



This is a peer-reviewed, post-print (final draft post-refereeing) version of the following published document, © 2016 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. and is licensed under All Rights Reserved license:

**Alam, Abu S and Bush, Vicky J ORCID logoORCID:
<https://orcid.org/0000-0002-8733-4358> (2016) HASKEU: An
editor to support visual and textual programming in tandem.
Proceedings of the 2016 SAI Computing Conference. pp. 805-
814.**

Official URL: <http://saiconference.com/Conferences/Computing2016>

EPrint URI: <https://eprints.glos.ac.uk/id/eprint/4243>

Disclaimer

The University of Gloucestershire has obtained warranties from all depositors as to their title in the material deposited and as to their right to deposit such material.

The University of Gloucestershire makes no representation or warranties of commercial utility, title, or fitness for a particular purpose or any other warranty, express or implied in respect of any material deposited.

The University of Gloucestershire makes no representation that the use of the materials will not infringe any patent, copyright, trademark or other property or proprietary rights.

The University of Gloucestershire accepts no liability for any infringement of intellectual property rights in any material deposited but will remove such material from public view pending investigation in the event of an allegation of any such infringement.

PLEASE SCROLL DOWN FOR TEXT.

HASKEU: An editor to support visual and textual programming in tandem

Abu Alam

Vicky Bush

Abstract

Typical text editors allow "Free Typing" (syntax/semantics-free editing) to construct and maintain textual programs rapidly and easily by expert programmers, but this style does not support novice programmers learning the syntax and semantics of a language. On the other hand, visual editors enforce correct construction of programs using syntax-directed editing. Additionally, they clearly represent the program flow in a visual way. Both textual and visual editors have pros and cons for learners and experts. This paper argues that an editor that offers both a visual and a textual representation of the program in tandem has advantages to offer both learner and expert programmers. It describes the editor, HASKEU, which embodies this principle by providing visual and textual support for editing Haskell functional programs.

I. INTRODUCTION

Editors to support program development commonly display the program in a purely textual format e.g. GNU Emacs [1], NetBeans [2]. However, editors that use a visual representation are becoming more popular e.g. Scratch (Resnick et al, 2009). It is expected that a learner programmer will find a visual program editor more useful than a textual one [3]–[6]. An expert programmer, on the other hand, is more likely to find that developing a program using a textual editor is easier and faster [7]. However, they might find it more effective to use a visual programming environment for maintaining a program and checking the semantics of the program coding [8], [9].

This paper demonstrates the usefulness of a combined textual and visual system where changes propagate between both interfaces. This use of visual and textual representations in tandem allows learner programmers to quickly develop programs, focusing on the visual format but at the same time seeing the textual representation. It also allows them to understand the propagation of changes in both formats. Then they can go on to more advanced levels of understanding textually when they are ready. The expert programmers can also benefit from such a system by using both systems effectively as appropriate for debugging and maintenance.

In the next section we will briefly define visual and textual programming and see an example in both. In the third section we will explore the benefits of visual programming for a learner programmer. In the fourth section we will see the benefits of textual programming for an expert programmer. In the fifth section we will examine the benefits of visual programming for an expert programmer. Finally we will conclude that an editor that combines visual and textual representations in tandem has advantages to offer both learner and expert programmers. Such an editor, HASKEU, was developed in a research project to support end-user functional programming for Haskell programs [10]. All the program examples in this paper were developed using HASKEU.

II. VISUAL VERSUS TEXTUAL PROGRAMMING

Learning to program is often a time-consuming and frustrating endeavour. Moreover, even after the skill is learned, writing and testing programs can be a laborious activity. A textual program is one where the program is presented as a set of commands using text to define functions, variables etc.

Textual programming exploits our ability to think analytically, logically, and verbally, whereas visual programming [3]–[5], defined as “the use of meaningful graphical representations in the process of

programming”, exploits our ability to think nonverbally [6]. Visual representations aid understanding and memory retention, and may provide an incentive to learn to program without language barriers. In generic programming (as opposed to domain-specific programming), visual versions of a textual language can ease program understanding [11].

A. Textual Programming in Haskell

Haskell is a well-developed, powerful functional programming language created in 1990 by a committee of functional programmers [12]. Functional languages are based on the lambda calculus [13]. These mathematical roots mean that programs have no notion of state. Instead, they are pure functions which take input and produce some output. In recent years there has been an increase in interest in functional languages, perhaps due to the ease with which they can handle concurrent programming because of lack of side-effects [14].

Many popular imperative languages now incorporate ideas from functional languages. For example, Java and C# have lambda expressions [15], [16]. This section describes the textual programming syntax of Haskell.

