VisualFP

Designing a Visual, Block-Based Environment to Create & Execute Haskell Code

Department of Computer Science
OST - Eastern Switzerland University of Applied Sciences
Campus Rapperswil-Jona

Student Research Project, Autumn Term 2023

Authors:	Lukas Streckeisen, Jann M. Flepp
Advisor:	Prof. Dr. Farhad D. Mehta
Project Partner:	IFS Institute for Software

Abstract

Most visual programming tools used to introduce children & young adults to the programming world are based on the imperative paradigm. Existing tools based on functional programming either lack a good user experience or hide critical concepts of the functional paradigm.

To address this gap, a visual, block-based tool for functional programming should be designed. This project aims to find a visual design of such a tool and then prove its feasibility in a proof of concept application.

Existing visual programming tools are examined before creating a visual design. The development of the design is approached in two iterations: In a first step, concept drafts are based on researched tools and evaluated using a survey. Then, a new concept is created using the received feedback and implemented in a proof of concept.

The created design concept focuses on function composition, guided by type holes that indicate the type required for a valid function definition.

The implemented application proves that the proposed concept for function composition works as envisioned. It includes an inference engine that determines the type of undefined parts of a function and is built using Electron.js & Haskell.

It is recommended that an additional project be conducted to implement missing features of the application so that it can be used in classrooms.

Keywords: Haskell, Functional Programming, Visual Programming

Management Summary

Initial Situation

Many teachers use tools like Scratch or LEGO Mindstorms when introducing children and young adults to the programming world. Such visual, block-based tools eliminate the hurdle of code syntax, allowing beginners to concentrate on the program they want to write.

However, almost all visual tools for teaching programming are made for the imperative programming paradigm. Visual tools exist for functional programming, but either lack a good user experience or hide essential concepts required to understand functional programming.

Objective

With VisualFP, a visual, block-based tool should be designed that can be used to teach functional programming. At the center of this project is a design concept for visual function composition, describing how the visual editor of such a tool would work. A proof of concept application with a visual function editor should be created to prove the concept is feasible.

A potent type inference engine is necessary for such an editor to work, which shall be implemented using a unification algorithm as proposed by Simon Peyton Jones [1]. An overview of a unification-based engine is shown in Figure 1.

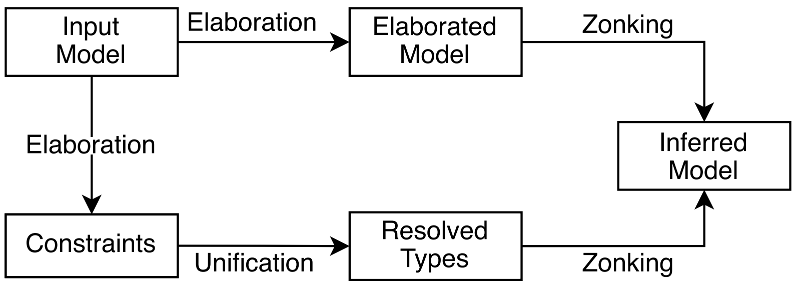


Figure 1: Type-inference engine components

Additionally, the application should run on the user's machine so that it can be used in classrooms without server infrastructure.

Results

The developed concept uses nested blocks to represent function definitions. Type holes guide the development flow as typed placeholders for missing pieces of a value definition. Users can drop value blocks into a type hole to fill it with that value. Value blocks are provided by the editor or are defined by the user.

The concept was implemented in a proof of concept application written using Haskell and Electron.js. A component-level view of the application is provided in Figure 2.

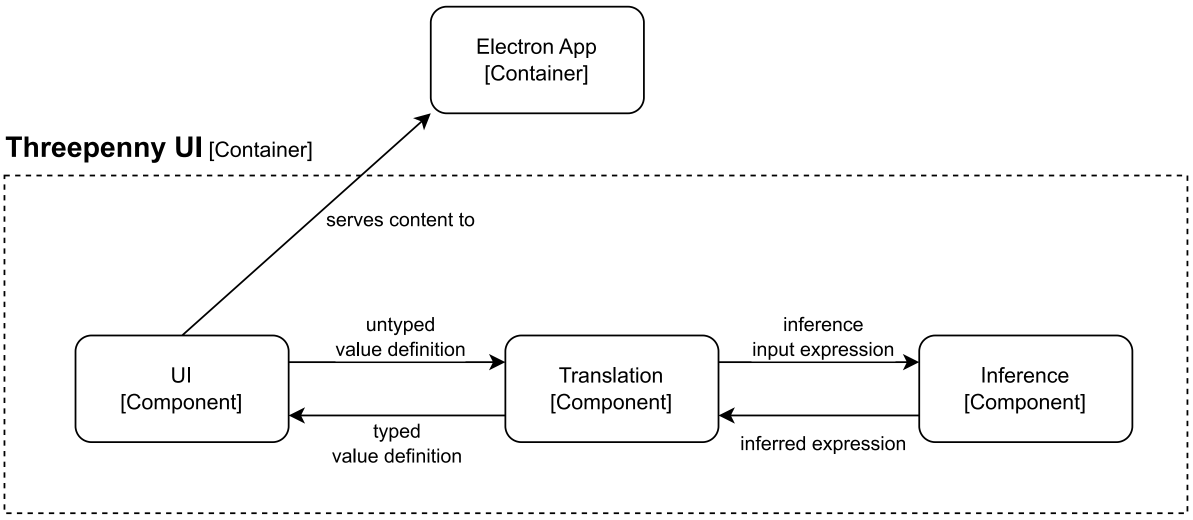


Figure 2: C4 Component Diagram for VisualFP PoC

The application includes a small selection of pre-defined values that can be used to build a user-defined function. A screenshot of the application is provided in Figure 3.

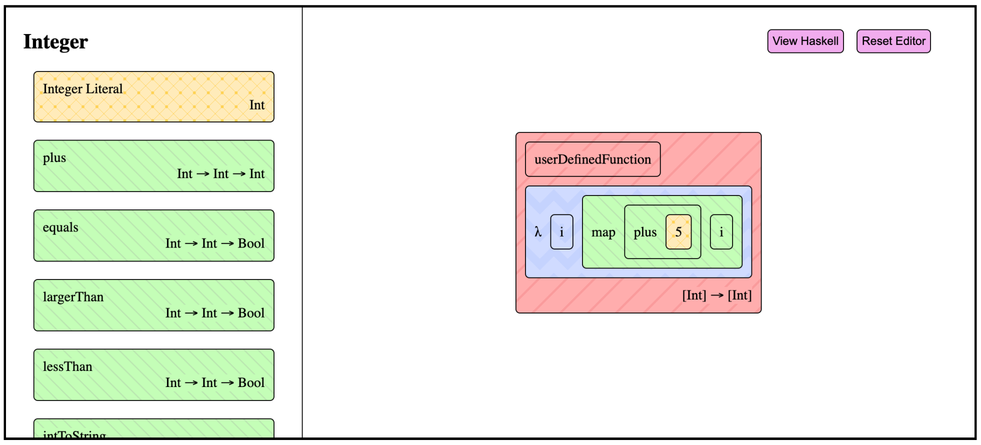


Figure 3: Screenshot of the mapAdd5 function definition in VisualFP

The implemented application proves that the developed design concept works as envisioned. However, the application is not yet ready to be used in classrooms as additional design and development is required to bring the idea to its full potential.

Table of Contents

Part I - Introduction

This part discusses the motivation behind VisualFP, what it aims to achieve, and the research that was done on existing alternatives.

Section 1 Motivation and Section 2 Goals are based on this project's task description, which can be found in full in Section 16.

1. Motivation

Block-based programming tools enable a purely visual introduction to programming for the imperative programming paradigm. Common beginner mistakes, such as compiling code with incorrect syntax, are impossible in such an environment. Examples include Scratch, developed by MIT, and the LEGO Mindstorms software.

There aren't many similar tools available to teach functional programming, which is why most educators start their courses by either showing the logical nature of functional programming through e.g. the lambda calculus, or by jumping right into code, leaving many students puzzled.

Some visual approaches to functional programming exist; the ones researched during this project are covered in Section 3. Unfortunately, none seem suitable as an introduction to general-purpose functional programming.

However, block-based tools don't need to be limited to education: A sufficiently powerful environment that allows conversion to and from a text-based programming language could also see adoption among professionals looking for ways to visualize their code.

2. Goals

The goal of this student research project is to design a tool, VisualFP, that allows the graphical development of functional code and is block-based. The target audience of VisualFP are students learning to program.

The core functionality of VisualFP will be implemented in a PoC (Proof of Concept) to prove the feasibility of the design.

The central part of this project is to document the design process used to create VisualFP transparently to allow others to better understand the decisions made during the project, which alternatives were considered, and to evaluate whether they want to follow the same path.

Due to the time constraints of this project, the goals deviate from the initial task description in Section 16. While support for experienced programmers who want to view their code in a visual context is included in Section 4, interoperability with Haskell isn't a goal for this project. In addition, the PoC is treated as a sample application to prove that the design concept works and not necessarily as a starting point for fully implementing the VisualFP application.

3. Existing tools

There already are tools available that could fill the role of a visual functional programming language as described in Section 1.

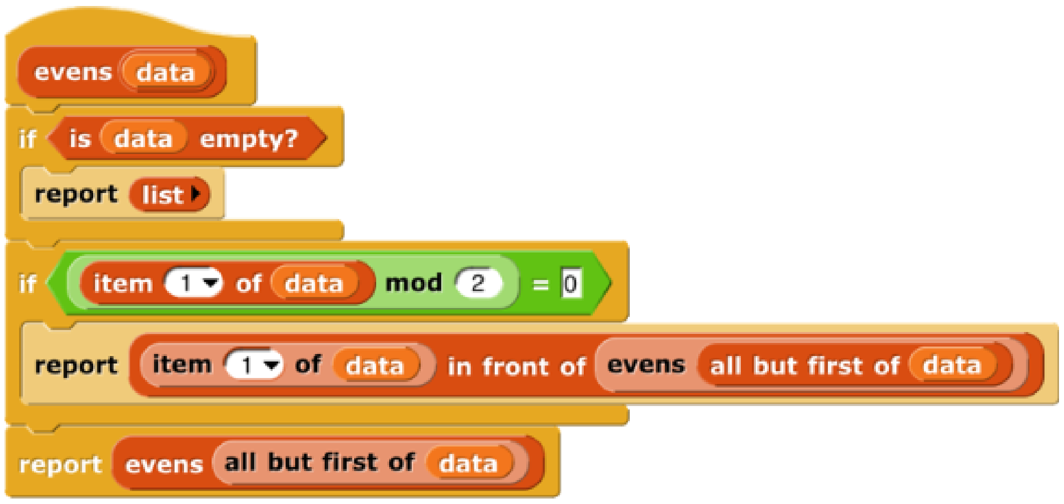
But even if they fail to live up to the specific goals of this project, they may still be helpful as inspiration or a starting point for a VisualFP.

This section discusses these tools, their strengths and weaknesses.

3.1. Snap!

Snap! is a block-based programming tool developed by Berkeley University which allows the creation of imperative programs in a Scratch-like manner.

A user can create programs to control a cursor in a graphical environment, e.g., navigate the cursor to a specific position and draw a line. For that, Snap! offers pre-defined commands like basic arithmetic operations, cursor-, pen- & sound-controls, and flow controls like "if" blocks. For lists, snap offers some control blocks that allow users to work with lists in a functional fashion, as seen in the red blocks in Figure 4.



```
evens data
if is data empty?
    report list
if item 1 of data mod 2 = 0
    report item 1 of data in front of evens all but first of data
report evens all but first of data
```

Figure 4: Screenshot of a block expression in Snap! [2]

However, everything else can only be done imperatively, making the functional aspects more of an additional feature than a core concept of Snap!.

Users can also create new block commands based on existing commands, which allows users to reuse their code. But unlike usual functions, custom blocks aren't able to take arguments from their caller [2].

Regarding usability, we feel that Snap! isn't very difficult to understand, but also not very intuitive, primarily because some icons and command names aren't obvious in their meaning.

3.2. Eros

Conal Elliott developed a way to visualize pure values in an interactive and composable way. He calls this technique "Tangible Functional Programming" [3]. The technique allows non-technical people to create content based on combinations of pure values.

At the core of the technique are *tangible values*, which are pure values, including functions, that can be visualized and composed with other tangible values through a graphical user interface.

To combine such values, a set of algebras is provided that allows values to be applied to each other, even if they are nested in functions or tuples. Elliot calls this concept *deep application*. [3]

Eliott also developed an application called Eros that implements these techniques. Eros is particularly suited to creative people with an artistic interest. A screenshot of Eros is shown in Figure 5.

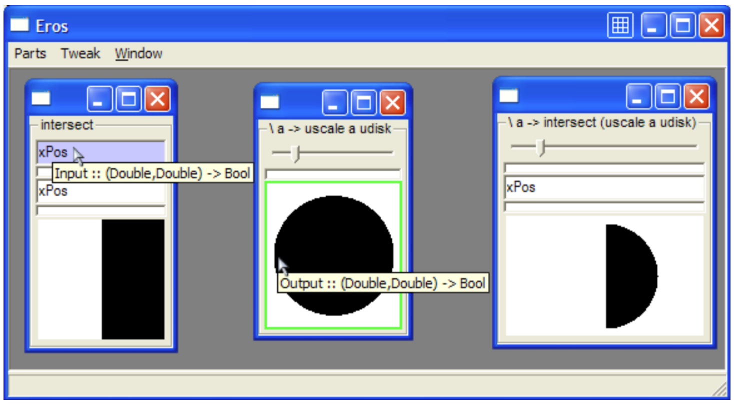

Eros
Parts
Tweak
Window
intersect
xPos
Input :: (Double,Double) -> Bool
xPos
\ a -> uscale a udisk
Output :: (Double,Double) -> Bool
\ a -> intersect (uscale a udisk)
xPos

Figure 5: Screenshot of Eros [3]

“Tangible Functional Programming” is a fascinating technique, and particularly the way Eros visualizes pure values can be an inspiration for VisualFP. But ultimately, the technique appears unsuitable for general-purpose functional programming, especially in an educational context.

3.3. flo

flo is a visual and functional programming language. It features a programming environment based on blocks connected using cables, as shown in Figure 6. The corresponding compiler converts the visual arrangements and connections into Haskell code.

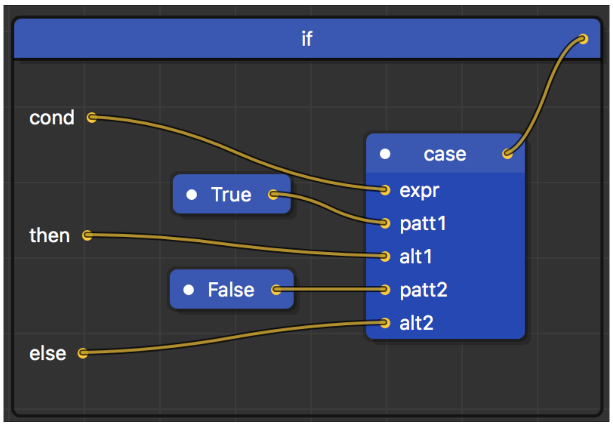


Figure 6: Screenshot of an if function definition in flo [4]

A block's parameters and outputs are represented by sockets, which can be connected to compatible sockets through click-and-drag. The compiler can infer the sockets' types and reject incompatible connections.

A specialty of flo is that blocks represent values and types. A type block is either a basic type, such as `Int` or `Bool`, or a constructor to a complex type with type parameters, represented through sockets. An example of a type being used as type parameter is shown in Figure 7.

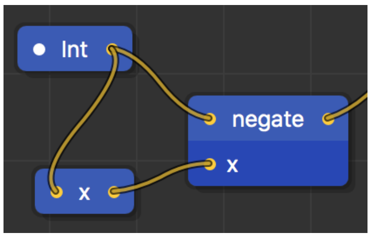


Figure 7: Screenshot of a negation function application in flo [4]

flo was a research project and has not been actively developed since 2016. Out of all researched tools, it is probably the one that comes closest to VisualFP in terms of its goals.

3.4. Enso

Enso is a functional programming language designed for data science that Enso International Inc [5] created. There is a text and a visual editor to create programs.

The visual editor allows a user to define components that can be connected, symbolizing the data flow from one component to another. The editor also offers previews for a component's data, which, e.g., allows a user to see a modified picture like in Figure 8.

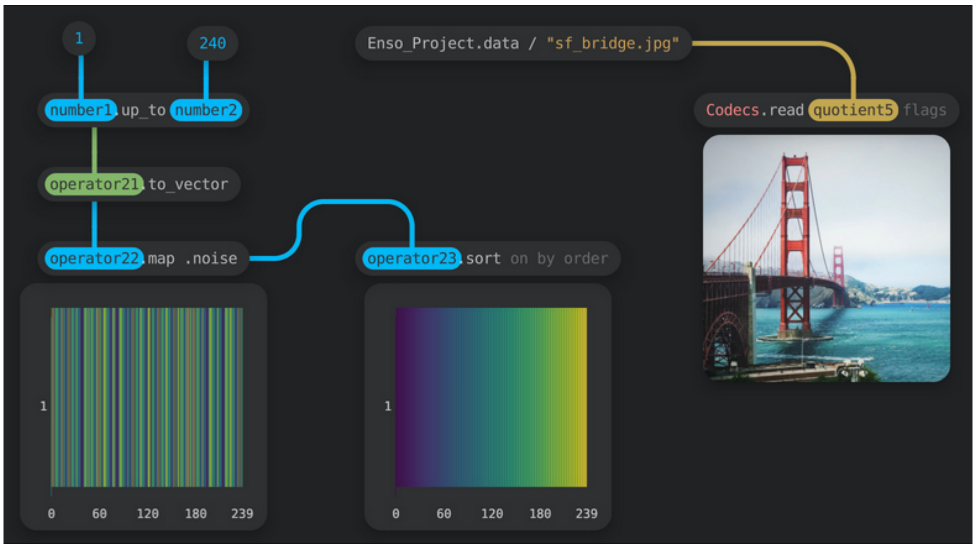


Figure 8: Example program in Enso [5]

Enso is visually impressive and largely intuitive regarding data flow. For example, downloading data from a public API (Application Programming Interface) and aggregating it is super easy. However, some operations, such as dividing a number with another number, are pretty complicated. Based on that, Enso seems to be an excellent tool to work with datasets but not so much for creating programs with complex logic.

3.5. Reddit Suggestion for visual pure functional programming

The Reddit user 'Jameshfisher' suggested a style of pure visual programming [6]. It is only theoretical and, therefore, not an existing tool. A picture of the suggestion is shown in Figure 9.

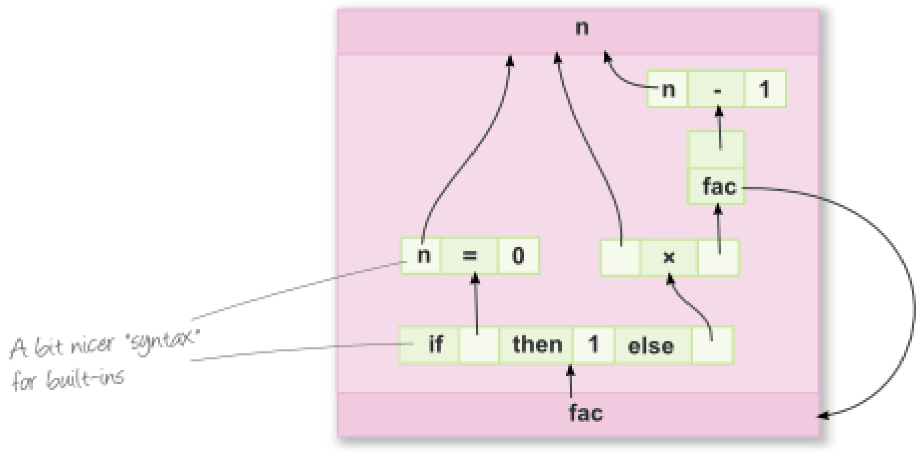


Figure 9: Suggestion for visual pure function programming by Reddit user Jameshfisher
Source: Adapted from [6]

But the proposed concept is very intriguing. Instead of drawing a function from the outside, functions are drawn from the inside.

