of random integers inside a `State` monad. The same functions were used during the lecture as examples (see Section 4.2).

### 2.1 Concatenating the Contents of Two Files

As an example we study a function without parameters in the `IO` monad. The function `concatFiles` below is written in the pure, lazy functional programing language Clean. It reads the contents of two files, `"in1"` and `"in2"`, concatenates their contents, and writes the result out to a third file named `"out"`.

```
concatFiles :: IO ()
concatFiles =      readLines  "in1"
    >>= \lines1 -> readLines  "in2"
    >>= \lines2 -> writeLines "out" (lines1 ++ lines2)
    >>= \ok     -> if (not ok) (abort "Something bad happened")
                               (pure ())
```

Tonic automatically generates a so-called *static blueprint* of this function, which is shown in Figure 1.

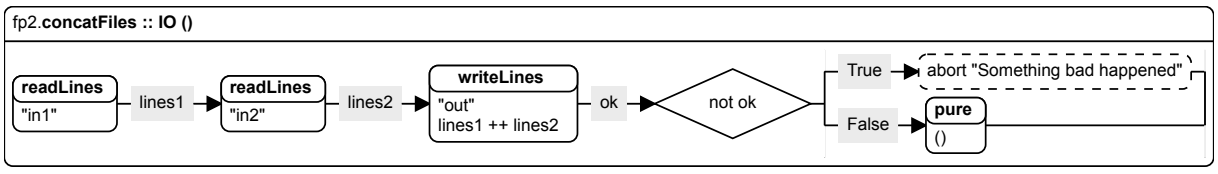



Figure 1: Blueprint of the `concatFiles` function, automatically generated by Tonic.

Visualizations generated by Tonic contain multiple aspects. The entire function is captured in an enclosing box with a title. The title in Figure 1 tells us that the blueprint is a visualization of the function concatFiles defined in module fp2 with type IO (). Inside the box, Tonic draws rounded rectangles for each application of monadic functions. The rounded rectangle for writeLines shows the name of the function in bold and each argument of the call on its own line: the string "out" and the list lines1 ++ lines2. The arrows represent the monadic *bind* operation and show the direction of control flow and data flow. Each arrow is annotated with the name of the parameter of the bound function (lines1, lines2 and ok) or the pattern to be matched after a case distinction (True and False). Variables bound by arrows can be used in the remaining part – the part to the right of the binding site – of the diagram. Case distinctions or conditionals are depicted with diamonds.

### 2.2 List of Random `Int`s

Blueprints get more interesting when taking typical functional programming features into account. The function `randomN` below uses recursion to create a list of `n` random integers inside a state monad.

```
randomN :: Int -> State Seed [Int]
randomN n = if (n == 0)
    (pure [])
    (           random
     >>= \x  -> randomN (n - 1)
     >>= \xs -> pure [x : xs] )
```

Figure 2 shows the generated blueprint by Tonic. It starts with a diamond to signify the conditional check on whether n == 0. The True branch results in an empty list inside the state monad. The False branch creates a random integer and recursively calls into randomN to generate the tail of the list. In addition to recursion, Tonic can handle higher-order functions and other functional concepts in a transparant way.

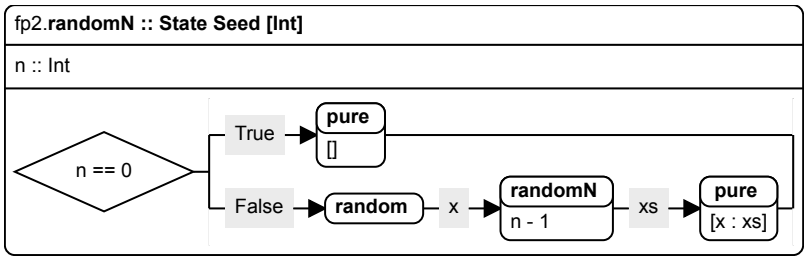



Figure 2: Blueprint of the recursive function `randomN` as generated by Tonic.

Both examples show how abstract, functional code is transformed into pictures. By following arrows from box to box, students can track the execution order of the monadic code. This is exactly the idea behind a notional machine, introduced in Section 1.2. By displaying blueprints on paper or on screen, students can trace program flow *by hand*. Because blueprints are generated from code, they can give a clue about what the original code could look like.

## 3 Technical Background

Tonic is a key component in this experiment. Tonic can generate its blueprints, because Tonic is implemented as part of the Clean compiler. The compiler generates static blueprints immediately after it

is done with its type-checking phase. At this point, the compiler can distinguish monadic code from non-monadic code. It then constructs a blueprint AST, which it subsequently writes to disk. Afterwards, a Tonic viewer program can read the blueprint from disc and render it. Tonic is able to visualize the structure of any monadic program, but it is up to the programmer to decide which monadic are visualized in a blueprint and which are not. The programmer does so by instantiating Tonic type classes for the monads that are to be visualized.