1) Functions: In Haskell, a function determines a result, which depends on one or more arguments. It may be defined by one or more equations. Each equation is called a clause and the arrangement of the list of parameters in each equation is often called a pattern, which may be recursive (that is, a function defined in terms of itself). For example, the factorial function may be defined recursively as follows:

```
fac 0 = 1
```

```
fac n = n * fac (n - 1)
```

Here, the first equation will be applied when the argument value is zero and the second otherwise. Parentheses are used to group parts of an expression explicitly, but operator precedence often allows them to be omitted. Haskell assigns numeric precedence values to operators, giving function application by juxtaposition a higher priority than all other operations. So, `fac n - 1` is equivalent to `(fac n) - 1`.

2) Types: Types describe values. Among the basic types in Haskell are Integer (infinite-precision integers), Char (characters) and Bool (booleans). Among the function types are Integer -> Integer (functions mapping infinite precision integers to infinite-precision integers). Polymorphic types contain variables such as `a`. For example, the identity function may be defined as follows:

```
id :: a -> a
```

```
id x = x
```

Here, the type `a ! a` can be read as “for all types `a`, a function from `a` to `a`”. By convention, a specific type begins with capital letter, and a variable type with a lower-case one.

A new type is defined by a data declaration. For example, a binary tree may be defined as follows:

```
data Tree a
```

```
= Leaf a
```

```
j Branch (Tree a) (Tree a)
```

Here, the identifiers `Leaf` and `Branch` are the constructors of the type `Tree`. As can be seen in this example, data types may be recursive and include polymorphic components.

Lists are one of the built-in types. All items in a list have the same type. There are two list constructors, `[]` and `(:)`, so that `[]` is an empty list, `3 : []` is a list of one item, and `1 : 2 : 3 : []` is a list of three items, which may be more conveniently written as `[1, 2, 3]`.

A tuple type is another built-in type. The items in a tuple may have different types. For example, (t_1, t_2, \dots, t_n) is a tuple type of values (v_1, v_2, \dots, v_n) , where each value v_i has the type t_i given in the corresponding position in the tuple type. These objects are usually called pairs, triples, quadruples and so on.

3) Higher-order functions: A higher-order function is one that takes one or more functions as arguments or returns a function as a result. Two important higher-order functions for list-processing are `map` and `filter`. A `map` constructs a list by applying a function, passed as the first argument, to all items in a list, passed as the second argument:

```
map :: (a -> b) -> [a] -> [b]
```

```
map f [] = []
```

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

A *filter* constructs a list from the items of a list passed as the second argument that satisfy a predicate passed as the first argument:

```
filter :: (a -> Bool) -> [a] -> [a]
```

```
filter p [] = []
```

```
filter p (x:xs) =
```

```
  if p x
```

```
  then x:filter p xs
```

```
  else filter p xs
```

4) Multiple arguments and currying: All functions of more than one argument are curried: i.e. they take a single argument and return another function if more arguments are needed. The following example is a function with three integer arguments:

```
multiplySum x y z = x * (y + z)
```

Given a single integer as an argument `multiplySum` yields a function of type `Int ! Int ! Int` as result.

By convention function application is left-associative, taking one argument at a time. So, the above function definition

```
multiplySum x y z
```

is equivalent to

```
((multiplySum x) y) z
```

that is, an application of multiplySum to x, the result of which is applied to y; so (multiplySum x) must be a function. The result of this application, ((multiplySum x) y), is then applied to z, so ((multiplySum x) y) must also be a function.

The advantage of currying is that a function can be applied by binding some but not all of its arguments. Hence, the function yields a specialized version with the given arguments “frozen in”, which is also known as partial application [17].

5) Local definitions: Frequently, local definitions are used to break a big calculation into a number of smaller ones. The following example uses where to make two local definitions:

```
sumSquares :: Int -> Int -> Int
```

```
sumSquares m n =
```

```
squareM + squareN
```

```
where
```

```
squareM = m * m
```

```
squareN = n * n
```

Another way to make a local definition uses let:

```
sumSquares m n =
```

```
let squareM = m * m
```

```
squareN = n * n
```

```
in squareM + squareN
```

6) Input and output: Haskell accommodates input and output using special values called actions. An action of type IO t may perform an input/output operation with a result of type t. For example, a function to read a string from a file handle (a File yields a handle when it is opened to perform any read/write operations on the contents of that file) has type:

```
hGetStr :: Handle -> IO String
```

The keyword do may be used to introduce a sequence of actions. For example, to read a string from a file handle and print it on the standard output, the instructions are:

```
do
```

```
s <- hGetStr
```

```
putStr s
```

B. Visual Programming in HASKEU

HASKEU is a prototype Haskell programming development environment developed by Alam [10] to support end-user programming in a purely functional programming language. This end-user programming system was developed to support both visual and textual programming, aiming to allow endusers to perform some useful visual programs with a small investment of time and for them then go on to more advanced levels of understanding textually when they are ready.