A function's input is displayed on the top, and the output on the bottom. The arrows show how the output is composed of other expressions.

Out of all the researched tools, this suggestion shows functional programming concepts best.

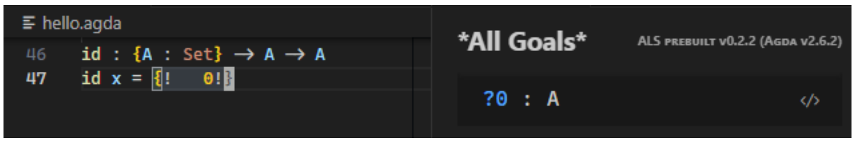
3.6. Agda

Even though Agda[1] isn't a visual programming environment, the tools for it have been designed in an interactive way and are of interest to this project.

The basis for these tools is a language-server, which, in combination with the powerful checker and editor extensions not only allow to verify the correctness of the code, but also to check incomplete programs. Additional context, such as checker errors or incomplete expressions, is then provided in a separate window inside the editor.

Using this tooling, it is, for example, possible to insert a hole in the code, a so-called *metavariable*, which the checker will detect and display alongside the expected type in the context window, as shown in Figure 10.

[1]https://wiki.portal.chalmers.se/agda/pmwiki.php



```
id : {A : Set} → A → A
id x = {!   0!}
```

```
?0 : A
```

Figure 10: Screenshot of Agda context window after inserting a hole in an expression using the ? sign

The Agda language server is also able to provide automatic case-splitting for pattern matching. During this process, the checker will determine the possible cases and present only those to the user. Thus, the user is freed from remembering and typing out all matching cases.

Additionally, the language server offers to normalize or deduce any given expression inside the context of the currently loaded code. The checker is also utilized for syntax highlighting, providing the user with semantical value in the form of coloring.

Part II - Design

This part describes the requirements, the design proposals, the design evaluation process, and the detailed final design for VisualFP.

4. Functional Requirements

The following section describes all actors and use cases identified for the VisualFP application.

4.1. Actors

VisualFP has two actors:

Student User The student user is the primary user of VisualFP and, therefore, the main influence on the visualization design. The student user wants to learn functional programming using VisualFP. They want to do that by visually composing functions in a simple UI (User Interface). The UI should simplify understanding functional concepts that many beginners struggle with.

Expert User The expert user is an experienced professional who wants to use VisualFP to help them understand their code better. For that, they want to import their existing Haskell project into VisualFP.

4.2. Use Cases

Figure 11 gives an overview of all identified use cases. By default, "user" in the use case description refers to the "student user".

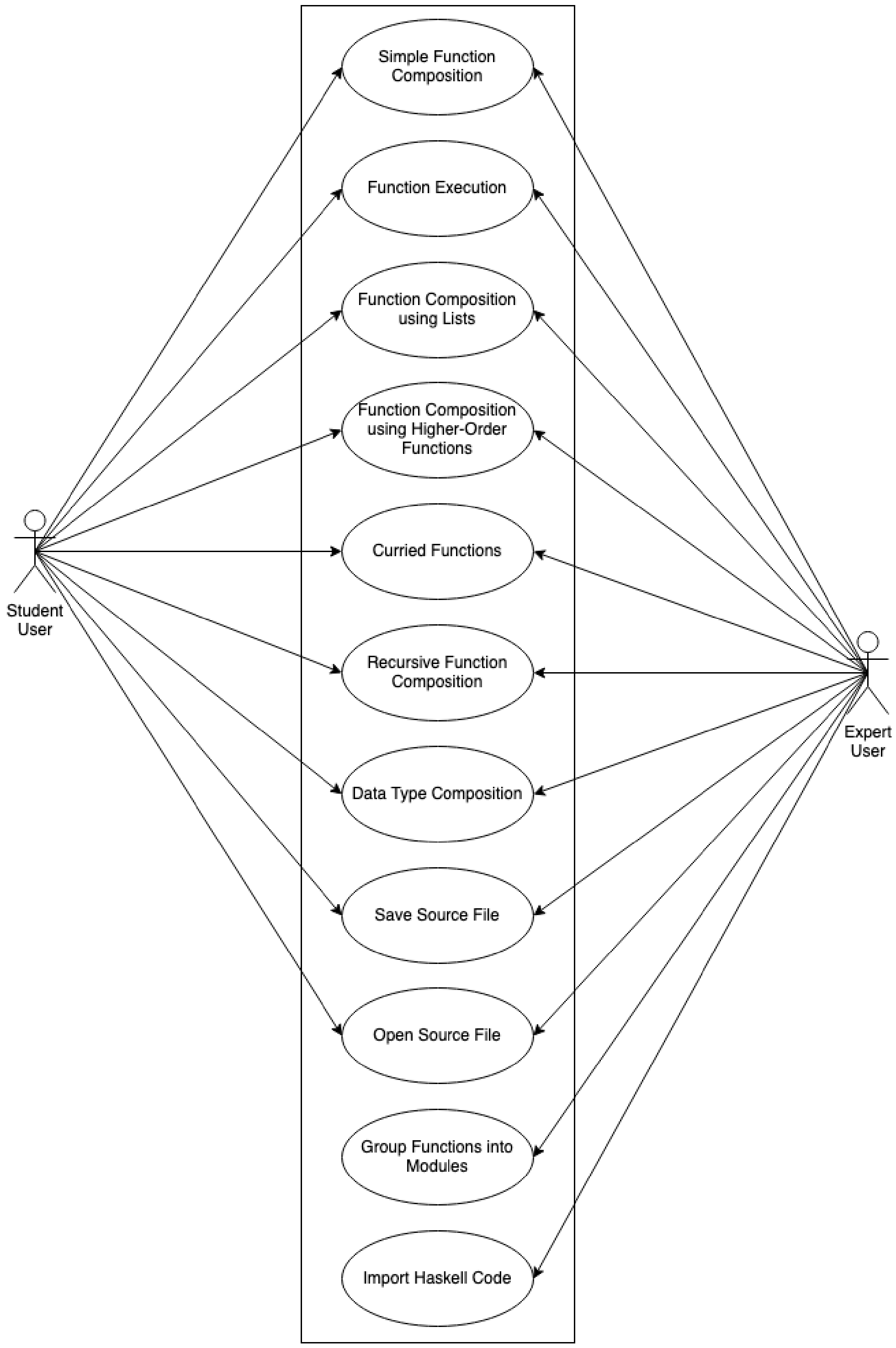

Simple Function Composition
Function Execution
Function Composition using Lists
Function Composition using Higher-Order Functions
Curried Functions
Recursive Function Composition
Data Type Composition
Save Source File
Open Source File
Group Functions into Modules
Import Haskell Code
Student User
Expert User

Figure 11: Use Case Diagram

As the aim of this project is to find a visual representation of functional programming, the use case descriptions are kept very brief and only state the intention behind the use case.

4.2.1. UC1 - Simple Function Composition

A user wants to compose a simple function using pre-defined functions, e.g., Integer parameters.

4.2.2. UC2 - Function Execution
A user wants to execute their visually composed functions to see the effects of their functions on data.

4.2.3. UC3 - Recursive Function Composition
A user wants to compose a function that is defined using itself. To do so, the user needs possibilities to distinguish between a recursive and a base case.

4.2.4. UC4 - Function Composition using Higher-Order Functions
To create reusable and composable functions, a user wants to compose functions that take other functions as their input, in other words, higher-order functions.

4.2.5. UC5 - Curried Functions
A user wants to create a function by partially applying a curried function.

4.2.6. UC6 - Function Composition using Lists
A user wants to compose a function using lists, so that they can collect data and process it further.

4.2.7. UC7 - Data Type Composition
A user wants to be able to create their own data types to represent data of their problem domain accurately.

4.2.8. UC8 - Save Source File
A user wants to save their composed functions in a source file so they can keep their work when, e.g., restarting their computer.

4.2.9. UC9 - Open Source File
A user wants to open a previously saved source file to continue working on their program.

4.2.10. UC10 - Group Functions into Modules
An expert user wants to group functions into modules to keep their code organized.

4.2.11. UC11 - Import Haskell code
An expert user wants to import their existing Haskell project into VisualFP so they can get a better understanding of their code from its visualization.

4.3. Prioritization & Scope
The focus of this project lies in creating a design that allows to develop functional applications visually and is suitable for beginners.

Use cases 1 - 6 have been deemed more important to reach this goal and thus have higher priority than use cases 6 - 9. Use cases 10 and 11 are not in this project's scope but are listed for completion.

5. Non Functional Requirements
This section describes all non-functional requirements identified for VisualFP. To find a relevant NFR (Non-Functional Requirement), ISO-25010 [7] was used for inspiration.

5.1. NFR1 - Platform Compatibility

VisualFP should be usable on Windows, MacOS, and Linux devices. No extra effort should be required to run VisualFP on any particular OS (Operating System).

Verification Test the usage of VisualFP on all three mentioned operating systems.

Acceptance Criteria The installation steps are the same or of equivalent effort for all three mentioned operating systems

Realisation Usage of platform-independent technologies

5.2. NFR2 - Learnability

Since VisualFP targets students who want to learn functional programming, the learning effort shouldn't be on the tool itself but on functional concepts.

Verification Usability Tests with a user without experience in functional programming

Acceptance Criteria A user without experience in functional programming understands how to use VisualFP within 1 hour

Realisation Keep the design of VisualPF simple; offer help buttons on more complex blocks

6. First Design Iteration

For the first iteration, multiple design directions were created and evaluated. The following process was chosen to gain as much insight as possible:

1. Section 6.1 defines a set of evaluation criteria to compare the designs in the form of a questionnaire.
2. Then, three proposals are presented: A Flo-based design in Section 6.3, a Scratch-based design in Section 6.2, and a design inspired by the Haskell function notation in Section 6.4.
3. The three designs are filled into a questionnaire and handed to a selected survey group. The feedback is then further discussed in Section 6.5.

6.1. Design Evaluation Criteria

Since it is difficult to compare designs in a quantitative manner, the design evaluation process is based on selected code scenarios and a non-quantitative questionnaire.

A survey will be conducted using the questionnaire and example visualizations for specific code scenarios to get further valuable feedback to improve initial designs. The survey targets a selected group of students and some more experienced functional programmers.

The code scenarios and questionnaire questions can be found below.

6.1.1. Code Scenarios

These code scenarios were defined to evaluate visualization designs regarding their simplicity and clarity of the underlying functional concepts.

Simple Addition Function

Listing 1 has been chosen to evaluate designs for the composition of a simple function using another function.

```
addition :: Num a => a -> a -> a
addition a b = (+) a b
```

Listing 1: Addition function for design evaluation

Even numbers from 1 to 10

Listing 2 has been chosen to evaluate designs for list handling.

```
evenOneToTen :: Integral a => [a]
evenOneToTen = [x | x <- [1 .. 10], even x]
```

Listing 2: Function that returns even numbers between 1 and 10

Product of Numbers

Listing 3 has been chosen to evaluate designs for recursive functions.

```
product :: Num a => [a] -> a
product [] = 1
product (n : ns) = (*) n (product ns)
```

Listing 3: Product function for design evaluation

Map Add 5 Function

Listing 4 has been chosen to evaluate designs for currying.

```
mapAdd5 :: Num b => [b] -> [b]
mapAdd5 = map ((+) 5)
```

6.1.2. Evaluation Questionnaire

The survey participants are asked to answer the following questions for every design proposal:

- Were you able to understand the meaning of the boxes and arrows?
- Do you find the concept nice to look at?
- Could you imagine teaching functional programming using this visualization?
- Could you imagine how the concept scales to more complex expressions?
- Do you have any suggestions for improvement or general comments on the concept?

Additionally, every survey participant can suggest a visualization concept of their own.

The questionnaire template handed out to survey participants can be found in Section 17.

6.2. Scratch-inspired design

The proposal of the scratch-inspired design takes scratch's imperative block style and converts it into the context of an expressional and declarative setup.

Functions, variables, parameters, etc., are portrayed as colorful blocks that can be dragged on top of each other.

6.2.1. Function Declaration, Composition and Application

Function declarations are displayed as red blocks with light-blue parameters below them, and the main expression above it, as shown in Figure 12.

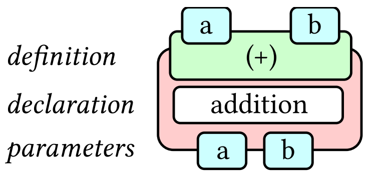


Figure 12: Example of scratch-inspired function definition for an addition function

Functions used in expressions are displayed as green blocks and can be applied by inserting blocks to the parameters declared above them. Their name is shown inside the block, whereas the alignment of the name (left, center, right) doesn't bear any semantic meaning.

6.2.2. Type Holes

If a parameter is left unapplied, a type hole is shown instead as a violet block with the expected value type as its name. Such a type hole can be seen in Figure 13.

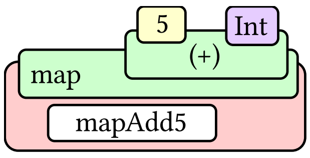


Figure 13: Example of scratch-inspired function definition with a type hole

In Figure 13, the second parameter of the `(+)` function is left unapplied, and thus, a type hole of type `Int` is shown.

6.2.3. Pattern Matching

Pattern matching is provided as a dedicated block that takes a value as a parameter and has the list of possible patterns as its pre-applied arguments. These cases then offer the matched patterns as values and can be supplied with blocks to specify the expression to be evaluated. An example of such a function can be seen in Figure 14.

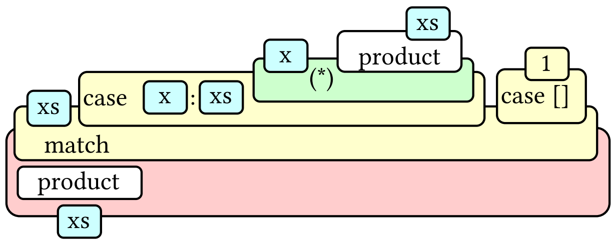


Figure 14: Example of scratch-inspired function definition with pattern matching

6.3. Flo-inspired design

This design proposal is inspired by flo (see Section 3.3). With this design, function elements are distributed on a canvas. Every element (e.g., a variable or function) can be connected to another element with arrows.

6.3.1. Function Parameter Editor

Parameters of a function are defined separately from the function body. A dialog, as depicted in Figure 15, appears next to the editor canvas when opening a function with the editor.

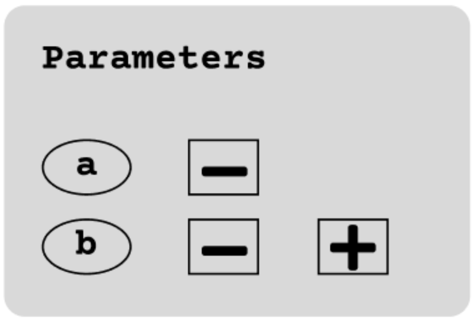


Figure 15: Draft of proposed function parameter editor

Users can add and name a parameter by clicking the plus sign. By clicking on the minus sign, a parameter can be removed again. The user can drag a parameter from the parameter editor onto the function editor canvas to use a parameter in the function body.

6.3.2. Function Editor

To define a function, the user can drag pre-defined functions, block elements, and self-defined functions from a sidebar onto the editor canvas. As described above, the same can be done with function parameters from the parameter editor. To connect a parameter to a function call, the user can create a connector-arrow between them. This is achieved by clicking on the parameter while holding Ctrl and then clicking on the function parameter slot.

To visually display currying, every function block has only one or no parameter. If a function has more than one parameter, the function block has dependent blocks for every additional parameter. The ":apply" suffix to the function's name recognizes such depending blocks. The last block of a function is the value returned by the function. This value can be used as a parameter for another function or marked as the function's return value.

The types of a function or variable block can be viewed by hovering over it.

An example of a simple function definition can be seen in Figure 16, a visual representation of the addition code scenario defined in Section 6.1.1.

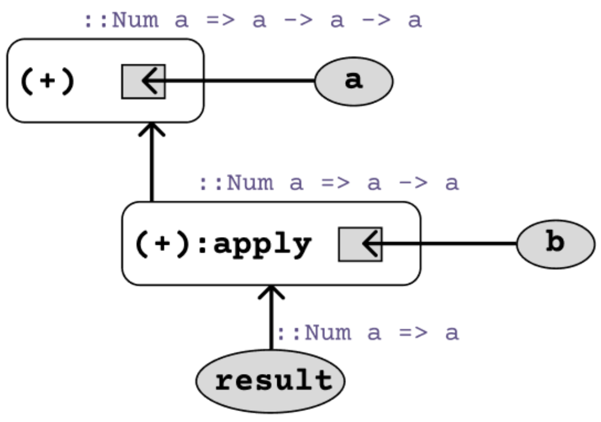


Figure 16: Example of flow-inspired function definition for an addition function

Functions can, of course, also be used as parameters themselves. For that, the user can create a connector arrow between a function block and a function parameter slot in the same way as with function parameters. When using functions as parameters, it is possible to leave some function parameters unapplied. Like this, a function parameter can be filled by the function it's used in.

An example of that can be seen in Figure 17, a visual representation of the "Map Add 5" code scenario in Section 6.1.1. The fill-color of the parameter slot recognizes the auto-filled parameter of the "(+)" function.

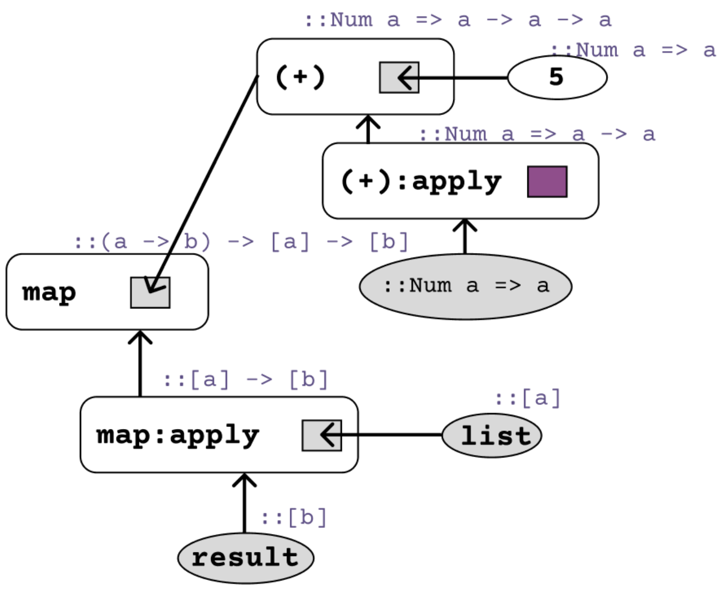


Figure 17: Example of functions used as parameters for other functions

Pattern-matching is a handy feature of Haskell. To support that in VisualFP, there is a pre-defined match block with match cases for typical scenarios (e.g. empty list and head-tail pattern). The match block has connector slots for each match case to which the user can connect the definition of the case behavior.