Tonic's ability to create dynamic blueprints extends beyond allowing the user to view the progress of the program while it is being executed. In the case of the iTasks-specific Tonic implementation, we can also inspect values at run-time, much like in a debugger. In iTasks, programs are constructed by composing tasks of the monadic type `(Task a)`. iTasks features so-called *editor* tasks, which use generic programming techniques to automatically generate an interactive graphical user interface for any first type for which we have derived an instance of a generic `iTask` class. We can use these generic editors to inspect task values at run-time in a Tonic viewer. Inside the `Task` monad, iTasks maintains an identification system distinguishing each running task. Tonic uses this task identifier to relate the program's progress to the blueprints and highlight the corresponding parts of the blueprint. In the iTasks-specific Tonic viewer, we treat the iTask system as a white box and can therefore can use all capabilities and internal administration of the iTasks system.

When implementing support for dynamic blueprints for arbitrary monads, we face some issues. In the Tonic for iTasks implementation, we make use of iTasks graphical user interface libraries to create a user interface for Tonic and to display blueprints. An arbitrary monadic program may only do I/O on a command line. In such a program, we cannot make use of the program's graphical user interface to display blueprints, since it does not have a graphical user interface. To support viewing blueprints for arbitrary monadic programs, we must construct a stand-alone Tonic viewer with its own graphical user interface. In this stand-alone viewer, we cannot make use of iTasks' editors to inspect the program's values at run-time either, because we cannot be sure that the types of the values we want to inspect have an `iTask` class instance. Supporting runtime value inspection would probably require us to integrate Tonic with the Clean run-time system as well as the Clean compiler. Work on the stand-alone Tonic viewer is underway, but still very preliminary. For use in this study, the stand-alone the viewer would need to be very stable, because it should not obstruct the students in their efforts to do their practical assignments. Since the viewer is not stable enough yet, we did not include it in this study and only generated static blueprints beforehand.

# 4 Case Study

Our case study uses multiple data sources. The usage of both quantitative data (questionnaire, final exam) and qualitative data (observations, interviews, students' notes) allows us to make an in-depth analysis of visualized monad education and students' perception. In this section we discuss the context in which the case study takes place, the events leading to the collected data and the methods used to analyze this data.

## 4.1 Context

Our participants were a group of university bachelor students from Radboud University in Nijmegen, the Netherlands, taking the course Functional Programming 2 (FP2). The FP2 course is mandatory for all Computer Science majors, except for those in the track Cyber Security. The course is also an elective for students with other majors, like Artificial Intelligence. All students are typically in the second year

of their education and have taken the mandatory Functional Programming 1 course (FP1). A total of 39 students participated in this year's FP2 course. Of these students, 6 took the course for the second time.

The basics of functional programming are taught in FP1. In FP1, students learn concepts such as higher-order functions, polymorphism, type classes, (recursive) algebraic data structures and program correctness. During the FP2 course, students learn about uniqueness typing, monads and dynamic typing. Our case study was integrated in the second course week of FP2, which is about monads. All data was collected during this week.

## 4.2 Data Collection

We performed data collection in three phases. In the first phase we prepared the lecture and the assignments. After that, in the second phase, students were closely monitored while they are working on their assignments. Finally, in the third phase, the students filled in questionnaires, were interviewed, and took part in the final exam. Both the slides[3] and the assignment are available on the Internet[4]. In the next subsections we describe our data collection in chronological order.

### Preparation

The week of the case study started with a lecture on Monday. In this lecture, students learn about monadic programming: cases where monadic code can be applied, ways to implement the monad type class, and proving the monad laws for simple monad instances. The lecture slides were augmented with Tonic blueprints. This way, all students got the opportunity to relate monadic code to its visual counterpart.

At the end of the lecture, we asked for volunteers to participate in this study. Six groups of two and one group of three people stepped forward. The composition of the groups was as follows. Three groups consisted of one male and one female participant, three groups were male only and one group was female only. The proficiency in functional programming varied between the groups. Some participants have performed well during the FP1 course, while others have not. All groups are informed that they and their screens would be recorded during the practical session in order to observe how they deal with the problems presented there. They were also informed that they would be interviewed about the practical sessions afterwards.

On Tuesday, a pair of students was invited on a pilot in office. They received an exercise set to put gathered knowledge from the lecture into practice. These exercises were also augmented with Tonic blueprints. Feedback received after this session, and questions asked by the students lead to small modifications in the exercise set to maximize readability and quality of the instructions. Specifically, we marked three questions on the verification of the monad laws as bonus exercises and distributed a skeleton source file with Clean code to eliminate the need for students to copy the code samples to their text editors manually.