The design of HASKEU makes extensive use of Human-Computer Interaction (HCI) techniques [18]. The HCI techniques include rules of user interface design, data display design, icon design, and direct manipulation. The 8 golden rules of user interface design [19] were applied to the design of HASKEU in order to improve the usability of the system:

- 1) Consistency - the user interface is consistent for all the operations. For example, in HASKEU a similar mechanism is used to select and edit each item (Item views change to indicate selection or editing. A blue outlined rectangle indicates selection and an annotation indicates an editing).
- 2) Shortcuts - frequent users. For example, ctrl + x to cut, ctrl + c to copy and ctrl + v to paste text and visual items in HASKEU. This also demonstrates consistency with other, widely used editors.
- 3) Feedback - for every action performed by the user. For example, the type information of an individual argument is shown with descriptions when the mouse pointer hovers over that argument slot and the whole type information of an application is shown when the mouse pointer is over the application box (tooltip text).
- 4) Simplicity - For example, the information of only one function is displayed at a time with function parameters, function body and local functions displayed in separate panes. It is possible to view the module level as well as the detail of a function.
- 5) Simple error handling - to make sure the user does not make serious errors. Errors are flagged up visually at the point they occur so the user can easily understand their source.
- 6) Easy reversal of actions - an 'Undo' option is provided.
- 7) Support internal locus of control - this lets experienced users feel that they are in control of the system. The system is designed so that it makes users the initiators of actions, not the responders. For example, every displayed item is editable and each pane is resizable and scrollable.
- 8) Reduce short-term memory load - by designing screens where options are clearly visible, such as provision of menus for undo and redo.

The visual display area of HASKEU has a layout inspired by that of the textual syntax of a function definition, which is

```
{<function-name> <pattern-parameters> = <function-body>
{where
<local-function>}*
}+
```

[Here, * denotes a function can have 0 or more local functions and + denotes a function can have 1 or more clauses]

Figure 1 shows the organization of the visual display area, which is split into five panes. The left-hand column lists the global and local functions and a numbered (in order) box for each clause in the function definition. When a clause is selected, the corresponding pattern parameter is displayed in the second column. Again, this is divided into global and local function parameter panes. The third column shows the body of the selected clause in the form of a dataflow graph, as explained later.

All items in this system are displayed as annotated icons (another example of data consistency). Incorporating graphics or pictures into the programming process adds an interesting and useful

dimension [6]. A significant number of iconic languages have been reported in the literature, LabVIEW being a notable example [20]. In general, they have the same goal: to use icons as programming language constructs. Icons, being pictorial, are generally easier to understand than text. There is a danger in iconic systems that icons can be ambiguous [21]. As there are no universally accepted icons, evolving icons may take time to be understood [22]. To overcome this, visual iconic languages can be designed based on the concept of generalized icons, which aim to overcome this drawback. In such an approach, each icon consists of a physical part - an image, and a logical part - a name and some additional attributes. The HASKEU visual system was built upon the concept of generalized icons. The design of HASKEU icons is explained below. With fewer icons to understand, it has an advantage over systems which do not use generalized icons such as Labview which has many different icons for the programmer to learn. The extra information in the logical part also aids understanding.

Figure 2 shows some icons used in this system. The icons in HASKEU were designed by applying a semiotic theory of five levels of icon design, where the first four levels were identified by Marcus [23] and the fifth one by Shneiderman [19]:

1) Lexical: Machine-generated marks including colour, brightness. In figures 2a and 2b, items in the left side of a function definition, which can be used in a function body, use a grey-blue rectangle (for example, a function name or a parameter variable). All the other items use a white rectangle as can be seen in figures 2d and 2h (for example, items in a function body or a wild card parameter in a pattern).