Another essential concept in any language is recursion. To create the recursive behavior, the user can drag the function they are defining from the sidebar onto the function editor canvas and use it as any other function.

Figure 18, a visual representation of the product code scenario defined in Section 6.1.1, shows how a recursive function definition using pattern-matching could look like.

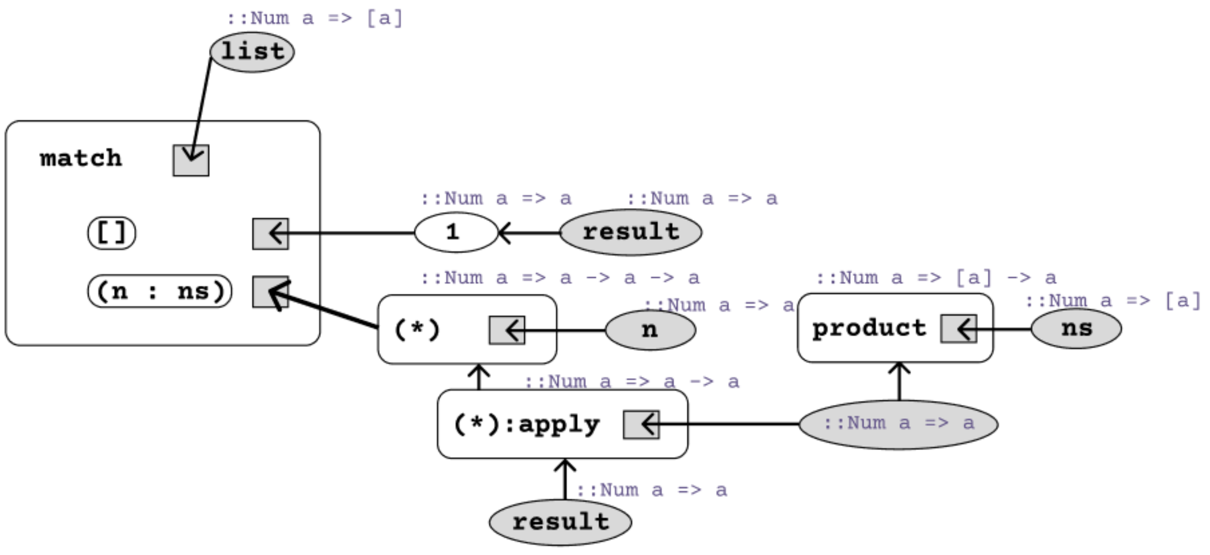


Figure 18: Example of a flow-inspired recursive function

6.4. Haskell function notation-inspired design

The Haskell function notation-inspired design takes the notation of functions in Haskell and converts the explicit and implicit parenthesis into blocks. The mapping from argument to value through an arrow remains the same. A simple addition definition can be seen in Figure 19.

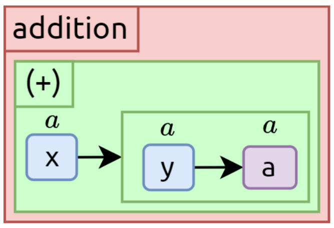


Figure 19: Example of Haskell function-notation inspired function definition

In order to bring more clarity to the definition, the blocks and values are annotated with their type.

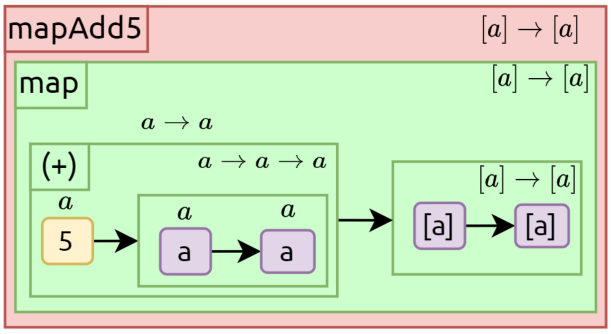


Figure 20: Example of Haskell function-notation inspired function definition with higher order function

The handling of type arguments is not yet defined, although a similar solution as for the scratch inspired-design described in Section 6.2, could be used.

6.5. Conclusions

Based on the questionnaire answers for the first three designs, which can be seen in Section 18, it can be concluded that each design received valid criticism.

6.5.1. Flo-inspired Design

It is clear that the Flo-inspired design received the most negative feedback. While this concept keeps blocks small due to blocks being connected through arrows, it quickly looks overloaded. This is partly because the type annotations on the questionnaire images are visible for every block (these are only supposed to be visible when hovering over a block). Still, the main reason is the currying visualization.

By trying to visualize currying, the amount of blocks grows with every additional function parameter, leading to a high total number of blocks. This problem could be improved by displaying functions as one block with multiple parameters, giving up the visualization of currying. As suggested by Rafael Das Gupta, an option could also be to offer the user the possibility to activate/deactivate currying in the function editor.

6.5.2. Scratch-inspired Design

The Scratch-inspired design received the most positive feedback but also some negative feedback.

The most consistent criticism was about the operator functions being aligned in the middle, implying that the design accounts for infix application but was written in parentheses (e.g., `(+)`), which in Haskell is a form of regular function application of operators that would support an infix notation. This is a very valid point and something the design wasn't supposed to imply. It is fairly easy to fix by aligning the operators to the left.

Another point of critique repeated by several participants was that unapplied parameters are not used correctly and/or inconsistently and some felt that the type holes were unintuitive.

Finally, all participants agreed that the design wouldn't scale, as it requires a lot of horizontal space to grow.

Compared to the other designs, the Scratch-inspired design was praised for its extensive use of coloring, and as the most easy to understand.

6.5.3. Haskell Function Notation-inspired Design

The Haskell function notation-inspired design received the most mixed feedback.

A common point of critique was that the design was too overloaded and difficult to understand, especially for beginners.

But some participants like the design the most, as it is the most similar to Haskell. Other participants disliked the design the most, maybe because the relationship between the design and Haskell was not clear enough. Still, it isn't a good sign if one needs to understand Haskell to understand the concept, as that goes against the goals of VisualFP.

7. Second Design Iteration

After the initial proposals received mixed feedback in the survey, project advisor Prof. Dr. Farhad D. Mehta suggested an additional design, which combines some aspects of the previous proposals and adds some new ideas.

Below, this suggestion is described in detail.

7.1. Final Design Proposal

The basic structure consists of nested blocks, each representing a different expression. In that regard, it is similar to the scratch-inspired design as described in Section 6.2, with the difference that the blocks are completely enclosing their children, as can be seen in Figure 21.

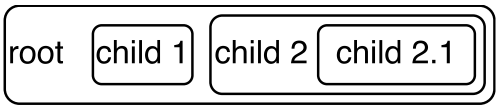


Figure 21: Proposal 2 - basic structure

Another similarity to the scratch-inspired design is the use of type holes for parameters that are not yet supplied. In such a case, a placeholder with nothing but the type of the parameter is shown, as can be seen in Figure 22.

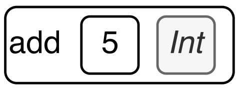


Figure 22: Proposal 2 - type hole

The main difference to the previous proposals is how scoping is solved: Instead of providing specialized expressions for constructs such as pattern matching, list comprehension, etc., the idea is to do scoping using only basic structures of functional programming such as lambda expressions. An example of that can be seen in Figure 23.

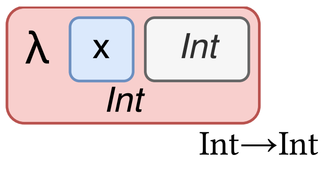


Figure 23: Proposal 2 - lambda

Regarding the re-use of expressions, the idea is to define multiple small functions and then stick them together rather than providing a `let ... in ...`-like expression to declare re-usable values.

7.1.1. Function Application

For function application, there are two possible application styles up for discussion:

Explicit Leave higher-order function values as such and apply them explicitly using a dedicated application function.

Elaborate Embedd a deeper understanding of application into the language, which allows to resolve nested curried function values to their arguments if necessary.

A side-by-side comparison of how double application of two `5` literals to an `addition` function would look like in both styles can be seen in Figure 24 and Figure 25.

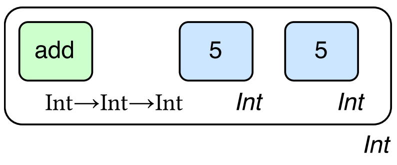


Figure 24: Elaborate application

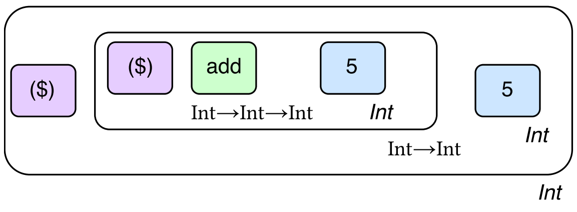


Figure 25: Explicit application

Of these two styles, the elaborate application style was chosen over explicit application since the elaborate style stays readable when scaling up to more extensive examples, while the explicit style would start to feel overloaded more quickly.

The type resolution for the elaborate application style works like this:

1. A type hole of a value $A_1 \to \ldots \to A_n$ is encountered.
2. A value of type $B_1 \to \ldots \to B_n \to A_1 \to \ldots \to A_n$ is inserted into the type hole,
3. The editor resolves the curried function into its nested values and matches the ending values $A_1 \to \ldots \to A_n$ with the expected type of the hole.

   The remaining arguments $B_1 \to \ldots \to B_n$ are then processed as new type holes to be filled in.

An example of the elaborate application system can be seen in Table 1.

Type hole	Inserted Value	Result
$A \to A$	A	*error*
A	$A \to A$	*new type hole: A*
$B \to C$	$A \to B \to C$	*new type hole: A*
C	$A \to B \to C$	*new type holes: A, B*

Table 1: Examples of elaborate application resolution

7.1.2. Sum Type Destruction

Sum types consist of a set of constructors, each with a different type. The type of a sum type is the union of the types of its constructors.

To work with a value of a sum type, it must be possible to destruct the value into its constructor and arguments. Since while developing one usually cannot know which constructor was used to create a value, all possible constructors must be handled.

A common approach in functional languages is to use *pattern matching*. Pattern matching allows to match values against a set of patterns and execute user-provided code per pattern. Each code path needs to have the same return type, which will then be used as the type of the matching expression. One of the pattern types usually destructs sum type constructors.

An example of pattern-matching in Haskell can be seen in Listing 5, which defines a sum type named `Expression` with two constructors, `Value` and `Addition`, and a function `calc` that pattern matches a value of type `Expression` against its constructors.

```haskell
data Expression where
  Value :: Int -> Expression
  Addition :: Expression -> Expression -> Expression

calc :: Expression -> Int
calc e = case e of
  Value v -> v
  Addition left right -> calc left + calc right
```

Listing 5: Example of pattern-matching in Haskell

An easier way to implement sum type destruction is to generate a function that takes a function parameter per constructor of the sum type. These parameter functions take the type constructor arguments as input parameters and map them to a common output type. These kinds of functions are called destruction functions. Listing 6 shows what such a destruction function would look like for the abovementioned example.

```haskell
data Expression where
  Value :: Int -> Expression
  Addition :: Expression -> Expression -> Expression

destruct :: Expression -> (Int -> b) -> (Expression -> Expression -> b) -> b
destruct = -- implementation omitted

calc :: Expression -> Int
calc e = destruct e (\x -> x) (\l r -> calc l + calc r)
```

Listing 6: Example of a destruction function in Haskell

Unfortunately, such a destruction function is not as powerful as pattern-matching in a few ways:

- It is not possible to specify multiple overlapping patterns, which are matched against in order of definition.
- It is not possible to combine patterns into more complex patterns.
- It is not possible to specify a default case.

This list is not exhaustive, but it already shows how the lack of pattern-matching would make the import of Haskell code into VisualFP more difficult.

How pattern-matching and destruction functions would look in VisualFP can be seen in Figure 26 and Figure 27.

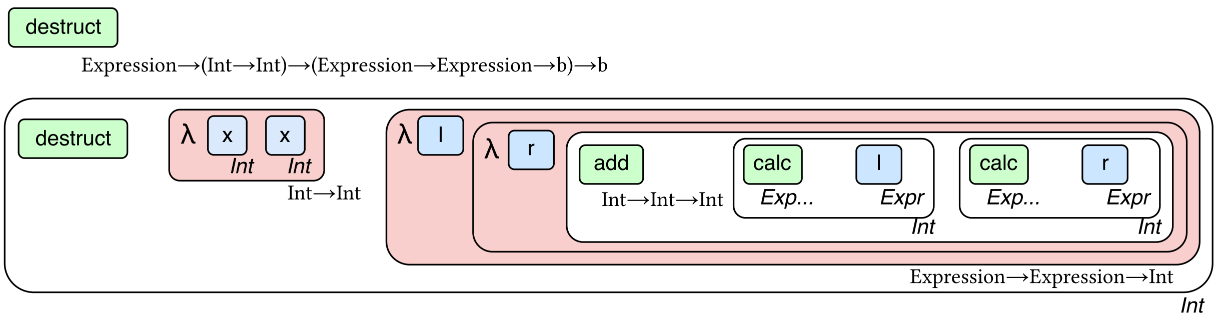


Figure 26: Example of a destruction function in VisualFP

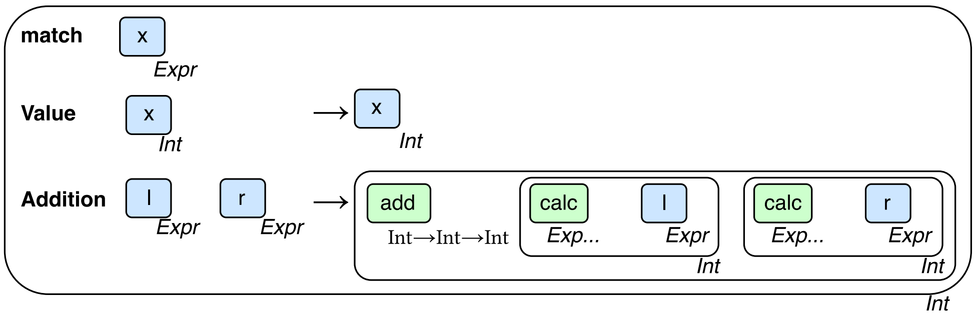


Figure 27: Example of pattern-matching in VisualFP

The decision was made in favor of the destruction function since the pattern-matching approach does not translate as well into an exclusively visual language. The fact that it makes the import of Haskell code more difficult is unfortunate but acceptable since the primary goal of VisualFP is to be a visual language to introduce beginners to functional programming.

7.2. Conclusion

During the first iteration, three very different approaches were explored. Valuable feedback was gained on their advantages and shortcomings through a survey. Prof. Dr. Farhad Mehta proposed a new design concept in the second iteration based on that feedback and some new ideas.

This new concept has the potential to provide beginners with more guidance during function composition than the previous concepts, which is why it is implemented in a PoC.

Part III - Proof of Concept

This part describes the development of the proof of concept application as described in the goals defined in Section 2.

8. Scope

The goal of the PoC application is to show that the visual concept described in Section 7 works.

To show that, the PoC needs to implement
- UC1 - Simple function composition
- UC3 - Function composition using lists
- UC4 - Function composition using higher-order functions
- UC5 - Curried Functions
- UC6 - Recursive function composition

as described in Section 4.2.

Although all other described use cases are integral to a fully functioning visual editor for functional programming as well, they are not included in the PoC due to time constraints.

9. Implementation Options

Different implementation options were considered for the PoC. The following sections describe the options that were considered and choices that have been made.

9.1. Deployment platform

For VisualFP, two possible deployment platforms were considered: A web application or a desktop application.

9.1.1. Desktop Application

A desktop application can be installed on the user's device with one installer. Both the frontend and backend are executed on the user's device.

This means that there is no special infrastructure required to host the application. The application can also access the local file system and other development tools such as Cabal[1].

Unfortunately, every user would need to install the application themselves. The application also would need to be built separately per operating system and depending on the chosen runtime also per processor architecture.

9.1.2. Web Application

A web application has a frontend and a backend. The frontend is statically served to the user in a web browser and communicates through an API with the backend, which is hosted on a server.

[1]https://www.haskell.org/cabal/

The advantages are that the frontend and backend can be written in entirely different programming languages, that the users don't have to install anything on their devices, and that there is no need to support multiple operating systems explicitly since most devices can run a web browser.

The drawbacks are that the application must be hosted on a server and other development tools that may require access to the user's device cannot be used.

9.1.3. Deployment Decision

VisualFP targets students, so it will primarily be used in classrooms. Such environments often do not allow for easy deployment and maintenance of application servers. Thus, the decision has been made to develop VisualFP as a desktop application.

9.2. UI Frameworks

VisualFP requires external technologies to implement a UI, especially to achieve cross-platform support without duplication. The following technologies were considered.

9.2.1. Electron.js

Electron.js is a framework for creating desktop apps using HTML (HyperText Markup Language), JS (JavaScript), and CSS (Cascading Style Sheets), implemented by combining chromium and Node.js.

Electron allows the creation of desktop apps in web-app style, which means that, as long as no native OS features are needed, an app is automatically capable of running on different OS platforms without adjustments. [8]

For VisualFP, Electron.js can implement the UI and the backend logic.

Since Electron apps can be created like any web app, many libraries are available for frontend development. This also allows the combination with any other UI framework that targets the web.

The authors already have experience with JavaScript and its ecosystem.

Electron, like any browser, requires the application to be written in JavaScript. JavaScript is an interpreted language and has a weak type system. It is also not considered a primarily functional language, as it offers many imperative features and APIs.

Considering that both a Node.js and a Chromium runtime are bundled in Electron, the resulting app sizes are quite large, even for apps with little logic. Rendering the app in a browser also requires much more resources than UIs built with native OS APIs (e.g., Win32 on Windows)

9.2.2. Haskell-gi

GTK (GIMP ToolKit) is a widget toolkit that allows the creation of UIs that work on many popular operating systems. GTK widgets can be created programmatically or with an XML UI definition. [9]

Haskell-gi is a Haskell library that offers bindings to GTK. The library allows widget creation in a rather imperative style.

There also is a library that supports widget creation in a functional style called 'gi-gtk-declarative'[1], but that library is still experimental. [10]

Using Haskell-gi to create the UI of VisualFP would allow the frontend and backend logic to be implemented in Haskell.

The downside to Haskell-gi is that GTK is unknown to both authors and it doesn't seem to be used as much as web-based frameworks, which could impact the availability of documentation and examples.

9.2.3. Threepenny-GUI

The Threepenny-GUI framework was written in Haskell to create desktop applications that run in a web browser. The framework can be combined with Electron for a tighter integration with the desktop environment.

Threepenny starts a local web server written in Haskell that serves a HTML page. Then, the server establishes a WebSocket connection to communicate with the browser. Finally, using a JavaScript FFI (Foreign Functional Interface), Threepenny sends JavaScript code via this connection to execute it on the client.

Threepenny also offers the possibility of implementing the application in FRP (Functional Reactive Programming) style [11].

Using Haskell on both the front and backend would be an advantage over other frameworks. Threepenny's FRP specific functions should allow for a good architecture.

It is to be noted that Threepenny is still in an experimental phase, according to the package author [12]. Also, neither author is familiar with the technology.

9.2.4. GHCJS

There are many other Haskell libraries for UI implementations, but many rely on compiling Haskell to JavaScript. This cross-compilation is often based on GHCJS. GHCJS implements a JS backend for GHC (Glasgow Haskell Compiler) and has recently been merged into the GHC repository [13].