One of the exercises from the practical assignment is depicted in Figure 3. Students had to write the monad instance for the `Either` data type. Next, they were asked to reason about a piece of code and rewrite it. The blueprint in the figure is a visual representation of the `parsePerson` function, which is is given in a non-monadic way. Students had to rewrite it using monadic operators. In the blueprint we can see that the parameter `age` is not directly used in the next bind, but only at the end, during the creation of a record of type `Person`.

[3] `http://cs.ru.nl/~steenvoo/research/documents/20160418-fp2-monad-lecture-handout.pdf`
[4] `http://cs.ru.nl/~steenvoo/research/documents/20160420-fp2-monad-assignment.pdf`

**The Either monad**

The `Either` type models a computation that might fail with an error message:

```
:: Either e a = Left e | Right a
```

By convention `Right` is used as a "right" answer and `Left` is used to signal a "not right" answer, or an error.

1. Implement the `Monad` instance for `Either`.

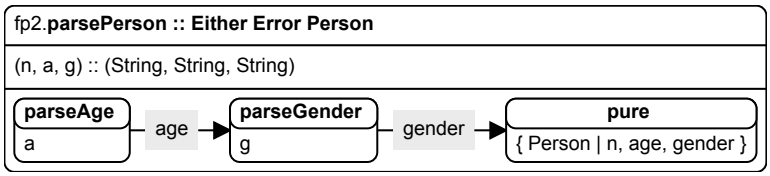
Lets take a look at a program which parses CSV formatted data. Each row contains person details: a person's name, his or her age and gender. The rows and cells are already split in lists of tuples of strings.

```
:: Person  = {name :: String, age :: Int, gender :: Gender}
:: Gender  = Male | Female
:: Error :== String

parseAge :: String -> Either Error Int
parseAge ""              = Left "Nothing to parse as age"
parseAge str
  # x = toInt str
  | x == 0 && str <> "0" = Left (str +++ " is not a valid age")
  | otherwise            = Right x

parseGender :: String -> Either Error Gender
parseGender "m" = Right Male
parseGender "f" = Right Female
parseGender  _  = Left "Unknown gender"

parsePerson :: (String, String, String) -> Either Error Person
parsePerson (n, a, g) =
  case parseAge a of
    Left e    -> Left e
    Right age ->
      case parseGender g of
        Left e       -> Left e
        Right gender -> Right {name = n, age = age, gender = gender}
```




2. Describe in your own words how the following program is evaluated and what its result is.

```
Start = map parsePerson
          [ ("Alice", "37",  "f")
          , ("Bob",   "2.5", "m")
          , ("Carol", "18",  "o")
          , ("Dave",  "",    "f") ]
```

3. Refactor the `parsePerson` function using the monadic `(>>=)` and `pure` functions. Use lambdas and helper functions however you see fit.

Figure 3: Excerpt from the assignments on monads handed out to students. The exercise incorporates the instance of the `Either` monad and the monadic equivalent of a case distinction waterfall.

### Observation

Two days after the lecture, on Wednesday, all remaining students took part in a practical session. During this session, they received a printed version of the improved exercise set. The assignments were handed out at the beginning of the practical session, so students were not able to prepare beforehand. The students were free to choose between Clean and Haskell as implementation language for the exercises.

During this four hour session, the five groups of two people and one group of three people were closely monitored. All other students worked in pairs as well, but are not observed. Working in pairs or triples forces the students to verbalize their thoughts and ideas. The student groups participating in this research were explicitly encouraged to communicate their thoughts verbally. A webcam recorded their conversation in both video and audio. Their screen was recorded using screen capturing software. Afterwards, all media was combined in one synchronized file. Empty sheets of paper were handed out for the students to take notes. The notes of the study's participants, as well as their exercise printouts, were collected at the end of the practical session, so we could see what the students had written down or had drawn.

All students got one week to complete the exercises. After one week, they had to hand in their answers. Student assistants graded the results and provided them with feedback, which the students received within one week of handing in the assignments.

### Evaluation

In the days after the practical session, Thursday until Tuesday, the seven groups of monitored students were invited for a semi-structured interview. Interviews were set up in a stimulated recall [9] fashion: students were confronted with key parts of their acting which helps them recall the situation and evoke reactions on their actions. To select appropriate video fragments, the interviewers skimmed the recordings beforehand to spot interesting events. Interview themes were set up in advance and incorporate topics on conceptual understanding, self-efficacy and usability of Tonic blueprints.

Students monitored during the practical session were invited to fill out the questionnaire just before the interview. This same questionnaire was sent to all participating students on Thursday, the day after the practical session. The questionnaire was about usability and usage of blueprints during the lecture and the practical session. Also, it asked some questions about perception on monads. The order in which the questions were presented is randomized. All questions could be ranked on a five point Likert scale, ranging from *totally disagree* up to *totally agree*. Totally disagree was rewarded with the lowest score of 1, while the highest score of 5 corresponded to totally agree.