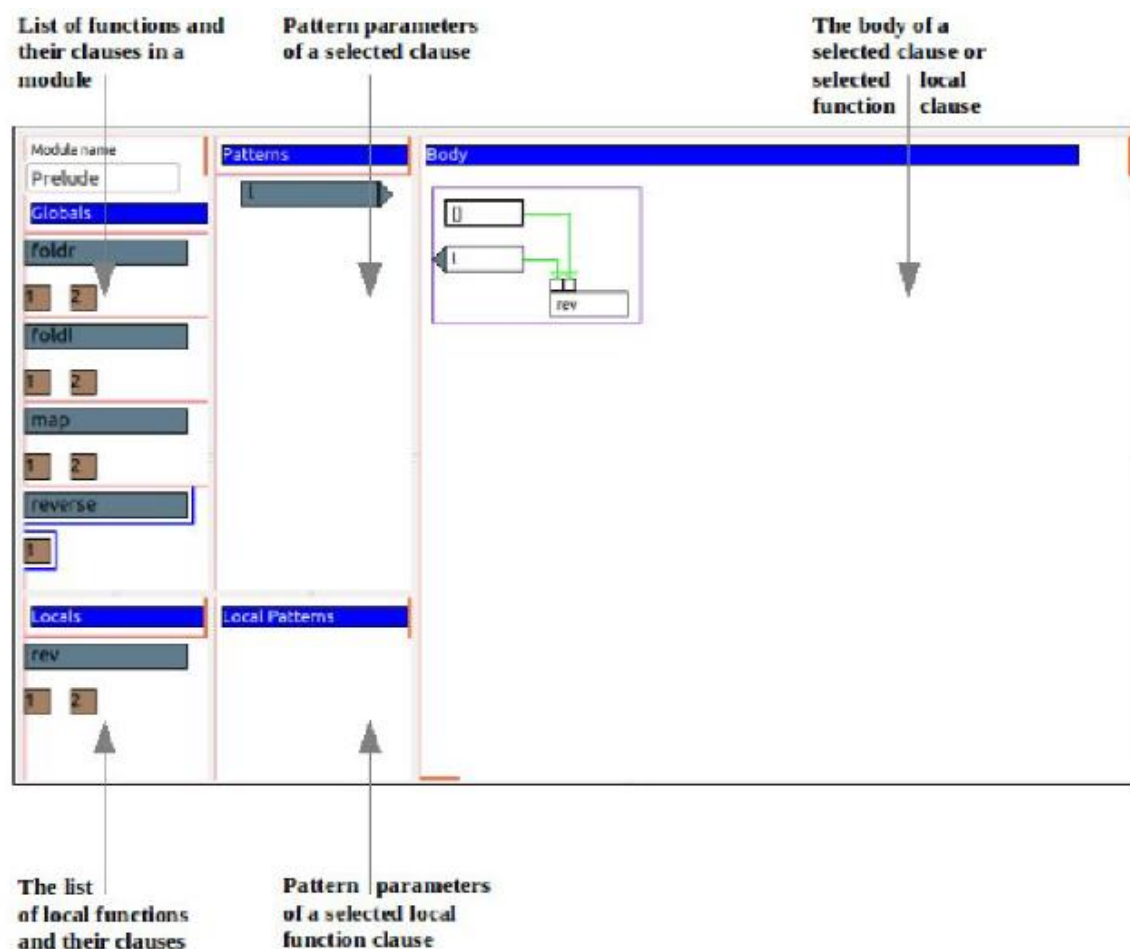


Fig. 1: HASKEU visual display area.

2) Pragmatics: Identifiable, memorable, overall legibility. In figure 2b, parameter variables have a triangle on their right-hand side to suggest to the user that they will be immediately used. This property can then be spotted easily among other parameters (e.g., wild cards (see Figure 2h) and constants (see Figure 2c)). Similarly, if an item used in the function body is a parameter, then it uses a triangle to its left (see Figure 2d).

3) Syntactics: Appearance and movement – modular parts, patterns, shape. This theory is used in figure 2c where any item annotated with 123 on its right side denotes an integer constant. In the same manner, abc denotes a string constant (see Figure 2f), 'c' denotes a character constant (see Figure 2e), T/F denotes a boolean constant (see Figure 2g). These icons are designed to be self-explanatory.

4) Semantics: The underlying item being represented - part versus whole, concrete versus abstract. This level incorporates the functionality of an item: what can be expressed.

5) Dynamics: Receptivity to click - highlighting, combining. In HASKEU, argument slots of a function can be dynamically created to express higher-order language features very effectively. (a) Function name (b) Variable parameter (c) Integer constant parameter (d) Parameter in function body (e) Character constant parameter (f) String constant parameter (g) Boolean constant parameter (h) Wild Card parameter

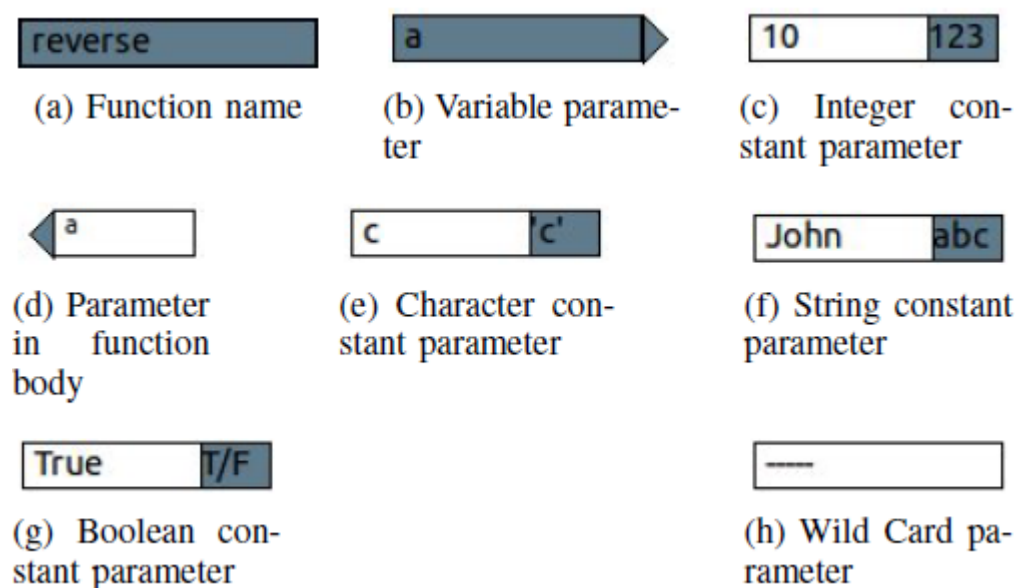


Fig. 2: Pictures of some data fields.