Using GHCJS with an accompanying UI library, which would optimally support FRP, could make a lot of sense for an application like VisualFP. Since all code would be compiled to JavaScript, it would automatically be platform-independent without the need for a supporting server infrastructure other than a way of serving static files.

But GHCJS has merged only recently [13], and thus its usage poses some challenges:

- No pre-built binaries are available at the time of writing, meaning the complete GHC compiler must be built manually.
- While there is some documentation, it doesn't seem to be very comprehensive.

The downsides could be overcome, and it is to be expected that GHCJS will get better tooling support in the future.

[1]https://github.com/owickstrom/gi-gtk-declarative

But after writing some samples in GHCJS, it is to be expected that a considerable amount of time would need to be invested to get GHCJS to work for the PoC.

9.2.5. Bolero

Bolero is an F# wrapper around Blazor[1], leveraging different libraries to create web applications. With Blazor, web applications can run both on the server side through web-sockets or on the client side through WebAssembly.

While Beloro doesn't implement FRP, it has a Model-View-Update architecture, clearly separating UI from business logic in a reactive way. In addition to defining the web page structure from F# code, Bolero also offers the possibility to use HTML templates with “holes” as placeholders for page values, event listeners, etc. [14]

The clear separation of the UI and business logic and the support of the established .NET platform is a significant advantage for Bolero.

The two downsides are that neither of the authors is familiar with the framework and that F# doesn't really fit into a project, that aims to create a Haskell-compatible visual editor.

9.2.6. UI Framework Decision

All technologies to implement UIs in a functional language are unfamiliar to the authors.

Because of that, it is crucial to have a good abstraction between UI and business logic in place so that the UI framework, if necessary, can be exchanged with another option with as little effort as possible.

FRP is an excellent concept for incorporating UI behavior and events into functional programming. Of the frameworks considered, only threepenny-gui includes a library that supports FRP. Despite not following the FRP concepts, Bolero also supports reactive handling of UI contents.

Since the project intends to create a tool for visual programming in Haskell, it also makes sense to implement the UI of VisualFP in Haskell. For that reason, Threepenny will be used for the PoC.

As already mentioned in Section 9.2.4, using GHCJS as a Haskell compiler that targets JavaScript would enable other interesting possibilities but has been deemed too time-consuming for the PoC.

9.3. Compiler Platform

A compiler platform is a set of tools and libraries that can be used to compile code. VisualFP requires a compiler platform to build the application itself but also as a library to compile programs created by the users in the visual editor.

These two use cases pose different requirements on the compiler platform:

- To be used as a compiler, the platform must have a set of support tooling available, such as a build tool and a language server.

[1]https://dotnet.microsoft.com/en-us/apps/aspnet/web-apps/blazor

- To be used as a library, the compiler platform must offer a well-defined API that can be used to create, parse, and compile an AST (Abstract Syntax Tree) programmatically.

For these two use cases, two different compiler platforms may be used.

The compiler used to build the PoC itself depends on the language and other frameworks chosen to implement the application. This section focuses on the use of a compiler platform as a library.

9.3.1. GHC

The most obvious choice for a Haskell compiler platform is GHC[1]. In combination with build tooling, such as Cabal[2] and the language server HLS[3], it provides a great development experience.

Unfortunately, the usage of GHC as a library is not as straightforward as using it as a compiler:

- Given the large amount of supported features, the API is more complex than necessary for the use-case of representing a visual editor code snippet in an AST.
- The API documentation is limited. Even though there is a great starting point for the internals of GHC available at GHC commentary [15], and some bloggers such as Stephen Diehl took the time to write about the GHC API [16], much of the available documentation seems to be out-of-date, incomplete, or missing.

9.3.2. Other Haskell Compiler-Platforms

Outside of the ubiquitous GHC, described in Section 9.3.1, a few other Haskell compilers were considered for this project. The most notable ones are:

Hugs, which is a compiler that provides an almost complete implementation of Haskell 98 [17]. Unfortunately, Hugs is is not actively maintained anymore [17], thus, it wasn't considered further.

Another Haskell compiler platform is the *Haskell Suite*, which is a collection of tools and libraries that aim to implement a complete Haskell compiler [18]. The AST interpreter is provided on hackage as the package `haskell-src-exts`[4]. After creating some example programs with it, it seems that the API is quite nice to use. Unfortunately, the Haskell Suite is also not actively developed anymore, and is currently on maintenance support [19].

9.3.3. Custom Compiler Platform

Given that the PoC is simple and limited in its features, it might be an option to skip the use of a compiler platform as a library altogether:

- It is not necessary for the compiler platform to parse Haskell code. For the PoC, being able to type-check simple expressions is sufficient.
- Execution of expressions is optional as well.

[1]https://www.haskell.org/ghc/
[2]https://www.haskell.org/cabal/
[3]https://github.com/haskell/haskell-language-server
[4]https://hackage.haskell.org/package/haskell-src-exts

So, instead of aligning with a complex API of a full-blown compiler platform, a custom implementation tailored to the specific needs of the PoC could be created. Given the requirements, the implementation could even be reduced to just type-checking.

9.3.4. Compiler Platform Decision

Since VisualFP will rely heavily on the chosen platform, a platform change later down the road would be expensive.

GHC is the only still actively developed compiler platform out of the considered platforms. But after testing and research, the GHC API was deemed to complex, and not easy enough to integrate within the limited timeframe available for the PoC.

This is why VisualFP will implement a custom compiler platform. Given the low requirements, it could be considered an exaggeration to call such an implementation a compiler platform, which is why it'll be called an inference engine from now on.

10. Architecture

This chapter describes the architecture chosen for the PoC application.

10.1. Client/Server Cut

As described in Section 9.2.6, the PoC uses Electron.js and Threepenny-GUI for its implementation.

Using the client-server cut classification by Klaus Renzel [20], the technology choice results in the application having a "remote user interface" as shown in Figure 28.

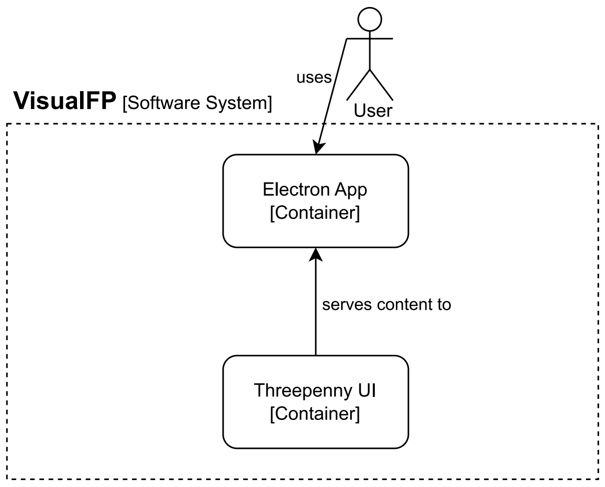


Figure 28: C4 Container Diagram for VisualFP PoC

The Threepenny UI starts a local web server from which it serves static files to the UI. The Electron.js app doesn't contain any logic and acts as a browser.

Theoretically, Threepenny could also host a regular web application, eliminating the need for an Electron app. However, Threepenny controls the browser via web sockets, so a performant server and a good network connection are required. It is recommended to avoid this, as a high latency connection would be noticeable through slow UI updates. [11]

10.2. Backend Components

Simon Brown suggests to use the C4 model to visualize the architecture of an application through diagrams on 4 levels [21]. The container diagram has been used to showcase the client/server cut in Figure 28, the component diagram of the backend is shown in Figure 29.

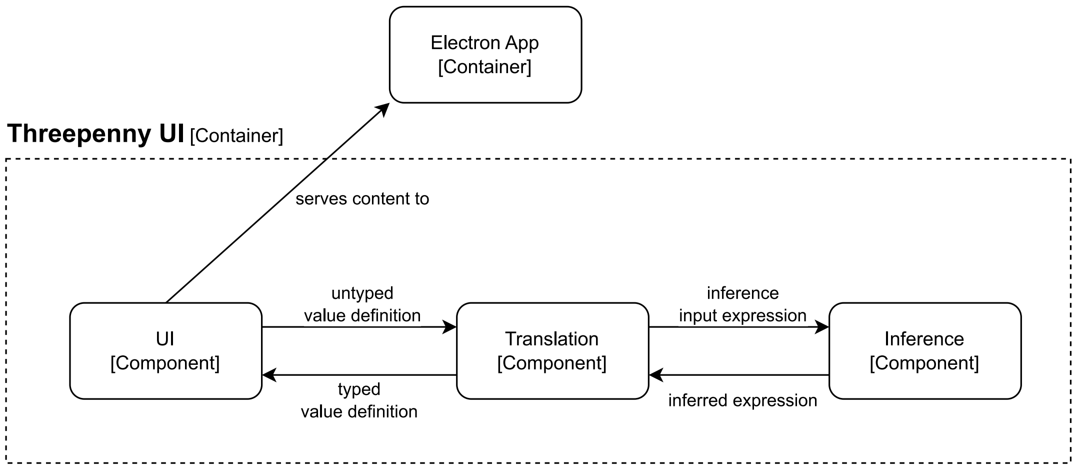


Figure 29: C4 Component Diagram for VisualFP's Threepenny UI

Splitting the backend into these three components isolates the UI from the business logic, making either of them easily replaceable.

The translation component is described in more detail in Section 11, the UI component in Section 12 and the inference engine in Section 13.

11. Translation Component

The translation component has been built between the UI and the inference engine. As can be seen in Figure 29, it is responsible for translating between the data models used by the UI and the type inference engine:

1. From the untyped model of the UI to the input model of the inference engine
2. From the inferred model of the inference engine to the typed UI model

This process isn't as straightforward as one might think, as the UI and inference engine have different representations of function application: The UI maintains a list of arguments per reference. In contrast, the inference engine expects application to happen in dedicated nodes. This discrepancy has also been described in Section 7.1.1.

Translating the inferred model into the typed UI model is quite simple: Applications are simplified into arguments of the underlying element. Translating untyped UI values, where the amount of arguments is unknown, into the input model of the inference engine requires a more elaborate process.

One approach might be to try out different numbers of nested applications and see if one might successfully type-check.

For VisualFP a different approach was chosen: In case the arguments of a reference need to be inferred, the UI model passes the original type hole along. The translation component then compares the arity of the type hole with the arity of the reference and adds as many applications as needed for the two to match.

12. User Interface

This section describes the features of the PoC application UI, the high-level implementation, and how functional reactive programming could be applied to VisualFP.

12.1. Features

The UI for the PoC application includes two main components, as shown in Figure 30: A sidebar with pre-defined value blocks and the function editor.

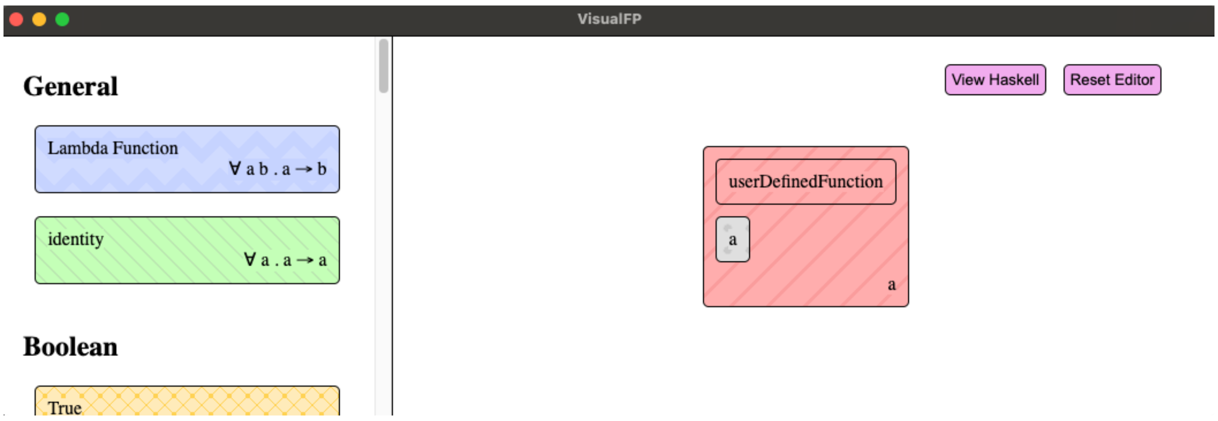


Figure 30: Undefined function value in the VisualFP UI

The PoC allows the construction of a value, the "userDefinedFunction", which starts with a generic type hole. Starting with a generic function type allows more flexible testing. In a completed application version, the user can define the function name and type when creating it.

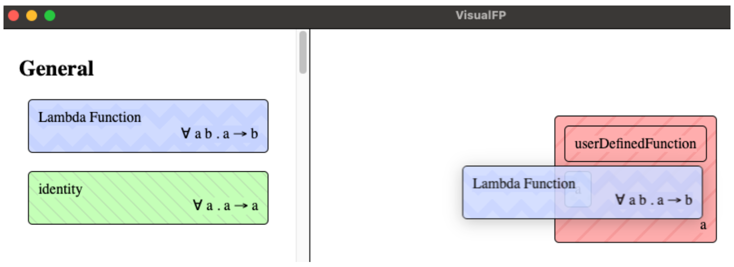


Figure 31: Dragging lambda block into value definition

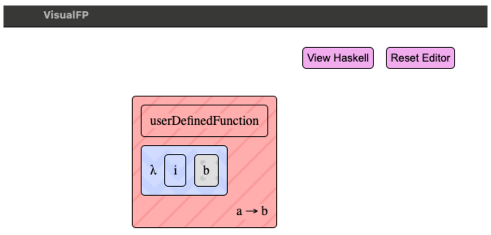
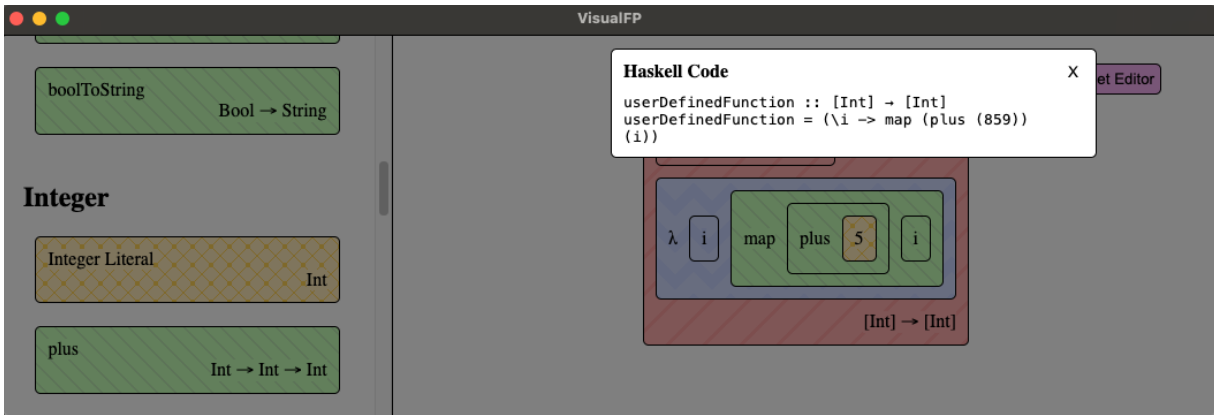


Figure 32: Updated function definition including a lambda block

Figure 31 and Figure 32 show how a lambda block is inserted into the value definition. To build the value definition, the user drags the lambda block from the sidebar into the type hole. The drop event then triggers the application to insert the lambda block into the function definition and infer the types of the new function definition. This process can be repeated with suiting value blocks until no type hole is left.

As the PoC is intended to test the concept, only a reset button exists to return to the initial empty definition. In a full version, this would be replaced with the possibility to remove specific blocks from the definition.

Finally, the user-built function definitions can be viewed as Haskell code by clicking the "View Haskell" button. Figure 33 shows the Haskell code for the `mapAdd5` function.



```
userDefinedFunction :: [Int] → [Int]
userDefinedFunction = (\i -> map (plus (859))
(i))
```

Figure 33: Haskell defintion of mapAdd5 function in VisualFP

12.2. Implementation

The UI implementation consists of an Electron.js app hosting a Threepenny UI. The Electron app is packaged with an executable of the Threepenny UI and all UI related static files, i.e. CSS & JavaScript files. When starting the Threepenny UI, the Electron app passes a usable port for the local web server and the file path of the static UI files to the Threepenny UI.

The function editor is the most significant part of the Threepenny UI and has two primary responsibilities:

- Rendering of function value blocks
- Reacting to value block drop events

The rendering part creates an HTML representation of each block in the value definition and annotates it with CSS classes according to its block type.

Reacting to the drop events is a bit more complicated. The block values in the application's sidebar carry their names as data transfer data. When the user drops a block value into a type hole, the data transfer data is included in the event data.

Unfortunately, the drop events cannot be registered when creating the type hole elements in the rendering part. So, to register the drop event listeners, the IDs of type holes need to be collected upfront. With these IDs, the HTML elements added to the DOM (Document Object Model) can be loaded, and the event handlers registered.

The drop event handlers always do the same, regardless of the block value that was dropped:

1. Replace the type hole with the dropped value
2. Infer the updated function definition
3. Clear all elements from the function editor
4. Render the inferred function definition

12.3. Functional Reactive Programming

Threepenny includes an FRP library, which follows the concepts described by Conal Elliott and Paul Hudak. FRP has two main concepts: Events and Behaviors. An Event is defined as a list of occurrences in time. A Behavior represents a value that changes over time. [22]

While the first intention was to build the PoC with an FRP architecture, it became clear over time that Threepenny's FRP library is not yet ready for more complex use-cases like

VisualFP's function editor. The main problem is that no function allows it to merge multiple events. Implementing the FRP architecture through Threepenny could be considered again once the FRP library is replaced by reactive-banana[1]. The author of Threepenny, Heinrich Apfelmus, plans to do that in a future release [23].

Generally, there is no reason why VisualFP couldn't be implemented using FRP. In such an implementation, there would be three kinds of events:

- "Reset Editor" button is clicked
- "View Haskell" button is clicked
- A block value is dropped into a type hole. This event combines all events from every type hole in the function definition.

The value definition of the user-defined function is a behavior that changes every time a block value is dropped into the value definition. When the value definition changes, the elements displayed in the function editor must also be updated.

13. Type Inference

VisualFP features a type-inference engine responsible for figuring out which expressions are valid, determine which type holes are necessary, and annotate sub-expressions with their according types.

It operates on a separate expression model to isolate the inference engine. The engine is not responsible for converting to and from the UI model; this is done by a translation layer described in Section 11.

Heavy inspiration for the implementation of the engine was taken from the talk "Type inference as constraint solving" by Simon Peyton Jones [1].

13.1. Phases

An overview of the inference engine is shown in Figure 34. It shows how the process is separated into 3 phases.

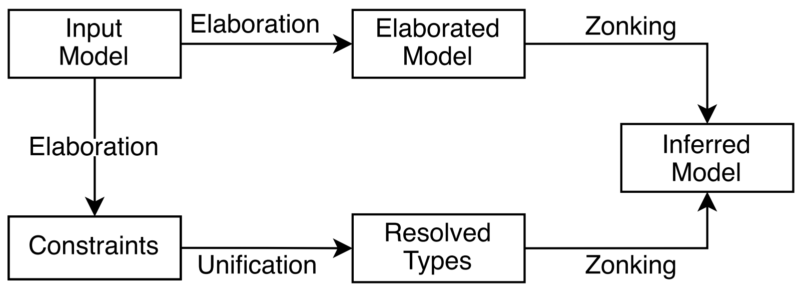


Figure 34: Type-inference engine components

1. **Elaboration**: The elaboration phase takes an input expression and annotates all elements with placeholder types called *unification variables*. Along the way, it collects these variables and creates a list of constraints for them. For example, if a value of type α is applied to a value of type β, then α must be a function taking β as the argument.