All students taking part in the final exam got four exercises on monads. The first exercise tested pure knowledge on the type signature of the monadic operators The other three asked students to create a monad instance and apply monad operators to a piece of code. These questions too were augmented with a Tonic blueprint to closely resemble the learning environment of the lecture and practical session.

To more easily compare multiple exams we classified all exam tasks into three categories. We base these categories on three levels of cognitive process from Bloom's revisited taxonomy [8]: remember, apply and create. *Remembering* is about retrieving relevant knowledge from long-term memory. This is for example the case when students are asked to give the type signatures of the monadic operators. Asking them to use the monadic operators to build a function is an example of *applying* knowledge. The highest level of cognitive process is *creating*, which is what students have to do when they have to construct a monadic instance for a data type not seen before. We choose these three levels in such a way, that the current and the past exam tasks can easily be classified.

### 4.3 Data Analysis

More than a week of data collecting resulted in: (1) audio and video recordings, including screencasts, from the practical session; (2) hand written notes from the practical session; (3) audio recordings from each interview; (4) answers on the questionnaire; and (5) marks from the final exam.

The data was analyzed in three phases. The first phase occurred between the practical session and the interviews. Material from the practical session was used to prepare the interviews, as discussed above in Section 4.2. Recordings and screencasts were scanned to extract potential points of interest. These could be errors or problems the students encountered during programming, but also misconceptions that led to wrong problem solving, or decisions to follow a path not intended by the teachers. We were particularly interested in the way the students used the blueprints. The points of interest were used to confront students during their interview. The interview aids in better understanding of students' behavior and complements the results of the questionnaire.

After the course week, the second phase took place. Here we analyzed the answers to the questionnaire and the interview. Quantitative analysis of the questionnaire resulted in medians ($Md$, the middle value of a data sample), interquartile ranges ($IQR$, a measure for dispersion), and modes ($Mo$, most common value in a sample). The lower the interquartile range, the lower the variability in the data. The results of the questionnaire show students' beliefs on the usage of blueprints and their perception on monads. From our own experience, response rates on questionnaires spread amongst students are usually low. Our research design anticipates on a low response rate by using the interview analysis as supportive material. Qualitative analysis of the interview data allows for a better interpretation of the questionnaire results providing concrete examples of students beliefs. Therefore, in this phase we combined our findings from the questionnaire with those of the interview to contribute to our findings.

In a final third analyzing phase, we compared the results of the exam from this year to the exams of previous years. Every year the student population is different and exam questions are changed. Also, teaching methods change slightly over years. This is even more true this year. Therefore we like to stress we can not make an comprehensive comparison based on exam results. One thing we might be able to say, is if the introduction of blueprints during the lectures has no *negative* effect on exam results.

## 5 Results

In this section we present the results from the questionnaire, the final exam and the interviews. The interpretation of this data is discussed in several subsections, together with the analysis of the interviews.

Table 1 lists the questions asked to all students participating in the course. The table shows medians and interquartile ranges for each question. As stated before, questions could be scored from 1 (*totally disagree*) up to 5 (*totally agree*). 21 of the 39 students students filled in the questionnaire, which results in a response rate of 54%. A visual representation of the same data can be found in Figure 4 as a box-and-whisker plot, including the mode.

As introduced in Section 4.3, we use the revisited taxonomy of Bloom [8] to categorize exam questions on three knowledge levels: remember, apply and create. Table 2 lists the fractions of each exam that was about a particular knowledge level. All figures *only* represent questions about monads. In this year for example, 8% of the total exam was about *creating* a monad instance for a selected data type. The resit of 2015 contained a question about applying monadic operators, which was good for 21% of the total points students could earn. As only the exams of 2016, 2015, 2013, and the resit of 2015 contained one or more questions on monads, we just list these four exams. The table demonstrates that each exam has a slightly different emphasis every year.

| | Question | Median | IQR |
|---|---|---|---|
| **A** | The pictures helped me to understand the examples in the slides better. | 4.0 | 0.00 |
| **B** | The pictures helped me to understand the exercises better. | 4.0 | 1.75 |
| **C** | The pictures are easy to understand. | 4.0 | 0.75 |
| **D** | The pictures do not get in the way of understanding what a function does. | 4.0 | 1.00 |
| **E** | The pictures do not add any value. | 2.0 | 0.75 |
| **F** | When I got stuck on an exercise, I would look at the pictures again. | 3.5 | 2.00 |
| **G** | When I would do an exercise, I would draw my own picture of a program on paper. | 2.0 | 1.00 |
| **H** | The concept of monads is easy to understand. | 2.5 | 1.00 |
| **I** | Monads confuse me. | 3.0 | 1.00 |

Table 1: Results from the questionnaire on monads and blueprints based on the answers of 21 students (54% of participating students). Questions are numbered *A* to *I*. For each of the nine questions the table shows the median and the interquartile range. Further information can be found in Figure 4.