Four standard colours and markings are used in the HASKEU system to attract attention [24]. The use of fewer colours leads to less visual clutter on the screen [25]. They are:

- 1) a blue rectangle to indicate focus on a selection (reverse in Figure 1);
- 2) a purple rectangle to indicate focus on a group of items in scope of an expression as seen in the righthand pane of Figure 1, showing the body of the function clause;
- 3) green lines to show dataflow as seen in the right-hand pane of Figure 1, showing the body of the function clause;
- 4) magenta to denote any unused argument slot.

The HASKEU visual programming system also facilitates the creation of local functions. Only one level of local definition is allowed so that the visual display is not cluttered by nested local functions. In informal observation (four widely used libraries were checked - wxHaskell, Reactive.Banana, Reactive.Banana.WX, haskell.type.exts, and among 240 local functions, only six used nested local definitions) it was noticed that one level of local function can serve many complex problems and so this limitation is not too restrictive in general.

To avoid cluttering the display by repetition of function names, function clauses are numbered in order underneath the function name. Figure 5 on the next page shows the factorial function has two clauses.

C. An Example in Haskell and HASKEU

Below is shown the textual and visual representation of a Haskell function `maxThree` to find the maximum of three numbers. Textual syntax of `maxThree` function:

```
maxThree a b c = max a (max b c)
```

The predefined function `max` computes the maximum of two numbers. `max` function uses the predefined function `cond`. In functional programming, the primary control construct is a function [26], and hence in HASKEU the if-then-else syntax is replaced by a predefined function

```
cond::Bool -> a -> a -> a.
```

The visual representation of the `max` function is given in Figure 3 and the visual representation of `maxThree` function is given in Figure 4. The `maxThree` function is global and defined with a single clause. When the clause is selected, the Pattern pane shows that the clause takes three numbers as its arguments (`a`, `b` and `c` in the above definition). The right-hand pane shows the body of the `maxThree` function as a dataflow graph. It first computes the maximum of arguments `b` and `c`.

The maximum of the last result and argument `a` gives the final result.

From the visual view of `maxThree` function, the flow of data, and scope of function applications (`max`) can be clearly seen. The advantage of such an approach is greater for more complex functions where the flow of data and scope are more easily discernible in a visible than a textual format.

D. The utility of syntax-directed editors

Modern program development typically involves the use of tools that support the programming activity. Some program editors, termed “syntax-directed editors”, only allow syntactically correct programs to be written [27]. When a program is developed textually, expert programmers frequently type incorrect syntax (such as pseudo-code) in intermediate versions of the code, capturing ideas at a high level and then refining details later [28]. Unless the textual editor is a syntax-directed editor, there is no restriction on what text can be typed.

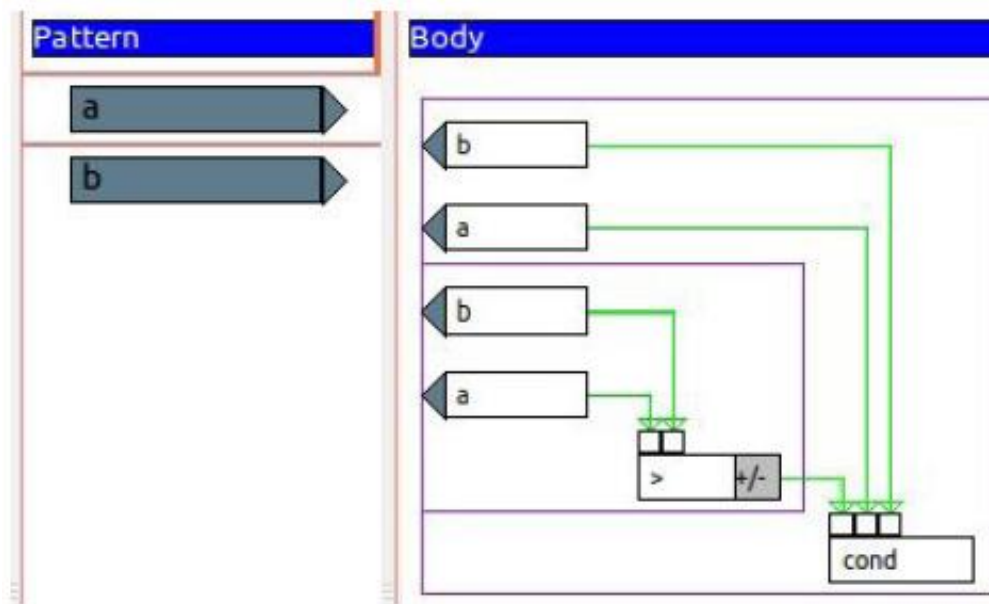


Fig. 3: Visual representation of max function.