   A more detailed description of these constraints can be found in Section 13.2.

[1]https://github.com/HeinrichApfelmus/reactive-banana

2. **Unification**: Next, the unification algorithm tries to find a solution for the unification variables that satisfy all constraints. If it succeeds, a concrete type can be assigned to each unification variable.

   The algorithm is based on Prolog, as described by Prof. Dr. Farhad D. Mehta in his lecture [24].

3. **Zonking**: Using the elaborated expression, still filled with unification variables, and the results of the unification phase, the zonking phase inserts the concrete types into a new expression.

13.2. Constraint Language

The inference engine is based on what Simon Peyton Jones describes as “the French Approach” [1]. It has been described in the paper “The Essence of ML Type Inference” by Francois Pottier and Didier Rémy [25].

In such an engine, the constraints are essential. They contain all the knowledge gained through the elaboration pass of the input and can be used by the unification algorithm to sort out all types. They are also what differentiates the “French approach” from a classical Dalmas-Milner inference: The solving of constraints is deferred, as opposed to being solved in one pass [1].

The constraint language describes the structure of constraints. Simon Peyton Jones’ implementation [1] inspires the constraint language used by VisualFP with a few adjustments.

- There is no implication constraint.
- Constant, constructed, and unification types are summarized as a single sum type.
- Conjunctions are represented as lists of constraints instead of combinations of two to form a tree. As a flat list, there is also no need for an empty constraint.

Part IV - Results & Outlook

14. Results

This section evaluates the results of the project, including a review of which of the initially stated requirements have been implemented, how the concept and the PoC application turned out, and a small demonstration of the PoC application.

14.1. Requirement Validation

Section 4 and Section 5 defined 9 use cases and 2 NFRs.

Table 2 shows which of these requirements were fulfilled during the project and which are still open.

ID	Requirement	Result
UC1	Simple Function Composition	Achieved in PoC
UC2	Function Execution	This requirement is considered as out-of-scope for this project.
UC3	Recursive Function Composition	Achieved in PoC
UC4	Function Composition using Higher-Order Functions	Achieved in PoC
UC5	Curried Functions	Achieved in PoC
UC6	Function Composition using Lists	Achieved in PoC
UC7	Data Type Composition	This requirement is considered as out-of-scope for this project.
UC8	Save Source File	This requirement is considered as out-of-scope for this project.
UC9	Open Source File	This requirement is considered as out-of-scope for this project.
NFR1	Platform Compatibility	The PoC application can be executed on all target platforms. The app's compatibility could be improved by switching to a GHCJS-based UI technology.
NFR2	Learnability	This requirement was achieved with some notes. A more detailed explanation can be found in Section 14.1.1

Table 2: Requirement verification

The project's main focus was to create a visual concept for function composition and prove that it is feasible with a PoC application. As can be seen in Table 2, all requirements related to function composition have been achieved.

Due to time constraints, the requirements UC2, UC7, UC8 and UC9 had to be considered as out-of-scope. However, during the implementation of the type inference engine, UC7 was kept in mind so that custom data types could be added without much effort.

14.1.1. Validation of NFR2

The non-functional requirement NFR2, as described in Section 5.2, states that a user that isn't familiar with functional programming should be able to use VisualFP within 1 hour.

This requirement was validated by showcasing the PoC to Samuel Bernhard, a software engineer at Hamilton Bonaduz AG. Samuel is a seasoned software developer but isn't familiar with functional programming.

He was able to use VisualFP within 1 hour, so we deem the requirement to be achieved.

Still, the trial pointed out some aspects that are worth to be noted for future development on the project:

- As he wasn't familiar with the function notation used by Haskell, he wasn't able to make use of type hints (type-holes, function signatures, etc) without additional explanation.
- He also wasn't familiar with the `cons` and `nil` construction of lists, additional explanation was necessary before he could use it.
- He would have liked to execute his created values.
- After the showcase, the envisioned outlooks were presented to him. He liked the option of the visual and textual language (Section 15.2) very much.

These shortcomings could be fixed by adding a getting-started tour for beginners, which would explain the different UI components, provide a tutorial for value construction, and explain Haskell's function type notation. In addition, block values provided by VisualFP could feature a brief description.

14.2. Design Concept

The design concept described in Section 7 focuses on visual function composition. It proposes a function editor that features all the basic functionalities needed to create functional applications in a block-based fashion.

The editor allows learners to approach functional programming through blocks instead of code. The type-inference engine offers much guidance, especially the automatically generated type holes, which can help to understand how e.g. parametric polymorphism or currying works.

But other aspects not covered in the concept are also essential to create modern functional programs. For example, a robust type system with support for sum types, type classes, etc., can be found in most current functional programming languages.

Given that the project's goal is to allow beginners to approach functional programming more easily, the concept should also address a clear transition to code-based programming. One possibility to integrate this idea has been described in a possible outlook in Section 15.2.

14.3. Proof of Concept

The project produced a proof of concept application that implements the design concept as described in Section 7. Although the application doesn't offer visualizations for some of the most common aspects of functional programming, such as the construction of custom data types, the application shows that the proposed concept for function composition works and is easy to use. So, in the author's opinion, the project's main goal was achieved.

To make the application ready for use in a classroom, the use cases that weren't achieved during this project, as shown in Section 14.1, need to be implemented.

In addition, there is some potential for improvement in the current implementation of the PoC. The choice for Threepenny as the UI framework was made primarily due to the given time constraints and the expectation that Threepenny allows for fast progress while implementing the PoC, which proved to be true. But a UI technology that doesn't require a local web server, probably a GHCJS-based framework, would be better suited to implement a full version of VisualFP, as it would enable the application to be served to any browser as a set of static files. A different UI framework may also provide better support for functional reactive programming, which is expected to make the UI implementation more concise.

14.3.1. macOS Electron App

For the project submission, the PoC Electron application was packaged for Windows, Linux & macOS. Unfortunately, the macOS app has an unforeseen issue: It doesn't pass the macOS gatekeeper checks.

Based on code signature, notarization, and comparison with known malware, gatekeeper flags potentially dangerous applications and restricts them from execution. [26]

In the case of VisualFP, the error message ""VFP.app" is damaged and can't be opened. You should move it to the Bin." appears when trying to execute the application. Gatekeeper can be bypassed using the command `xattr -c VFP.app`, with "VFP.app" being the application name, to execute the application anyway. A future project would need to address this issue to make the application usable for a broader audience.

14.4. UI Demonstration

Figure 35 through Figure 40 depict a step-by-step construction of the `mapAdd5` code scenario, described in Section 6.1.1, using the PoC application.

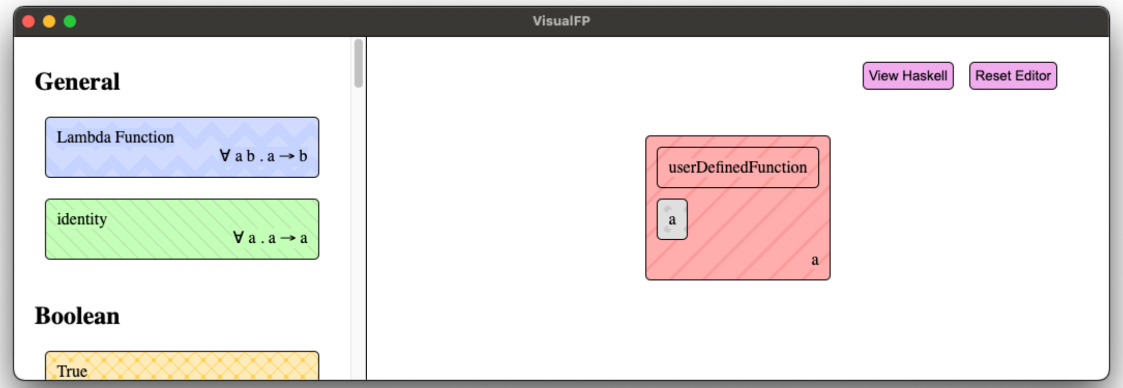


Figure 35: Step by step demonstration of mapAdd5 construction - Part 1

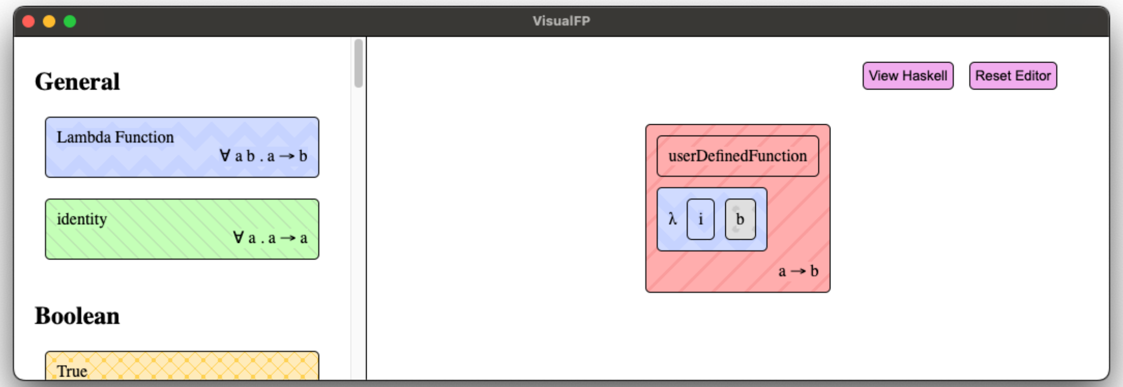


Figure 36: Step by step demonstration of mapAdd5 construction - Part 2

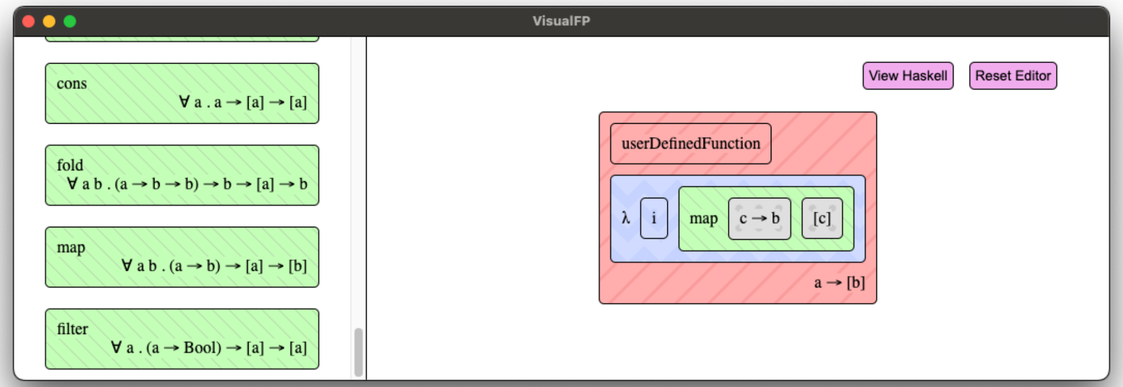


Figure 37: Step by step demonstration of mapAdd5 construction - Part 3

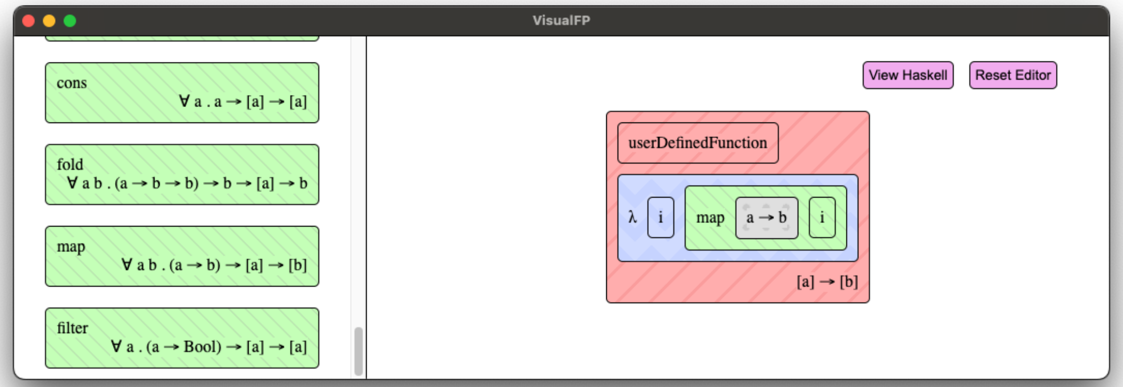


Figure 38: Step by step demonstration of mapAdd5 construction - Part 4

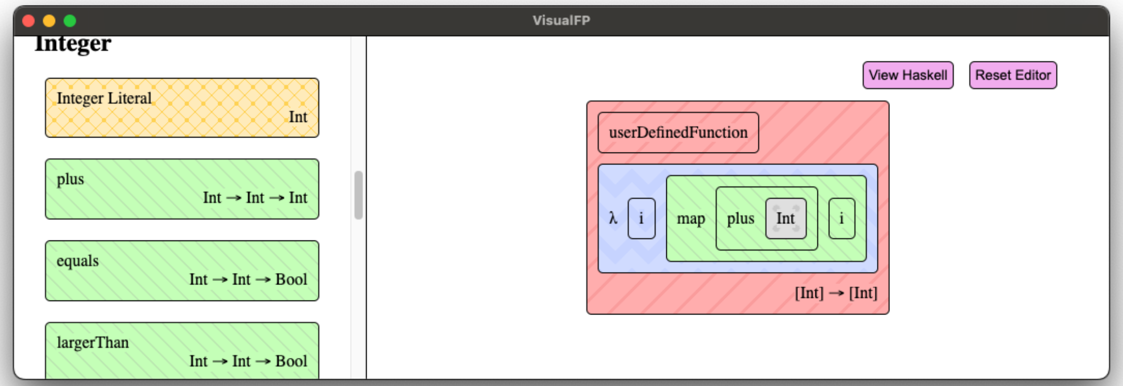


Figure 39: Step by step demonstration of mapAdd5 construction - Part 5

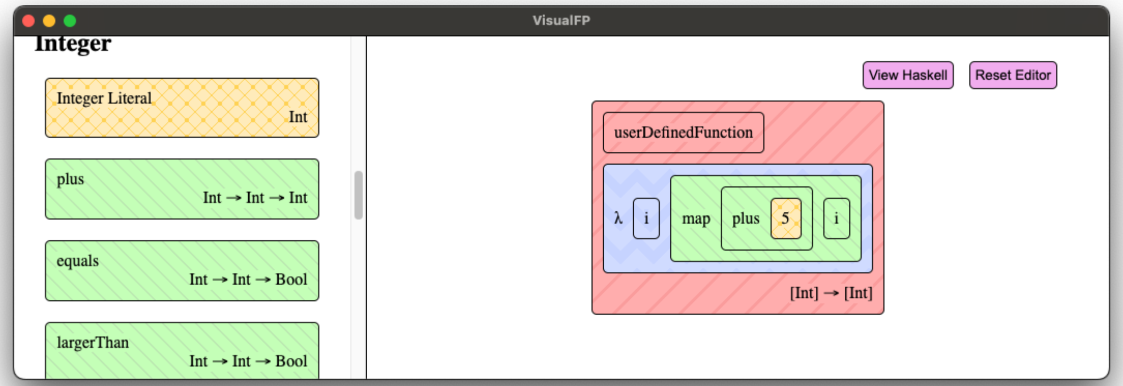


Figure 40: Step by step demonstration of mapAdd5 construction - Part 6

15. Outlook

During the PoC development, two possible variants of a feature-complete application emerged. Both are briefly described below.

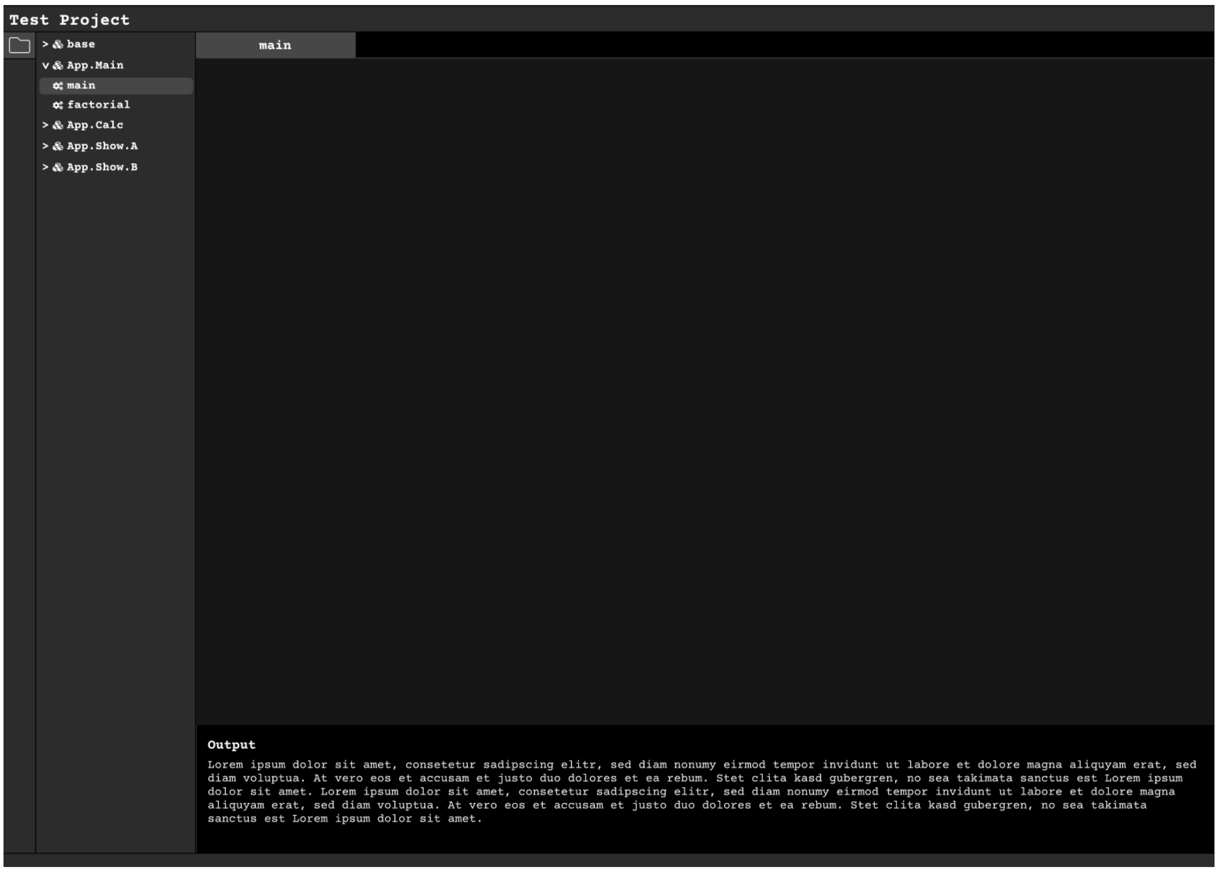
15.1. IDE for visual functional programming

In a future project, VisualFP could be further enhanced into a visual IDE for Haskell.

Besides the functionality already implemented in the PoC application, additional features offered by Haskell would also need to be implemented visually, such as an editor for Haskell-type definitions, a way to organize user definitions into modules, integration with a package manager, etc.