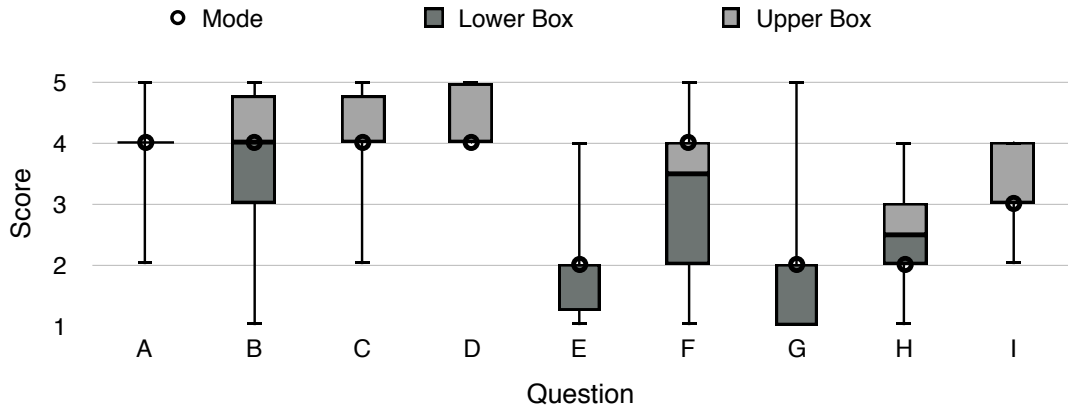



Figure 4: Box-and-whisker plot of survey questions. Questions are scored from 1 (*totally disagree*) up to 5 (*totally agree*). A vertical bar between boxes show the median, the box itself represents 50% of the answers. Whiskers show the minimum and maximum score students provided. The mode is represented by a dot. Refere to Table 1 as a legend about questions *A* till *I*.

| | 2016 | 2015 | 2015 (resit) | 2013 |
|---|---|---|---|---|
| **Part of exam on monads** | | | | |
| – Remember | 5% | 5% | 5% | 0% |
| – Apply | 12% | 10% | 21% | 0% |
| – Create | 8% | 10% | 9% | 10% |
| Total | 25% | 25% | 35% | 10% |
| **Number of participants** | 29 | 36 | 4 | 46 |
| **Average mark monad questions** | 6.6 | 6.3 | 4.4 | 4.7 |
| **Average mark total** | 7.0 | 7.0 | 6.1 | 4.2 |

Table 2: Percentages of four final exams, denoting the fraction that is about monads, the number of participants for each exam, and the average final mark on the monad part of the exam. Questions are classified in using the knowledge levels *remember*, *apply* and *create* as discussed in Section 4.3.

Table 2 also displays the number of participants of each exam and the average mark on a scale from 0 to 10. This year 29 of the 39 students participated in the exam, which results in a participation rate of 74%. The average mark *on the monad part* this year was a 6.6. The differences in question types and the number of participants every year make it hard to compare the exam results. We can state that, despite these dissimilarities, the usage of blueprints throughout the course does not have a negative impact on the average mark.

In the next subsections we relate the results of the questionnaire to answers given by students during the interview. Quotations start with a label $S_{i,j}$ or $R_i$, indicating a student or researcher respectively. A student $S_{i,j}$ is student $j$ in group $i$.

## 5.1 Impact of Blueprints

From questions $C$ ("The pictures are easy to understand", $Md = 4.0$, $IQR = 0.75$), $D$ ("The pictures do not get in the way of understanding what a function does", $Md = 1.0$, $IQR = 1.00$) and $E$ ("The pictures do not add any value", $Md = 2.0$, $IQR = 0.75$) we can conclude students are mildly positive about the comprehensibility of blueprints. Especially the answers to $D$, which only range between 4 and 5, strongly suggests blueprints at least have no negative effect. Figure 4 also shows that more than 50% of the participants rated question $E$ with 1 or 2, where most students choose the second option. This is also apparent from the interview. For example, one of the students stated:

> $S_{2,2}$: *"It particularly helped me to get an idea of what a monad is."*

Blueprints help students to understand the examples given during the lecture. The related question $A$ ("The pictures helped me to understand the examples in the slides better", $Md = 4.0$, $IQR = 0.00$) is the only one which at least half of the participants judge with 4. Adding blueprints to the lecture had a positive effect.