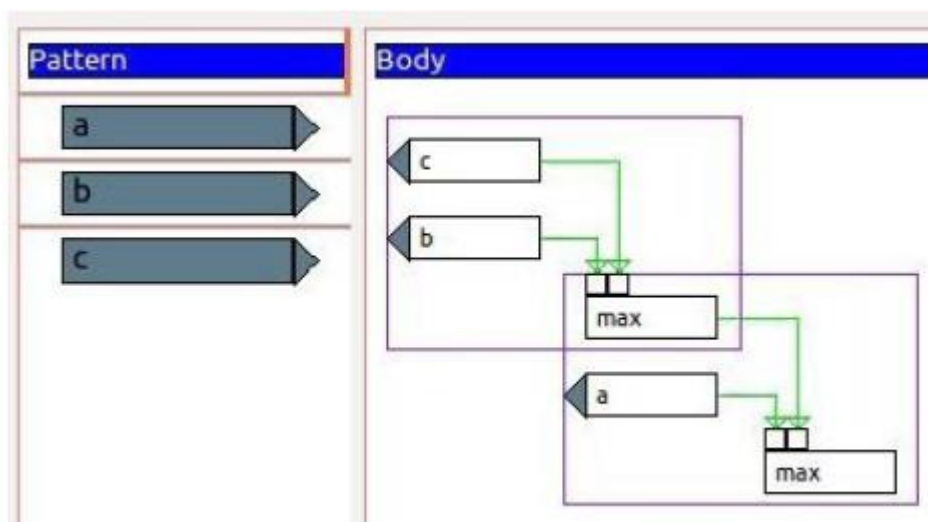


Fig. 4: Visual representation of maxThree function.

Programs developed using a visual notation are more constrained in that the notation captures the syntax of the language exactly, with no extraneous symbols. The development of a visual program requires tools that understand the program's structure and present the programmer with a visual view of such structures [29]. A visual editor can enforce a syntaxdirected way to construct programs [30]. A syntax-directed visual editor maintains an internal model of a program while it is being edited and manipulates it by checking the consistency of the model so that correct syntax is always produced. Such a syntax-directed visual editor is good for learners, as they are presented with a menu of syntactically correct options to develop their programs. Many programming systems aimed at teaching novice programmers, particularly children, use such editors [31].

However, syntax-directed editing is unsuitable when the user wants to radically restructure a program as this necessarily involves breaking up existing structures. The use of a syntaxdirected editor for trivial tasks can be expensive and a reliance on visual editors can add more difficulty for users moving onto more complex, textually-based tools [7]. Research has found that text-based

syntax-directed editors with some ability for “free typing” can be more effective [28]. Many expert programmers find it easier to do some editing by entering text.

Both textual and visual editors have advantages / disadvantage for learners and experts. While visual system can help the learners to learn syntax, a combined system can help experts to develop, debug and maintain their program by switching between the systems effectively. The following section shows how a visual system can help a novice programmer.