On the other hand, such a development would offer the opportunity to outsource functionality, such as compilation and execution, to Haskell tooling. A Haskell compiler could also replace the already-built type-inference engine, and an integration with, e.g., the Haskell Substitution Stepper[1] could be helpful as well.

Figure 41 shows a mockup of what such an IDE could look like. The two main elements are the sidebar, which gives the user access to libraries and self-defined functions, and the large editor section containing the visual editor.



```
Output
Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed
diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum
dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna
aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata
sanctus est Lorem ipsum dolor sit amet.
```

Figure 41: Mockup of VisualFP IDE

If interpreted as a learning tool, the VisualFP IDE would offer a clear learning path from dragging visual blocks to writing Haskell code. Learners could first build an understanding of functional concepts visually and then apply the same understanding to Haskell code.

Haskell programmers could find use in such a tool as well. Specifically, debugging tools could offer an advantage over a pure textual development approach. Visual representations

[1]https://eprints.ost.ch/id/eprint/991/

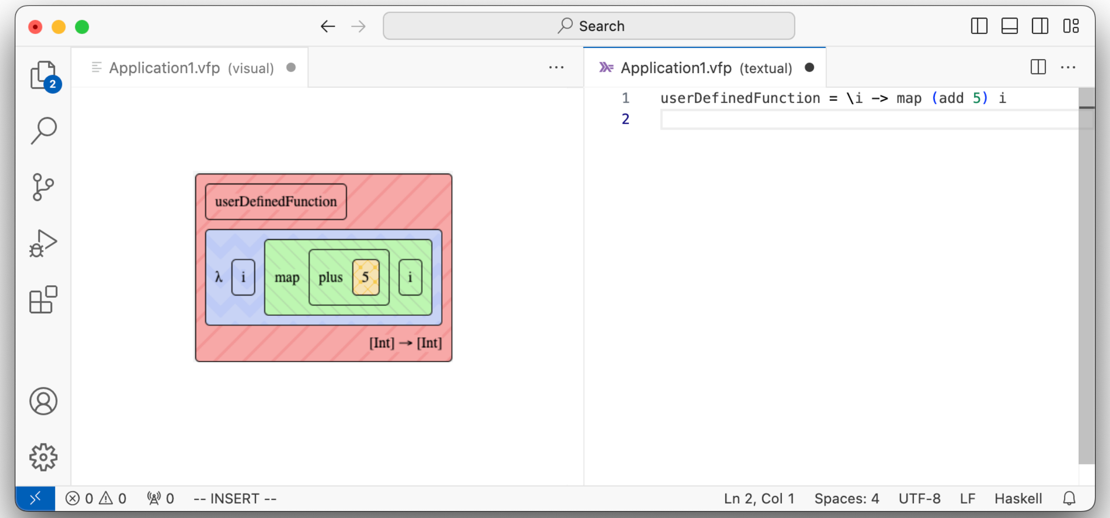
would be equivalent to textual code, so VisualFP could be considered a new tool in the toolbox of seasoned Haskell programmers.

Full compatibility with a typical Haskell project would not be a trivial goal. A deep integration with GHC, Cabal, etc., would be necessary. It would bind VisualFP exclusively to Haskell, support for other functional languages, such as F# or Scala, could not be easily added afterward.

15.2. Visual and Textual Language

The function editor shown in the PoC application could be extended into a language that can be viewed, edited, and executed in both a visual and textual fashion simultaneously.

One could imagine a dedicated application for such an environment, but a web application or an extension for existing editors could also be envisioned. A Visual Studio Code extension mockup can be seen in Figure 42 to illustrate the idea.



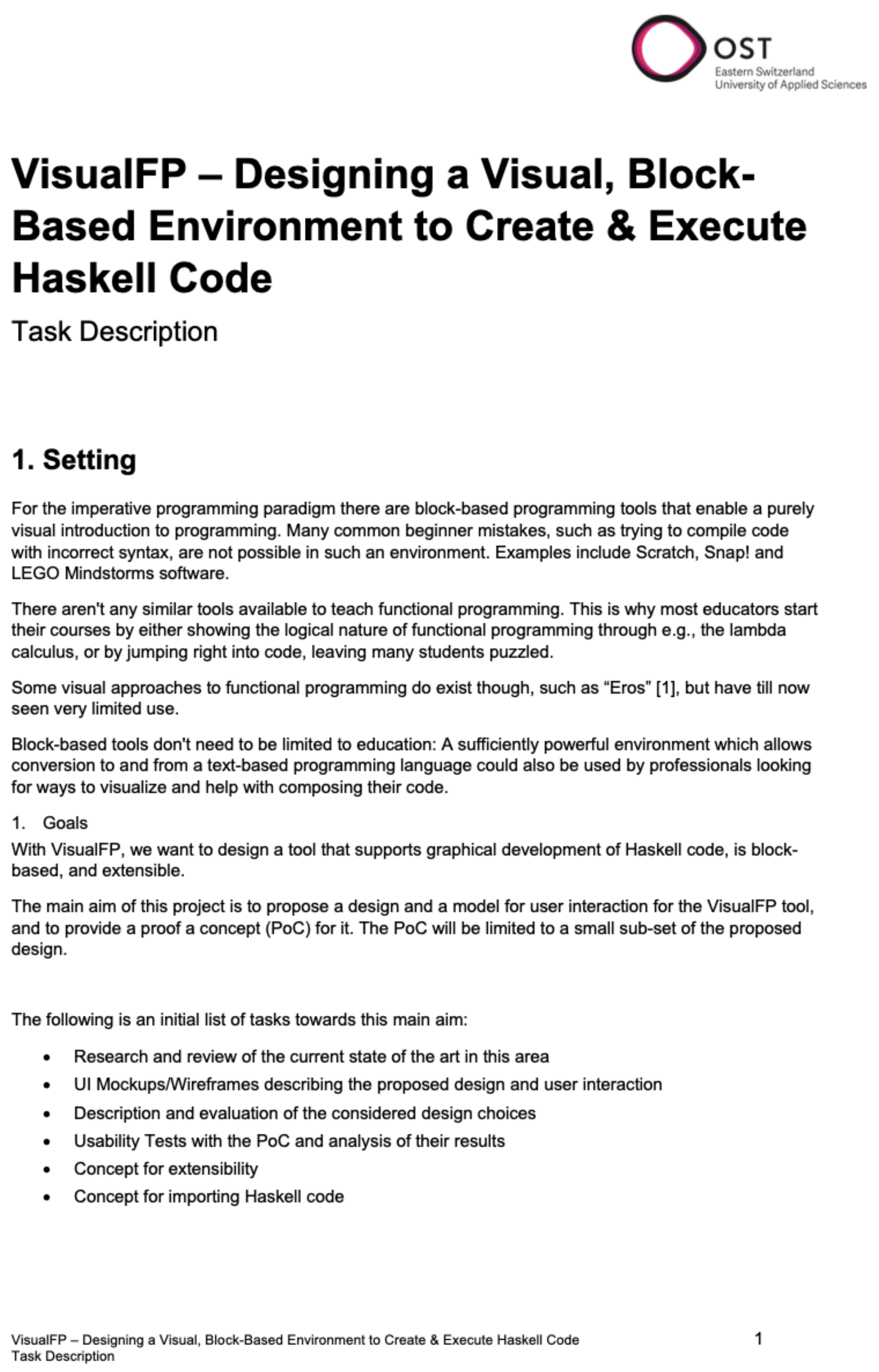
```haskell
userDefinedFunction = \i -> map (add 5) i
```

Figure 42: Visual Studio Code Concept

The advantage of such an approach is that learners can build an understanding of functional concepts graphically and then have a clear learning path into code.

Part V - Appendix

16. Task Description



OST
Eastern Switzerland
University of Applied Sciences

VisualFP – Designing a Visual, Block-Based Environment to Create & Execute Haskell Code

Task Description

1. Setting

For the imperative programming paradigm there are block-based programming tools that enable a purely visual introduction to programming. Many common beginner mistakes, such as trying to compile code with incorrect syntax, are not possible in such an environment. Examples include Scratch, Snap! and LEGO Mindstorms software.

There aren't any similar tools available to teach functional programming. This is why most educators start their courses by either showing the logical nature of functional programming through e.g., the lambda calculus, or by jumping right into code, leaving many students puzzled.

Some visual approaches to functional programming do exist though, such as "Eros" [1], but have till now seen very limited use.

Block-based tools don't need to be limited to education: A sufficiently powerful environment which allows conversion to and from a text-based programming language could also be used by professionals looking for ways to visualize and help with composing their code.

1. Goals

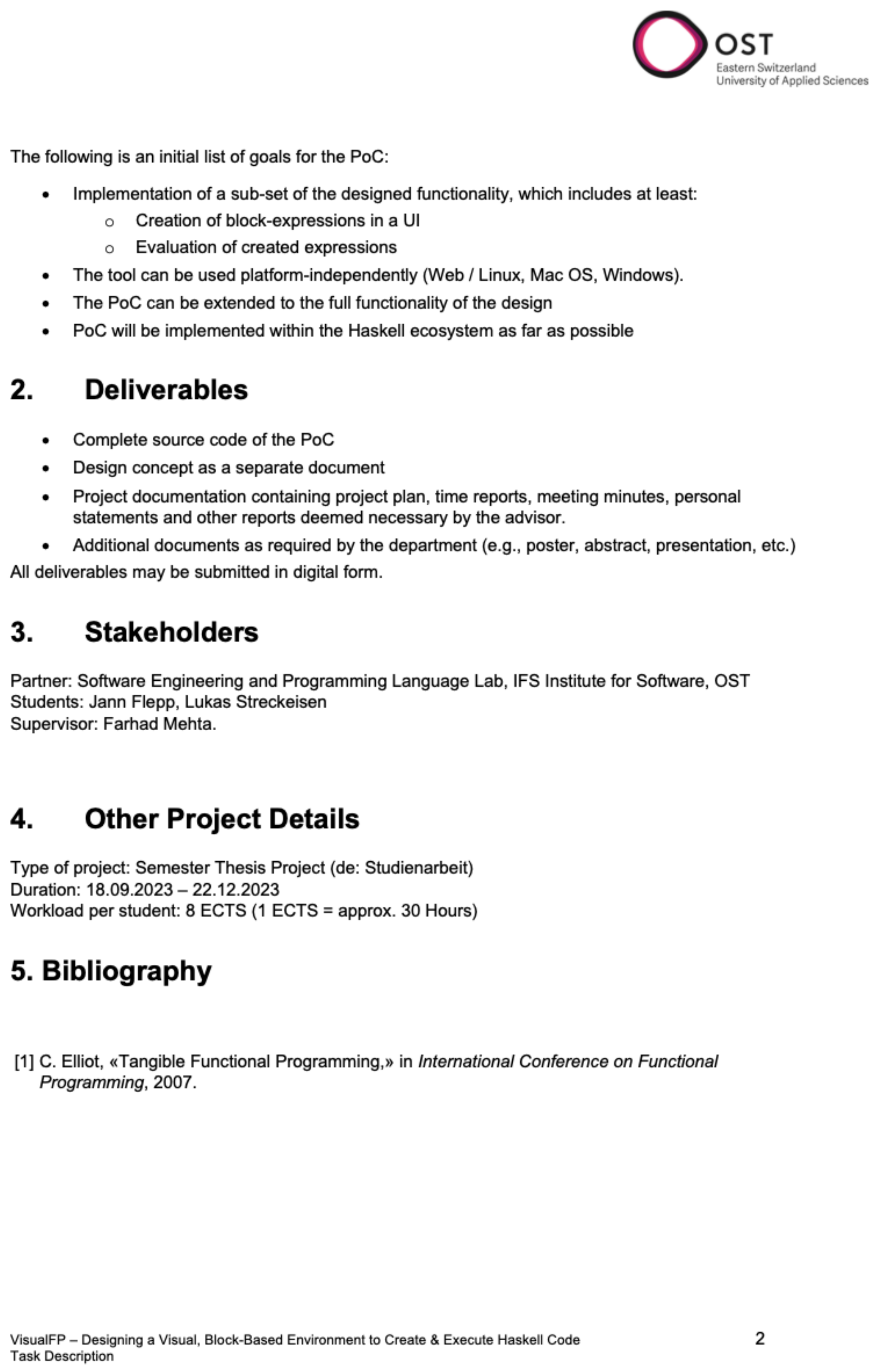
With VisualFP, we want to design a tool that supports graphical development of Haskell code, is block-based, and extensible.

The main aim of this project is to propose a design and a model for user interaction for the VisualFP tool, and to provide a proof a concept (PoC) for it. The PoC will be limited to a small sub-set of the proposed design.

The following is an initial list of tasks towards this main aim:

- Research and review of the current state of the art in this area
- UI Mockups/Wireframes describing the proposed design and user interaction
- Description and evaluation of the considered design choices
- Usability Tests with the PoC and analysis of their results
- Concept for extensibility
- Concept for importing Haskell code

VisualFP – Designing a Visual, Block-Based Environment to Create & Execute Haskell Code 1
Task Description



The following is an initial list of goals for the PoC:

- Implementation of a sub-set of the designed functionality, which includes at least:
  - Creation of block-expressions in a UI
  - Evaluation of created expressions
- The tool can be used platform-independently (Web / Linux, Mac OS, Windows).
- The PoC can be extended to the full functionality of the design
- PoC will be implemented within the Haskell ecosystem as far as possible

2. Deliverables

- Complete source code of the PoC
- Design concept as a separate document
- Project documentation containing project plan, time reports, meeting minutes, personal statements and other reports deemed necessary by the advisor.
- Additional documents as required by the department (e.g., poster, abstract, presentation, etc.)

All deliverables may be submitted in digital form.

3. Stakeholders

Partner: Software Engineering and Programming Language Lab, IFS Institute for Software, OST
Students: Jann Flepp, Lukas Streckeisen
Supervisor: Farhad Mehta.

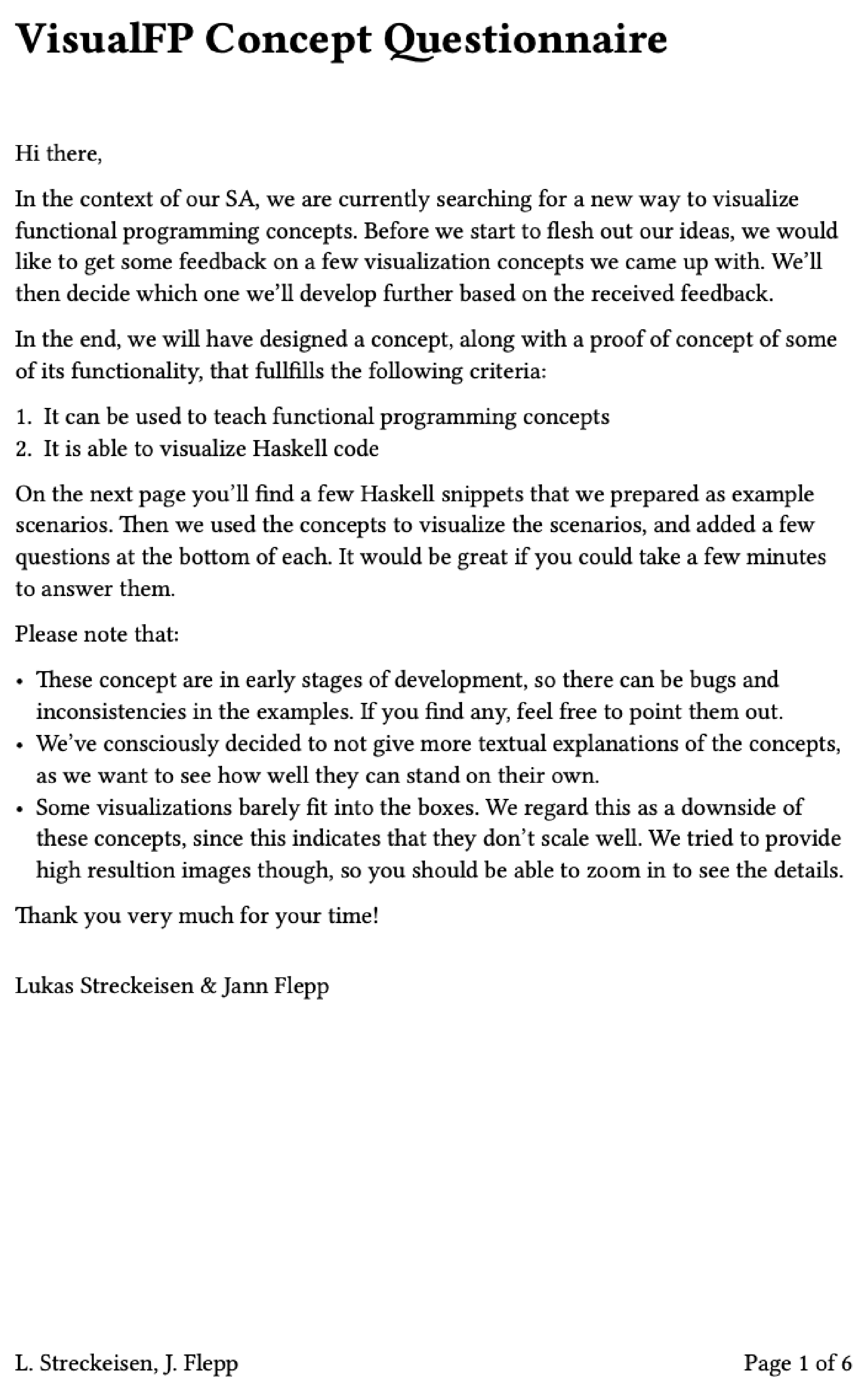
4. Other Project Details

Type of project: Semester Thesis Project (de: Studienarbeit)
Duration: 18.09.2023 – 22.12.2023
Workload per student: 8 ECTS (1 ECTS = approx. 30 Hours)

5. Bibliography

[1] C. Elliot, «Tangible Functional Programming,» in *International Conference on Functional Programming*, 2007.

17. Design Evaluation Questionnaire Template



VisualFP Concept Questionnaire

Hi there,

In the context of our SA, we are currently searching for a new way to visualize functional programming concepts. Before we start to flesh out our ideas, we would like to get some feedback on a few visualization concepts we came up with. We'll then decide which one we'll develop further based on the received feedback.

In the end, we will have designed a concept, along with a proof of concept of some of its functionality, that fullfills the following criteria:

1. It can be used to teach functional programming concepts
2. It is able to visualize Haskell code

On the next page you'll find a few Haskell snippets that we prepared as example scenarios. Then we used the concepts to visualize the scenarios, and added a few questions at the bottom of each. It would be great if you could take a few minutes to answer them.

Please note that:

- These concept are in early stages of development, so there can be bugs and inconsistencies in the examples. If you find any, feel free to point them out.
- We've consciously decided to not give more textual explanations of the concepts, as we want to see how well they can stand on their own.
- Some visualizations barely fit into the boxes. We regard this as a downside of these concepts, since this indicates that they don't scale well. We tried to provide high resultion images though, so you should be able to zoom in to see the details.

Thank you very much for your time!