> $S_{1,1}$: *"I think the lecture was put together surprisingly well. [...] [You got an idea of] why you would use [a monad]. Especially during programming, to recognize when you can use something like [a monad]. So I liked it very much."*

During the exercises, several students also indicated that the inclusion of blueprints was useful for them.

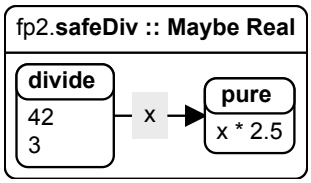



Figure 5: One of the blueprints in the assignment. Refered to by a student during the interview.

> $S_{1,2}$: *"It was nice that [pictures] were [printed] on paper. You did not need to look back at the code every time to create a new picture [in your head]."*

One pair of students explicitly stated that they consistently studied the blueprints first, before studying the code for the exercise. They used the the generated pictures to understand the programs, rather than the code.

> $R_1$: *"So instead of looking at the code you looked occasionally at the pictures, to see how the code works? Do I summarize that correctly?"*
>
> Both $S_{1,1}$ and $S_{1,2}$: *"Yes."*

Another pair of students indicated that the blueprints helped them in devising a working solution to an exercise.

> $R_1$: *"How arrive at this answer so quickly?"*
>
> $S_{6,2}$: *"By using the picture. [...] Here you just divide the two numbers which are passed as arguments.* [points at the blueprint in Figure 5] *From this you get an [answer]* `x`*. And then you multiply this* `x` *by* `2.5` *and you make a pure [value] of it."*

Students that did not use blueprints in the first place, still point out they can use them and might use them in the future.

> $S_{5,1}$: *"In itself [the picture] is very clear [...] Maybe it'll help me in the last part about the state monad after all."*

When asking another student if he now understands the picture:

> $R_1$: *"Can you use the picture if you take a look at it now?"*
>
> $S_{6,1}$: *"In retrospect: yes. I think it will help me remember [monads] better."*

While some students did indicate that they would draw their own blueprint on paper in an attempt to solve an exercises, most used them as a passive tool: they merely studied the provided blueprints.

> $S_{2,2}$: *"We did not create pictures ourselves."*

## 5.2 Mental model

Blueprints help students to create a mental model. They can reson with the different parts of the blueprint without being explicitly introduced to them during the lecture.

> $S_{2,1}$: *"[In this picture] you do not have so many steps.* [pointing at the blueprint in Figure 3] *Something comes out of this [box] and you use that to put it in the other [box]. And something will come out of that [box] again. It is useful to have that whole idea of down passing thing on displayed in a picture, because the code looks quite abstract."*

The pictures help in ordering steps to reach an answer:

> $S_{1,1}$: *"If you write these functions down in bind notation, they are exactly in the same order as the pictures."*

Also, blueprints support the mental model students create in their minds.

> $S_{1,1}$: *"I believe that [...] if you know what you want to do, you already have such a picture in your head, I guess."*

### 5.3 Visual Orientated versus Type Orientated

Not all students are visually orientated. Those who are not, take less advantage of the blueprints.

> $S_{5,1}$: *"No, I must say that I used the pictures very little and that I hardly looked at them. [...] I didn't need them."*

This student however, states she is not visually orientated.

> $S_{5,1}$: *"No, [I am] not [visually oriented] at all. [I am] definitely very theoretically oriented."*

Indeed, she takes a theoretical point of view towards monads and reasons a lot with types.

> $S_{5,1}$: *"Especially the realization that you can get a lot of [information] out of the types. So if you look at the types, that tells you a lot about the functions."*

This attests the big spread in the answers to questions $B$ ("The pictures helped me to understand the exercises better", $Md = 4.0$, $IQR = 1.75$) and $F$ ("When I got stuck on an exercise, I would look at the pictures again.", $Md = 3.5$, $IQR = 2.00$) from the questionnaire. Both have the highest interquartile range and span the entire domain of possible answers (1 up to 5).

## 6 Discussion and Conclusion

Using blueprints in the lecture slides and in the assignments has had mostly positive impact. In general, students indicate that blueprints helped them to understand both the slides and the exercises. The students found blueprints mostly easy to understand. Blueprints can help the students to create a mental model about the execution of monadic code.

Adding blueprints to the slides and assignments did not get in the way of learning. This is reflected in the exam results, which for the monad-related questions are higher than last year's results, but not significantly so. The average marks on the entire exam from 2016 are as good as those from 2015, and substantially better than the 2013 exams.

Students did not draw blueprints themselves during their work on the exercises. They do use the blueprints provided with the exercises and exam as a reference. In some cases, blueprints are preferred over the code when the students attempt to comprehend the example programs. Whether they would experiment with creating their own blueprints if they had some way of generating them is unknown. Some students did not use blueprints at all. These students indicated that they are not visually orientated or are more comfortable to reason on the level of types.