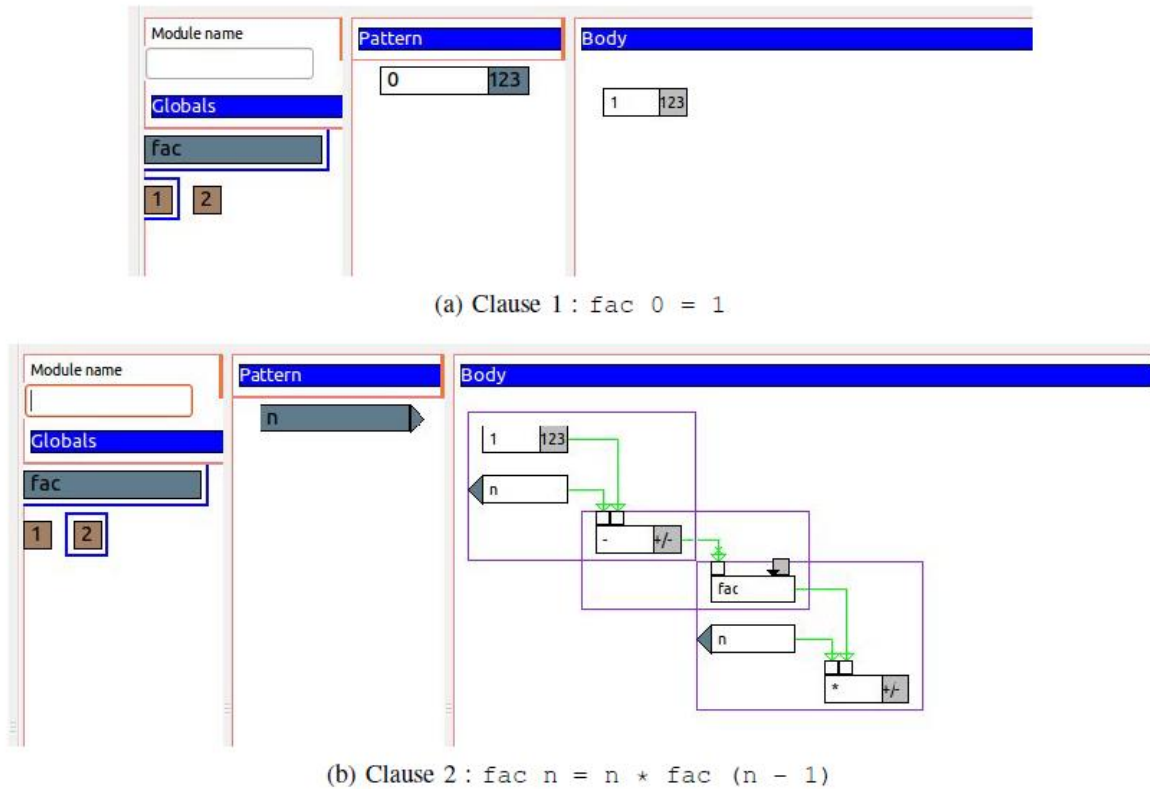


Fig. 5: The visual equivalence of the factorial function in HASKEU.

III. VISUAL IS GOOD FOR LEARNERS

In this section we will concentrate on the ways visual programming supports learners, using the Haskell language:

A. Organization of the visual editor

As demonstrated in Figure 1, in a visual program editor, different parts of a function, module or project can be shown in different panes of the editor. Hence the learning of program development is easier because the visual version gives more information about the program structure, so we know what are global, local and pattern parameters. From the textual syntax of the above function, it is hard to identify the global items, local items, pattern parameters or the items in the function body. The organization of the visual view makes these items easily recognizable.

B. Program Icons

Incorporating graphics or pictures into the programming process adds an interesting and useful dimension [6]. All items in a visual system are displayed as icons for easy recognition. Icons can help

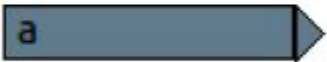

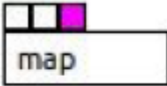

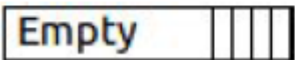
with both syntax and semantics. The table below shows the meaning of the different icons in HASKEU:

The relationship between items is indicated by lines so that the structure of the program and the flow of data through the program are clearly visible.

C. Advantages of Dataflow Graphs

Further to the discussion about representing dataflow and scope of a Haskell program in HASKEU (see Section II-B), the benefits of such approach are summarised here. It has been known for a long time that dataflow and the indenting of block structured programs to give a two-dimensional display

TABLE I: Icons for items in HASKEU

Icon	Description
	parameter variables have a triangle on their right-hand side, to suggest to the user that they will be immediately used, and so that they can be spotted easily among other parameters . This is an instance where icons can help with syntax.
	any item annotated with <i>123</i> on its right side, denotes an integer constant.
	magenta is used to denote any unused argument slot.
	An unobtrusive “!” symbol is shown at the top-right corner of an undefined application.
	An empty list.

are helpful aids in program understanding [32]. The reasons for describing a dataflow visually are the following:

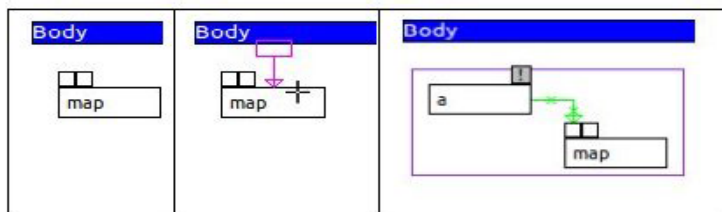
- 1) Dataflow languages sequence program actions by data availability: a node is said to be executable when its arguments are available. The node’s result is then sent to other functions, which need these results as their arguments. This means that a program can be suitably drawn as a directed graph in which each node represents a function and data item flows through a directed arc;
- 2) Smaller dataflow programs can be easily combined into larger programs;
- 3) Graphs present a natural view of the execution of a program;
- 4) By using graphs, a formal meaning can be attributed to components of a program.

Figure 4 shows the body of the maxThree function. A purple rectangle is used to indicate the scope of a group of items in an expression. Green lines ending in an arrow show the travel of direction of data into a function. In this way, the flow of data, and scope of the max function can be clearly seen.