Lukas Streckeisen & Jann Flepp

L. Streckeisen, J. Flepp Page 1 of 6

Scenarios

Simple Addition Function

```haskell
addition :: Num a => a -> a -> a
addition a b = (+) a b
```

Even Numbers from 1 to 10

```haskell
evenOneToTen :: Integral a => [a]
evenOneToTen = filter even
  (takeWhile
    ((<= 10))
    (iterate (+1) 1))
```

Product of Numbers

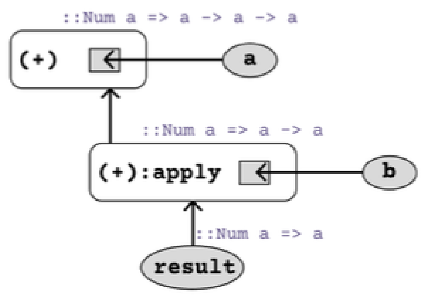
```haskell
product :: Num a => [a] -> a
product [] = 1
product (n : ns) = (*) n (product ns)
```

Map Add 5 Function

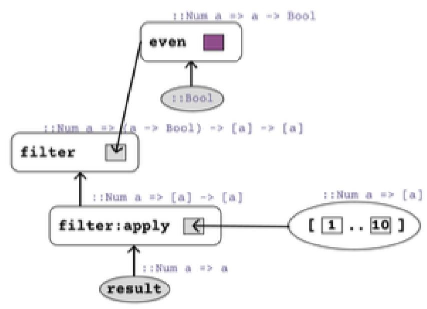
```haskell
mapAdd5 :: Num a => [a] -> [a]
mapAdd5 = map (+ 5)
```

Flo inspired

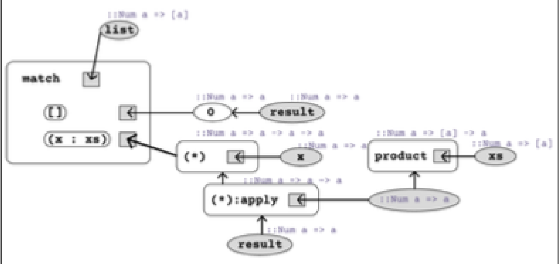
Simple Addition Function



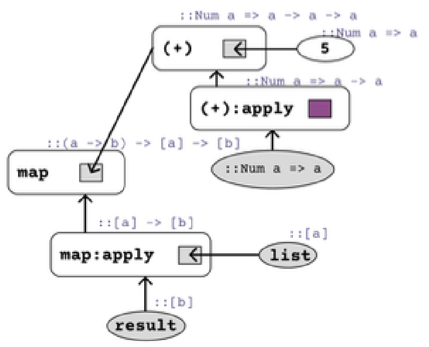
Even Numbers from 1 to 10



Product of Numbers



Map Add 5 Function



Were you able to understand the meaning of the boxes and arrows?

Do you find the concept nice to look at?

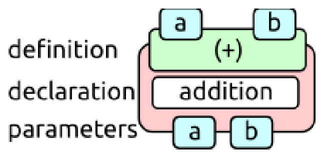
Could you imagine teaching functional programming using this vizualization?

Could you imagine how the concept scales to more complex expressions?

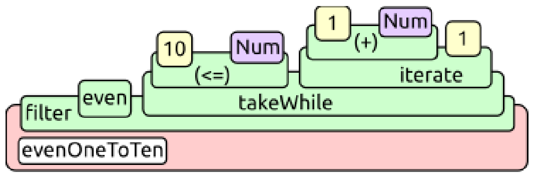
Do you have any suggestions for improvement or general comments on the concept?

Scratch inspired

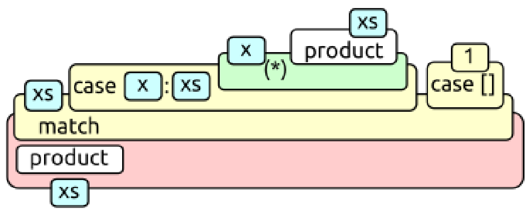
Simple Addition Function



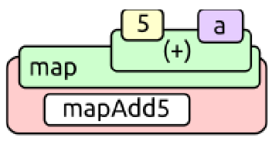
Even Numbers from 1 to 10



Product of Numbers



Map Add 5 Function



Were you able to understand the meaning of the boxes and arrows?

Do you find the concept nice to look at?

Could you imagine teaching functional programming using this vizualization?

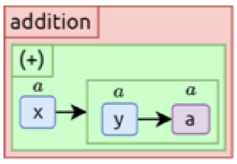
Could you imagine how the concept scales to more complex expressions?

Do you have any suggestions for improvement or general comments on the concept?

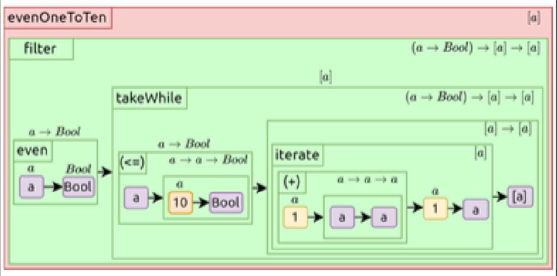
L. Streckeisen, J. Flepp Page 4 of 6

Haskell Function-Notation inspired

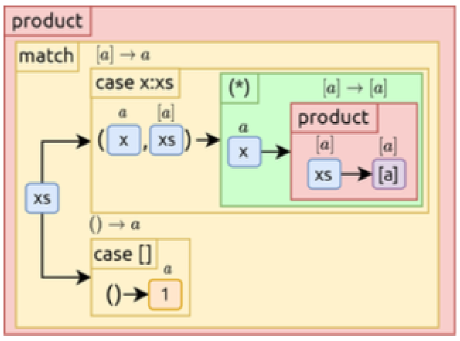
Simple Addition Function



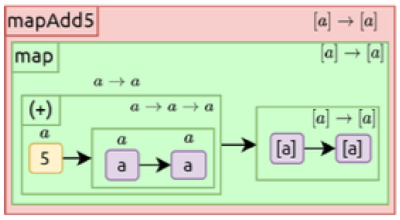
Even Numbers from 1 to 10



Product of Numbers



Map Add 5 Function



Were you able to understand the meaning of the boxes and arrows?

....

Do you find the concept nice to look at?

....

Could you imagine teaching functional programming using this vizualization?

....

Could you imagine how the concept scales to more complex expressions?

....

Do you have any suggestions for improvement or general comments on the concept?

....

L. Streckeisen, J. Flepp Page 5 of 6

If you have an own idea for a visualization concept, we would be happy to see it!

L. Streckeisen, J. Flepp Page 6 of 6

18. Design Iteration One - Survey Results

The design evaluation questionnaire (as described in Section 6.1.2) was given to 7 students and exprienced programmers. These are the results:

18.1.1. Survey Results from Prof. Dr. Farhad Mehta

Prof. Dr. Farhad Mehta is a lecturer at OST and advisor of this project.

Flo-inspired

Question	Answer
Were you able to understand the meaning of the boxes and arrows?	Not really. Semantics & the arrows are unclear (insertion or reverse result)
Do you find the concept nice to look at?	Not really.
Could you imagine teaching functional programming using this vizualization?	Not really. The arrows obsucre the denotational semantics.
Could you imagine how the concept scales to more complex expressions?	Yes. The arrows allow blocks to remain small.
Do you have any suggestions for improvement or general comments on the concept?	

Table 3: Design questionnaire answers for Flo-inspired design from Prof. Dr. Farhad Mehta

Scratch-inspired

Question	Answer
Were you able to understand the meaning of the boxes and arrows?	Somewhat better than the Flo-inspired version.
Do you find the concept nice to look at?	Somewhat better than the Flo-inspired version.
Could you imagine teaching functional programming using this vizualization?	Somewhat better than the Flo-inspired version.
Could you imagine how the concept scales to more complex expressions?	Somewhat better than the Flo-inspired version.
Do you have any suggestions for improvement or general comments on the concept?	Without types, one has no guidance on which blocks fit where

Table 4: Design questionnaire answers for Scratch-inspired design from Prof. Dr. Farhad Mehta

Haskell Function-Notation inspired

Question	Answer
Were you able to understand the meaning of the boxes and arrows?	Better than the other two, but not quite there yet.
Do you find the concept nice to look at?	Better than the other two, but not quite there yet.
Could you imagine teaching functional programming using this vizualization?	Better than the other two, but not quite there yet.
Could you imagine how the concept scales to more complex expressions?	Better than the other two, but not quite there yet.
Do you have any suggestions for improvement or general comments on the concept?	

Table 5: Design questionnaire answers for Haskell Function-Notation inspired design from Prof. Dr. Farhad Mehta

General Comments

The questionnaire may not do full justice to the visual programming methods since it only reviews the end state, and not the method of programming.

All methods seem to have a "bottom-up" strategy on constructing programs (i.e. start with small steps with what is available, and tinker with it unit you come up with something that you can use). The imperative paradigm forces one to do this (top level blocks are always ";", and therefore uninteresting). In FP, we are able to design our programs "top-down", starting with a specification (type definition at least). This specification often admits a top-level function that is often interesting (e.g. filter, map), with further "holes" that can similarly be filled successively.

It may be a good idea to design the VP tool around to support the method we want people to learn "how to design programs" (see "recipe for defining functions" & video on "Schreib dein program").

There are huge parallels between programming & constructing formal proofs (Curry-Howard-Lambek isomorphism) that can be a mental aid in designing such a tool - even if one does not immediately expose this to the beginner (please don't).

The more I think about it, the more I am under the impression the VP tool and concept should support the existing recommended methodology and process of designing (functional) programs. This process has been quite well thought out, and does not need to be re-invented. What I feel is missing, when doing this in a textual form, is that the "visual model" of what this text should look like in the minds of the learners is not immediately visible. A visual tool can help learners build the correct visual model/intuition faster. Once this visual model/intuition is finally in place, the tool will often little benefit and become tedious to use. The users will then switch to the textual representation, but still always have the visual model in mind.

18.1.2. Survey Results from Raphael Das Gupta

Raphael Das Gupta is a scientific employee at the institute for software at OST

Flo-inspired

Question	Answer
Were you able to understand the meaning of the boxes and arrows?	Mostly. I was first wondering why the arrow in the "Product of Numbers" example goes from the interim-result-ellipse 'Num a => a' to the argument slot of "(*):apply", instead of the product-block as with all other cases where functions are passed as parameters. But then I realised that the result of the function call with xs is passed and not the function itself.
Do you find the concept nice to look at?	No, too noisy.
Could you imagine teaching functional programming using this vizualization?	Perhaps, but only as an aid to show certain signatures of a partial expression, not in general to teach functional programming from the ground up.
Could you imagine how the concept scales to more complex expressions?	It'll get very complex very fast.
Do you have any suggestions for improvement or general comments on the concept?	• Move type-signatures into the blocks instead of above them • Make type-signatures hideable • Option to switch between curried-interpretation and n-ary-function interpretation

Table 6: Design questionnaire answers for Flo-inspired design from Raphael Das Gupta

Scratch-inspired

Question	Answer
Were you able to understand the meaning of the boxes and arrows?	I think so.
Do you find the concept nice to look at?	Yes
Could you imagine teaching functional programming using this vizualization?	Yes, but I don't see an advantage compared to a pretty AST.
Could you imagine how the concept scales to more complex expressions?	It'll look like a mountain-skyline.
Do you have any suggestions for improvement or general comments on the concept?	Highlight which argument-instances belong to which argument-bindings when hovering over them.

Table 7: Design questionnaire answers for Scratch-inspired design from Raphael Das Gupta

Haskell Function-Notation inspired

Question	Answer
Were you able to understand the meaning of the boxes and arrows?	Mostly, but I'm not sure if I understood everything right.
Do you find the concept nice to look at?	Yes
Could you imagine teaching functional programming using this vizualization?	Perhaps, but only as an aid to show certain signatures of a partial expression, not in general to teach functional programming from the ground up.
Could you imagine how the concept scales to more complex expressions?	It would probably get complex too, but probably not as complex as the other two designs.
Do you have any suggestions for improvement or general comments on the concept?	• Put the function-types next to the function name, so that there is no danger of confusing them. • Your approach for pattern-matching nicely shows that you don't have access to parts of a pattern that aren't named. But somehow the way it's visualized seems strange to me and is somewhat unsatisfying. But I don't know how to do it better.

Table 8: Design questionnaire answers for Haskell Function-Notation inspired design from Raphael Das Gupta

General Comments

I quite like the bock-arrow diagrams in "The state monad" in "Programming in Haskell" by Graham Hutton (second edition, chapter 12.3 Monads, pages 168 - 141). I don't know how well that approach generalises without overloading it like the Flo-inspired examples. In contrast to your examples the diagrams from the book show the data flow (but not how calls are plugged together syntactically).

18.1.3. Survey Results from Noah Geeler

Noah Geeler is a third-year software-development apprentice at Vontobel.

Flo-inspired

Question	Answer
Were you able to understand the meaning of the boxes and arrows?	No, but I assume that the squares are some kind of input?
Do you find the concept nice to look at?	If I understood this concept, I assume that I would've thought that it looked to complicated.
Could you imagine teaching functional programming using this vizualization?	
Could you imagine how the concept scales to more complex expressions?	
Do you have any suggestions for improvement or general comments on the concept?	

Table 9: Design questionnaire answers for Flo-inspired design from Noah Geeler

Scratch-inspired

Question	Answer
Were you able to understand the meaning of the boxes and arrows?	I think I understood this concept the most.
Do you find the concept nice to look at?	Yes
Could you imagine teaching functional programming using this vizualization?	Probably.
Could you imagine how the concept scales to more complex expressions?	I think complex expressions would take up a wide space and would be very complicated to understand.
Do you have any suggestions for improvement or general comments on the concept?	Keep the explanation (like in the first example) of the boxes (definition, declaration & parameters)

Table 10: Design questionnaire answers for Scratch-inspired design from Noah Geeler

Haskell Function-Notation inspired

Question	Answer
Were you able to understand the meaning of the boxes and arrows?	No.
Do you find the concept nice to look at?	
Could you imagine teaching functional programming using this vizualization?	
Could you imagine how the concept scales to more complex expressions?	
Do you have any suggestions for improvement or general comments on the concept?	

Table 11: Design questionnaire answers for Haskell Function-Notation inspired design from Noah Geeler

General Comments

18.1.4. Survey Results from Mathias Fischler

Mathias Fischler is a student at OST and has visited the functional programming lecture.

Flo-inspired

Question	Answer
Were you able to understand the meaning of the boxes and arrows?	
Do you find the concept nice to look at?	No, very confusing with too many arrows and annotations.
Could you imagine teaching functional programming using this vizualization?	
Could you imagine how the concept scales to more complex expressions?	
Do you have any suggestions for improvement or general comments on the concept?	

Table 12: Design questionnaire answers for Flo-inspired design from Mathias Fischler

Scratch-inspired

Question	Answer
Were you able to understand the meaning of the boxes and arrows?	
Do you find the concept nice to look at?	Yes
Could you imagine teaching functional programming using this vizualization?	
Could you imagine how the concept scales to more complex expressions?	
Do you have any suggestions for improvement or general comments on the concept?	• No type-annotations, so it's difficult to tell what goes where • Type-Hole isn't intuitive • Operators should be treated like any other function

Table 13: Design questionnaire answers for Scratch-inspired design from Mathias Fischler

Haskell Function-Notation inspired

Question	Answer
Were you able to understand the meaning of the boxes and arrows?	
Do you find the concept nice to look at?	Yes
Could you imagine teaching functional programming using this vizualization?	
Could you imagine how the concept scales to more complex expressions?	
Do you have any suggestions for improvement or general comments on the concept?	

Table 14: Design questionnaire answers for Haskell Function-Notation inspired design from Mathias Fischler

General Comments

It would be nice to have 'referential-transparency', i.e. hovering over a block to see the type of a specific argument.

18.1.5. Survey Results from Lukas Buchli

Lukas Buchli is a technical employee at the institute for software at OST

Flo-inspired

Question	Answer
Were you able to understand the meaning of the boxes and arrows?	I don't know Flo and for me it is not a very obvious notation. I can guess the semantics though.
Do you find the concept nice to look at?	It looks a bit cluttered to me.
Could you imagine teaching functional programming using this vizualization?	I think I would visualize it differently.
Could you imagine how the concept scales to more complex expressions?	It will probably clutter quite fast, I already find 'Product of Numbers' hard to read. I don't see a simple way to split it into multiple parts.
Do you have any suggestions for improvement or general comments on the concept?	Maybe multiple argument functions can have the argument in the same block instead of the :apply notation? I understand that this is to highlight currying, but I think you could also explain this by only highlighting the empty argument boxes. This would reduce clutter and make it more scalable.

Table 15: Design questionnaire answers for Flo-inspired design from Lukas Buchli

Scratch-inspired

Question	Answer
Were you able to understand the meaning of the boxes and arrows?	I find this quite easy to read. The only confusing bits I find are the type annotations (purple), especially because it mixes up constraints and types, but also because it could be interpreted as being part of the lower layer (i.e. in 'Map Add 5 Function' it could be interpreted as describing the (+) and not the 5).
Do you find the concept nice to look at?	Yes, it looks clean and colorful.
Could you imagine teaching functional programming using this vizualization?	Yes.
Could you imagine how the concept scales to more complex expressions?	It seems to clutter up less fast, and even then, it could be possible to split it up into multiple towers with references to each other (maybe when visualizing Haskell code, definitions in 'where' or in a let expression could be a separate tower, this would also solve the problem of multiple references.
Do you have any suggestions for improvement or general comments on the concept?	• Type annotations: There could be a separate type annotation tower that can be enabled or disabled. Or it should be more obvious where the type annotation applies. At the moment it looks like the types are arguments to the function (which is actually the case in GHC Core or with the TypeApplications extension, but not in normal Haskell). Constraints should be ignored or handled differently. • Infix functions should look like +, not (+), if they are visualized in an infix way.

Table 16: Design questionnaire answers for Scratch-inspired design from Lukas Buchli

Haskell Function-Notation inspired

Question	Answer
Were you able to understand the meaning of the boxes and arrows?	I find this one difficult to read. I especially have difficulty with the apparent mix-up of types and values. It seems that the last part of an arrow chain is the return type, and the rest is a value if present and a type if partially applied? I like the currying visualization with nested boxes though.
Do you find the concept nice to look at?	It looks more formal than Scratch-inspired, which to me is a disadvantage. It also has more text.
Could you imagine teaching functional programming using this vizualization?	No, I find it difficult to describe the semantics of single components. Maybe I'd be able to if you gave me an explanation of their meaning.
Could you imagine how the concept scales to more complex expressions?	I guess it would be possible to use cross references. It looks less cluttered than the Flo -inspired one.
Do you have any suggestions for improvement or general comments on the concept?	It seems like the single component semantics are not entirely consistent here.

Table 17: Design questionnaire answers for Haskell Function-Notation inspired design from Lukas Buchli

General Comments

- I think it is important to have clear and simple semantics for single components of your visualization. In order to ensure this, it may be useful to think about reduction rules for your visualization.
- I like your use of color and how it distinguishes different things (types, value, arguments, …)
- Type polymorphism and constraints seems to be a challenge to visualize. For polymorphic types, TypeApplications may be a useful inspiration (i.e. receive types as a different kind of argument to functions). Constraints could maybe then be applied to these kinds of type arguments. Con of this approach is that in Haskell, you don't pass types as arguments.
- Do you also plan on visualizing type definitions?
- My vote is on a Scratch-inspired version.