# 7 Future Work

We would like students to be able to generate both static and dynamic blueprints for their own programs. This would give them an environment in which they could experiment with the relationship between code and blueprints, possibly aiding them in understanding monads better. A new case study could then be performed to measure the usefulness of having such a tool.

To create such an experimentation environment, we would need a stable stand-alone Tonic viewer. Such a stand-alone viewer presents us with several challenges. First, we must communicate between the application being executed and the viewer. We currently use a TCP/IP connection to do so. Second, we must keep track of the program execution, so that we can highlight the correct blueprint nodes. Rather than relying on iTasks' built-in task identification system, we must treat programs as black boxes and we need to come up with some external mechanism to identify our progress. Using a stand-alone Tonic viewer in the context of teaching monads is a next logical step for our research.

Having a stand-alone Tonic viewer would also bring benefits outside of education. The viewer could be used as a general tracer/debugger tool. It could be a stand-alone tool or part of an IDE. Integration with an IDE is particularly interesting, because it could allow blueprints to become an integral part of the development process.

Should a stand-alone Tonic experimentation environment be received favorably by the students and possibly IDE users, we could consider performing an experiment in which we would test the effect of such a tool on the students' grades. In such an experiment, we would need to create an experimental group and a control group, where the former would use Tonic and the latter won't. Performing such an experiment would be challenging, however. It is difficult to minimize the number of variables in the experiment, for example. Additionally, on the logistics side we would require a larger student population and additional teaching facilities and staff. Ethical questions would also come into play. Is it right to deprive a group of students from a learning tool, possibly negatively impacting their academic performance?

The recordings from the practical session were mainly used to prepare the interviews. This material can be analyzed in more detail in future research. We plan to use the recordings made for this research to get a better understanding of students problems working with monads. We may be able to answer questions like "When do students encounter difficulties with monads?", "What kind of difficulties do they encounter?" and "How do they solve such difficulties?" using these recordings.

# 8 Related Work

Many studies concentrate on difficulties novices encounter during programming education. Robins et al. [19] give a detailed overview of, amongst others, the differences between *novice* and *expert* programmers, and programming *knowledge* versus programming *strategies*. The ITiCSE working group lead by McCracken compared programming skill assessments of multiple institutions across the globe [10]. Common in these and other studies [26, 25, 5, 6] is that they are oriented towards procedural and object orientated languages. In addition, they only use programming exercises made by students as their main data source.

Tonic is not the first visual language for functional programming. Visual Haskell [17] and VisaVis [16] already attempted to put forth a graphical language in the 1990's. A decade later, Vital [3] attempted to do so again. In 2012, Henrix et al. [4] introduced GiN: Graphical iTasks Notation as a graphical programming language for iTasks programs. What separates Tonic from the aforementioned approaches are

mainly two things. First, Tonic does not aim to be a visual programming language, whereas the other approach do. Instead, Tonic is a visual language geared towards *understanding*, rather than creating. With Tonic, programmers still write code in a textual formalism. Blueprints are subsequently *generated*. Other approaches attempt to *replace* the textual formalism with a graphical one. Second, earlier graphical languages visualize a program at the function level. As a result, all low-level bookkeeping aspects need to be visualized. This distracts from the core of what a program is intended to do: support the workflow of its users. Tonic, on the other hand, visualizes programs on the monadic level, abstracting from most bookkeeping operations. As a result, Tonic's blueprints are less cluttered and therefore possibly easier to understand.

### Acknowledgements

The authors like to thank Peter Achten for his constant feedback and his enthusiasm in this project.

This research is supported by the Dutch Technology Foundation STW, which is part of the Netherlands Organization for Scientific Research (NWO), and which is partly funded by the Ministry of Economic Affairs.

## References

[1] Peter Achten, Jurriën Stutterheim, László Domoszlai & Rinus Plasmeijer (2015): *Task Oriented Programming with Purely Compositional Interactive Scalable Vector Graphics*. In: *Implementation and Application of Functional Languages*.

[2] Benedict du Boulay (1986): *Some difficulties of learning to program*. *Journal of Educational Computing Research* 2(1), pp. 57–73.

[3] Keith Hanna (2002): *Interactive visual functional programming*. *ACM SIGPLAN Notices* 37(9), pp. 145–156, doi:10.1145/581478.581493. Available at `http://dl.acm.org/citation.cfm?id=581478.581493`.

[4] Jeroen Henrix, Rinus Plasmeijer & Peter Achten (2012): *GiN: A Graphical Language and Tool for Defining iTask Workflows*. *Trends in Functional Programming*, pp. 163–178, doi:10.1007/978-3-642-32037-8_11.

[5] Simon Holland, Robert Griffiths & Mark Woodman (1997): *Avoiding object misconceptions*. In: *ACM SIGCSE Bulletin*, 29, ACM, pp. 131–134.