18.1.6. Survey Results from Eliane Schmidli

Eliane Schmidli is a master student & scientific assistant at the institute for software at OST

Flo-inspired

Question	Answer
Were you able to understand the meaning of the boxes and arrows?	Ja, ich bin mir jedoch nicht sicher, ob man es ohne Haskell Erfahrung versteht. Ausserdem hätte ich die Pfeile fürs Verständnis eher von unten nach oben gemacht (siehe erste Box). Ich möchte nicht vom Resultat zurück gehen, sondern wende ein Argument nach dem anderen an und gelange am Schluss zum Resultat. (wenn man jedoch die Argumente im UI dann so hinziehen kann macht von unten nach oben mehr Sinn)
Do you find the concept nice to look at?	Grundsätzlich ja, es wird jedoch schnell unübersichtlich. Es bräuchte noch mehr Farbe und die Pfeile könnten je nach Funktionalität unterschiedlich gestaltet werden.
Could you imagine teaching functional programming using this vizualization?	So wie es jetzt ist, eher nicht, da es zu unübersichtlich ist. Aber wenn es etwas ausgereifter ist, denke ich schon. Man kann es ja dann wahrscheinlich Schritt für Schritt einblenden, bzw. zusammensetzen.
Could you imagine how the concept scales to more complex expressions?	Ich glaube es wird immer unübersichtlicher…
Do you have any suggestions for improvement or general comments on the concept?	Ich finde die Rekursion nicht so verständlich. Man sieht nicht, dass product rekursiv aufgerufen wird. Ich hätte die match Box als noch mit product beschriftet und mit Farbe gearbeitet. Die ::Num a -> a Box verwirrt mich. Ausserdem fände ich es besser die Applikation in einer Box zu machen

Table 18: Design questionnaire answers for Flo-inspired design from Eliane Schmidli

Scratch-inspired

Question	Answer
Were you able to understand the meaning of the boxes and arrows?	Ja, ich finde hier sieht man am besten, wie die Parameter in einander verschachtelt sind
Do you find the concept nice to look at?	Ja, die Farben sind mega gut fürs Verständnis und es ist sehr übersichtlich. Rein visuell der beste Vorschlag.
Could you imagine teaching functional programming using this vizualization?	Gut ist hier, dass man sieht wie man Schritt für Schritt etwas einblenden könnte. Ich weiss jedoch nicht, ob es wirklich einen Mehrwert gegenüber dem Code bietet… Bzw. Man sieht wie im Code die Zusammenhänge nicht ganz
Could you imagine how the concept scales to more complex expressions?	Ich könnte mir vorstellen, dass es schnell zu überladen wird
Do you have any suggestions for improvement or general comments on the concept?	• Type annotations: There could be a separate type annotation tower that can be enabled or disabled. Or it should be more obvious where the type annotation applies. At the moment it looks like the types are arguments to the function (which is actually the case in GHC Core or with the TypeApplications extension, but not in normal Haskell). Constraints should be ignored or handled differently. • Infix functions should look like +, not (+), if they are visualized in an infix way.

Table 19: Design questionnaire answers for Scratch-inspired design from Eliane Schmidli

Haskell Function-Notation inspired

Question	Answer
Were you able to understand the meaning of the boxes and arrows?	Ich finde es schlechter verständlich als der erste Vorschlag. Ich könnte mir aber vorstellen, dass eine Kombination aus diesem und dem ersten funktionieren könnte.
Do you find the concept nice to look at?	Farben und Boxen finde ich gut und dass die Applikation und der Zusammenhang zwischen Argumenten und den Typen besser sichtbar ist. Aber es sieht irgendwie zu mathematisch aus :) Ich könnte mir vorstellen, dass das Personen abschrecken könnte
Could you imagine teaching functional programming using this vizualization?	So nicht unbedingt. Aber wenn man es mit dem ersten Vorschlag verbinden würde, denke ich schon
Could you imagine how the concept scales to more complex expressions?	Ich glaube, es wird mega kompliziert mit der Verschachtelung. Ich finde die Pfeile beim ersten Vorschlag besser
Do you have any suggestions for improvement or general comments on the concept?	It seems like the single component semantics are not entirely consistent here.

Table 20: Design questionnaire answers for Haskell Function-Notation inspired design from Eliane Schmidli

General Comments

Anhang Questionnaire

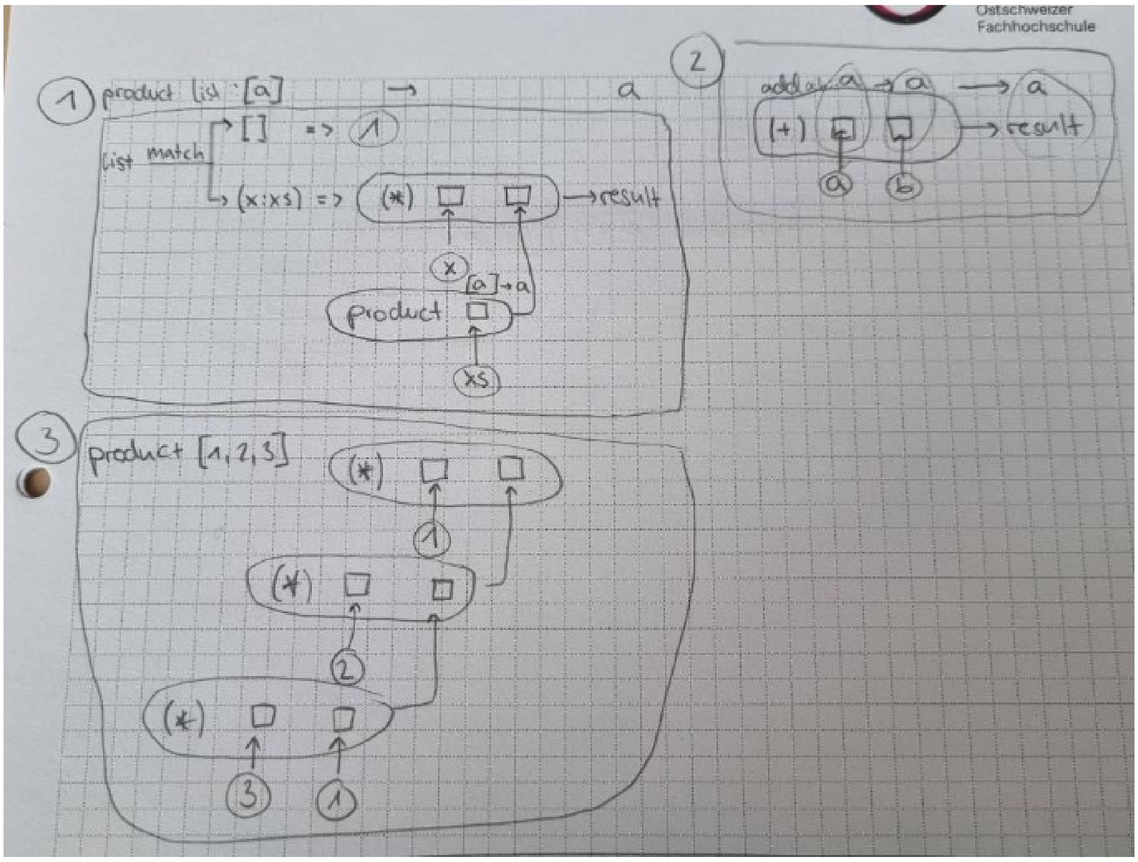
Eliane Schmidli

Ich habe hier versucht, die Vorschläge 1 und 3 von euch zu verknüpfen.

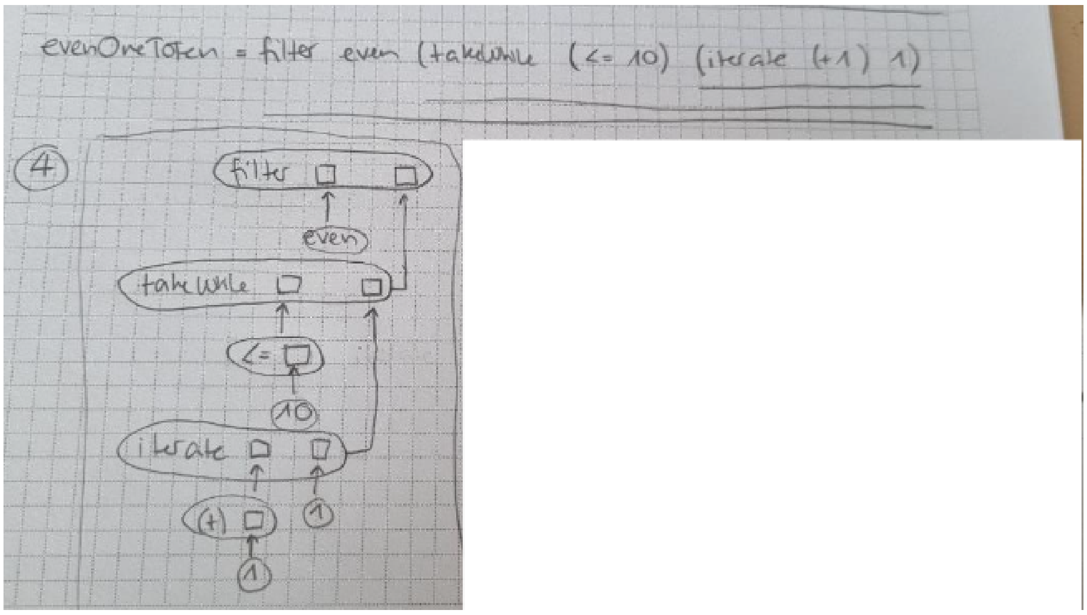
Im Allgemeinen fände ich es besser, wenn man den Zusammenhang zwischen den Typen und Argumenten sehen würde, wie bei (2) habe es jetzt mal eingekreist… Wenn jedoch überall noch Num :: usw. steht ist es auch wieder überladen. Vielleicht könnte man da noch mit Farben arbeiten, so dass alle Boxen, die eine Nummer erwarten eine einheitliche Farbe haben oder so.

(1) Ich finde es ein wenig übersichtlicher, wenn alle Argumente einer Funktion zusammen gezeichnet werden.

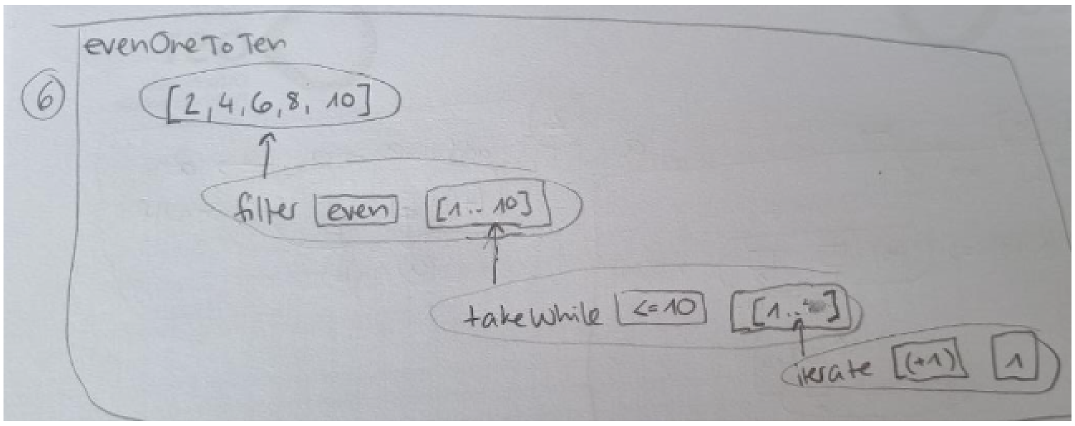
(3) Die Rekursion versteht man wahrscheinlich erst richtig, wenn man ein Beispiel verwendet. Habe das hier mal im selben Stil versucht.



Als ich versucht habe, das folgende Beispiel (4) evenOneToTen zu zeichen, ist mir aufgefallen, dass es fürs Verständnis wichtig wäre, dass man auf die Deklarationen der Standardfunktionen zurückgreifen kann. Die müsste man dann aber irgendwie auch verstehen…


evenOneToTen = filter even (takeWhile (<= 10) (iterate (+1) 1)
4
filter
even
takeWhile
<=
10
iterate
(+)
1
1

Zum Thema Haskell lernen: Ich finde eine grosse Challenge ist auch, bestehende Funktionsdeklarationen zu verstehen. Ich glaube, ihr fokussiert euch ja mehr aufs Programmieren, dann sind die Ansätze sicher gut, bei denen man die Argumente so zusammensetzen kann. Wenn es aber ums Verständnis geht, fände ich auch wichtig, dass man sieht, dass diese Argumente eins nach dem anderen mit ihrer Definition ersetzt werden. Also bei (6) würde evenOneToTen ja ungefähr so aufgeschlüsselt werden. Im Unterricht könnte ich fast eher so ein Tool gebrauchen… 😊



18.1.7. Survey Results from Timon Erhart

Timon Erhart is a scientific assistant at the institute for software at OST

Flo-inspired

Question	Answer
Were you able to understand the meaning of the boxes and arrows?	Mostly. It is confusing, that the input (e.g.) a and output (results) have the same arrow direction. It is not clear where to begin and how the data 'flows'
Do you find the concept nice to look at?	No. In my opinion it looks more complicated than the code
Could you imagine teaching functional programming using this vizualization?	No
Could you imagine how the concept scales to more complex expressions?	No. More complex would probably look more messy
Do you have any suggestions for improvement or general comments on the concept?	If grey boxes are type only, draw just a line or place it inside. But use no arrow

Table 21: Design questionnaire answers for Flo-inspired design from Timon Erhart

Scratch-inspired

Question	Answer
Were you able to understand the meaning of the boxes and arrows?	The match-case are where confusing to me. But the rest yes
Do you find the concept nice to look at?	Better than Flo. Cleaner and smaler. It has some structure visible
Could you imagine teaching functional programming using this vizualization?	Rather no
Could you imagine how the concept scales to more complex expressions?	Yes (at least better than the others)
Do you have any suggestions for improvement or general comments on the concept?	Maybe an other syntax for match-case to dinstinguish between functions names

Table 22: Design questionnaire answers for Scratch-inspired design from Timon Erhart

Haskell Function-Notation inspired

Question	Answer
Were you able to understand the meaning of the boxes and arrows?	No
Do you find the concept nice to look at?	No, gets to big/messy soon
Could you imagine teaching functional programming using this vizualization?	No
Could you imagine how the concept scales to more complex expressions?	No, gets big very soon
Do you have any suggestions for improvement or general comments on the concept?	

Table 23: Design questionnaire answers for Haskell Function-Notation inspired design from Timon Erhart

General Comments

Maybe something like a tree structure (similar to expression trees) that goes from top to bottom? It would may be some kind of mix between Flo and Scratch. Make a own symbol for match-cases (to distinguish from functions). Make sure it is tidy (same thing on same height level etc.)

19. Glossary & List of Acronyms

API: Application Programming Interface

AST: Abstract Syntax Tree - Tree representation of a program's structure

CSS: Cascading Style Sheets - Language to style the content of, e.g., web sites

DOM: Document Object Model - Tree representation of, e.g., a HTML document

FFI: Foreign Functional Interface - Interface between two different programming languages

FRP: Functional Reactive Programming - A concept that defines types and functions for interactive applications written in a functional language [22]

GHC: Glasgow Haskell Compiler

GTK: GIMP ToolKit - Toolkit for creating graphical user interfaces

HTML: HyperText Markup Language - Language to define the content structure of, e.g., web sites

JS: JavaScript

NFR: Non-Functional Requirement

OS: Operating System

PoC: Proof of Concept

UI: User Interface

20. Bibliography

[1] S. P. Jones, “Type inference as constraint solving.” https://simon.peytonjones.org/assets/ppts/type-inference.pptx (accessed: Dec. 5, 2023).

[2] B. Harvey, and J. Mönig, *SNAP! Reference Manual*, 8.0, (2020). [Online]. Available: https://snap.berkeley.edu/snap/help/SnapManual.pdf

[3] C. M. Elliot, “Tangible functional programming,” Oct. 2007. [Online]. Available: http://conal.net/papers/Eros/eros.pdf

[4] E. Lawrence, *Flo: A Visual, Purely Functional Programming Language*, (Apr. 2016). [Online]. Available: https://github.com/elliottlawrence/flo/blob/master/Documents/CSC%20411%20Final%20Report.pdf

[5] Enso International Inc, “Enso the language.” https://enso.org/language (accessed: Sep. 23, 2023).

[6] Jameshfisher, “A humble suggestion for visual pure functional programming.” https://www.reddit.com/r/haskell/comments/q7m8i/a_humble_suggestion_for_visual_pure_functional (accessed: Sep. 30, 2023).

[7] “Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models,” International Organization for Standardization, Geneva, CH, Mar. 2011.

[8] OpenJS Foundation, “Electron js.” https://www.electronjs.org/ (accessed: Sep. 28, 2023).

[9] T. G. Team, “The gtk project - a free and open-source cross-platform widget toolkit.” https://www.gtk.org/ (accessed: Oct. 20, 2023).

[10] I. G. Etxebarria, “Haskell-gi.” https://github.com/haskell-gi/haskell-gi (accessed: Oct. 20, 2023).

[11] H. Apfelbaum, “Threepenny-gui.” https://github.com/HeinrichApfelmus/threepenny-gui (accessed: Oct. 21, 2023).

[12] H. Apfelbaum, “Threepenny-gui: gui framework that uses the web browser as a display. https://hackage.haskell.org/package/threepenny-gui (accessed: Oct. 21, 2023).

[13] S. Henry, J. M. Young, L. Stegeman, and others.

[14] fsbolero.io, “Getting started | bolero: f# in webassembly.” https://fsbolero.io/docs/ (accessed: Nov. 6, 2023).

[15] GHC Developers, “The GHC Commentary.” https://gitlab.haskell.org/ghc/ghc/-/wikis/commentary (accessed: Sep. 30, 2023).

[16] S. Diehl, “Dive into ghc: pipeline.” https://www.stephendiehl.com/posts/ghc_01.html (accessed: Sep. 30, 2023).

[17] hugs Developers, “hugs online.” https://www.haskell.org/hugs/ (accessed: Sep. 30, 2023).

[18] R. Cheplyaka, E. Hesselink, and others, “Haskell Suite.” https://github.com/haskell-suite (accessed: Sep. 30, 2023).

[19] R. Cheplyaka, E. Hesselink, and others, “Haskell Source Extensions.” https://github.com/haskell-suite/haskell-src-exts#maintenance (accessed: Sep. 30, 2023).

[20] K. Renzel, and W. Keller, “Client/server architectures for business information systems a pattern language,” 1997. [Online]. Available: https://api.semanticscholar.org/CorpusID:59847813

[21] S. Brown, “Type inference as constraint solving.” https://www.infoq.com/articles/C4-architecture-model (accessed: Dec. 20, 2023).

[22] C. Elliott, and P. Hudak, “Functional reactive animation,” in *Int. Conf. Functional Program.*, 1997. [Online]. Available: http://conal.net/papers/icfp97/

[23] H. Apfelmus, “Dynamic switching events.” https://github.com/HeinrichApfelmus/threepenny-gui/issues/180 (accessed: Nov. 28, 2023).

[24] P. D. F. D. Mehta, “Lecture slides: programming in prolog unification and proof search,” Hochschule für Technik Rapperswil, 2019.

[25] F. Pottier, and D. Rémy, “The essence of ml type inference,” pp. 389–489.

[26] “Gatekeeper and runtime protection in macos.” https://support.apple.com/en-gb/guide/security/sec5599b66df/web (accessed: Dec. 15, 2023).

21. List of Figures

22. List of Tables

23. List of Code Listings

Disclaimer

Parts of this paper were rephrased using the following tools:

- GitHub Copilot[1]
- Grammarly[2]

[1]https://github.com/features/copilot/

[2]https://www.grammarly.com/