[6] Maria Hristova, Ananya Misra, Megan Rutter & Rebecca Mercuri (2003): *Identifying and correcting Java programming errors for introductory computer science students*. *ACM SIGCSE Bulletin* 35(1), pp. 153–156.

[7] Simon Peyton Jones (2003): *Haskell 98 language and libraries: the revised report*. Cambridge University Press.

[8] D R Krathwohl (2002): *A revision of Bloom's taxonomy: An overview*. *Theory into practice* 41(4), pp. 212–218.

[9] John Lyle (2003): *Stimulated Recall: A Report on Its Use in Naturalistic Research*. *British Educational Research Journal* 29(6), pp. 861–878. Available at `http://www.jstor.org/stable/1502138`.

[10] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting & Tadeusz Wilusz (2001): *A multi-national, multi-institutional study of assessment of programming skills of first-year CS students*. *ACM SIGCSE Bulletin* 33(4), pp. 125–180.

[11] Eugenio Moggi (1991): *Notions of Computation and Monads*. *Inf. Comput.* 93(1), pp. 55–92, doi:10.1016/0890-5401(91)90052-4.

[12] Object Management Group (2009): *Business Process Model and Notation (BPMN) Version 1.2*. Technical Report, Object Management Group.

[13] DN Perkins & Fay Martin (1986): *Fragile knowledge and neglected strategies in novice programmers*. In: *first workshop on empirical studies of programmers on Empirical studies of programmers*, pp. 213–229.

[14] Simon L. Peyton Jones & Philip Wadler (1993): *Imperative Functional Programming*. In: *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, ACM, New York, NY, USA, pp. 71–84, doi:10.1145/158511.158524.

[15] Rinus Plasmeijer, Bas Lijnse, Steffen Michels, Peter Achten & Pieter Koopman (2012): *Task-Oriented Programming in a Pure Functional Language*. In: *Proceedings of the 2012 ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, PPDP '12*, ACM, Leuven, Belgium, pp. 195–206.

[16] Jörg Poswig, Guido Vrankar & Claudio Morara (1994): *VisaVis: a Higher-order Functional Visual Programming Language*. *Journal of Visual Languages & Computing* 5(1), pp. 83–111, doi:10.1006/jvlc.1994.1005. Available at `http://linkinghub.elsevier.com/retrieve/pii/S1045926X84710056`.

[17] H John Reekie (1994): *Visual Haskell: a first attempt*. Technical Report.

[18] Hideki John Reekie (1995): *Realtime Signal Processing – Dataflow, Visual, and Functional Programming*. Ph.D. thesis, University of Technology at Sydney, Australia.

[19] Anthony Robins, Janet Rountree & Nathan Rountree (2003): *Learning and teaching programming: A review and discussion*. *Computer Science Education* 13(2), pp. 137–172.

[20] James Rumbaugh, Ivar Jacobson & Grady Booch (2004): *Unified Modeling Language Reference Manual, The*. Pearson Higher Education.

[21] Juha Sorva (2013): *Notional Machines and Introductory Programming Education*. *Trans. Comput. Educ.* 13(2), pp. 8:1–8:31, doi:10.1145/2483710.2483713. Available at `http://doi.acm.org/10.1145/2483710.2483713`.

[22] Jurriën Stutterheim, Peter Achten & Rinus Plasmeijer (2016): *Static and Dynamic Visualisations of Monadic Programs*. In: *Implementation and Application of Functional Languages*, ACM, Koblenz, Germany, pp. 1–13, doi:10.1145/2897336.2897337. Available at `http://dx.doi.org/10.1145/2897336.2897337`.

[23] Jurriën Stutterheim, Rinus Plasmeijer & Peter Achten (2015): *Tonic: An Infrastructure to Graphically Represent the Definition and Behaviour of Tasks*. In Jurriaan Hage & Jay McCarthy, editors: *Trends in Functional Programming*, *Lecture Notes in Computer Science* 8843, Springer International Publishing, pp. 122–141. Available at `http://dx.doi.org/10.1007/978-3-319-14675-1_8`.

[24] Philip Wadler (1995): *Monads for Functional Programming*. In: *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, Springer-Verlag, London, UK, UK, pp. 24–52.

[25] Susan Wiedenbeck & Vennila Ramalingam (1999): *Novice comprehension of small programs written in the procedural and object-oriented styles*. *International Journal of Human-Computer Studies* 51(1), pp. 71–87.

[26] Susan Wiedenbeck, Vennila Ramalingam, Suseela Sarasamma & CynthiaL Corritore (1999): *A comparison of the comprehension of object-oriented and procedural programs by novice programmers*. *Interacting with Computers* 11(3), pp. 255–282.