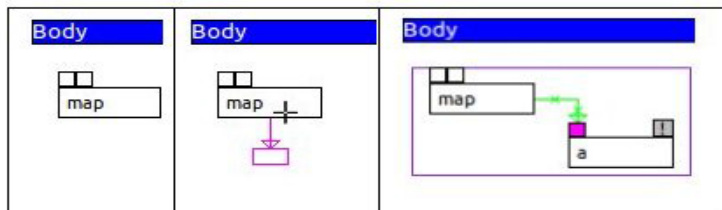
D. Direct-Manipulation

“Direct manipulation” is where program structures are dragged and dropped from a menu to develop a program. It is very popular in Visual Programming Languages [33]. A mouse or touch screen can be used for the direct manipulation of items in a visual programming editor. Hence the programmer has the impression of directly constructing a program rather than having to abstractly design it as in the case of writing a program directly in text. Such direct manipulation can lower the barrier to learning the syntax and semantics of a new programming language, avoid syntax errors by constraining syntax and provide a concrete visual representation of the code, thus reducing memory load.

Direct manipulation techniques are used in the five panes of HASKEU visual editor. Figure 6 illustrates two procedures for adding an argument in HASKEU. A new argument can be added to an existing item as in Figure 6a and an existing item can be added as an argument to a new item as in Figure 6b. If the mouse cursor is positioned at the upper part of an item (in this example, map), then a symbol indicating “add argument” appears (see Figure 6a), and if it is positioned at the lower part of an item, then a symbol indicating “add as argument” appears (see Figure 6b). The argument will be added to the first free argument slot.



(a) Adding an argument to an existing item



(b) Adding an existing item as an argument to a new item

Fig. 6: Different ways of adding an argument.

E. Error Reporting

Even with syntax-directed editing using direct manipulation, it is still possible to develop programs with errors. Error reporting can be shown visually to aid learners to understand and locate an error more precisely. Below are some examples of how errors are indicated in a visual environment.

CASE 1: Consider the following small function fn:

```
fn a = map 'h'
```

The map function takes a function as its first argument but we are trying to add a character constant h to its first argument. The purpose of the map function was given in Section II-A earlier. The Glasgow Haskell compiler gives detailed type error messages textually as below:

TestProg.hs:1:12:

Couldn't match expected type 'a0 ! b0'

with actual type 'Char'

In the first argument of 'map', namely 'h'

In an equation for 'fn': fn a = map 'h'

The visual error report in HASKEU can express the above error details in a single view (see Figure 7).

Cross marks in the dataflow arc in the function body indicate type errors and both end-points contain the type information as tooltip text.

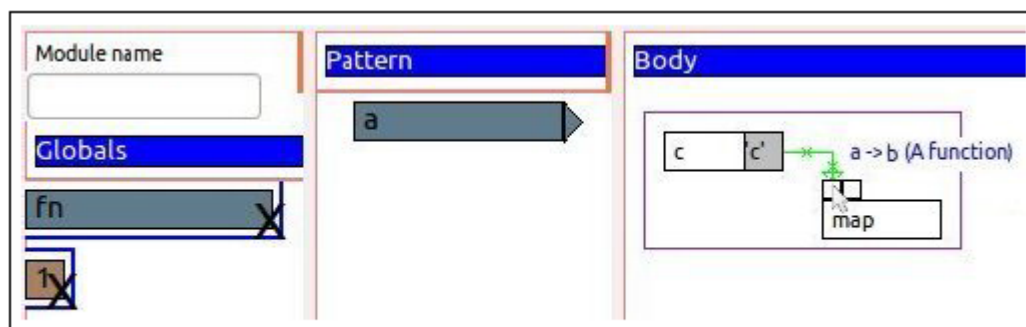


Fig. 7: Error - type mismatch.

CASE 2: Consider another small Haskell function:

```
fn a b k = map a b k
```

The map function can take up to two arguments whereas in the above function we are trying to add three arguments (a, b and k). So, the Glasgow Haskell compiler says after compilation:

TestProg.hs:1:12:

The function 'map' is applied to three arguments, but its type '(a0->b0) ! [a0] ! [b0]' has only two

In the expression: map a b k In an equation for 'fn': fn a b k = map a b k

This is quite complex for novice programmers to understand and they can find debugging a difficult experience. In contrast, the visual system in HASKEU shows a magenta argument slot (see Figure 8) as soon as any unused argument has been added during program editing. The learner programmer can clearly see the problem as soon as it arises.

These two sections have described and demonstrated the value of using a visual programming system for learner programmers. Visual support can help in constructing correct programs from the start by direct manipulation of icons. The scope of each arguments is clearly indicated and the flow of data is intuitively represented as a graph. Errors are clearly visible. The next section will look at how such a system can aid expert programmers.

IV. TEXTUAL IS GOOD FOR EXPERTS

The use of a visual editor for trivial tasks can be expensive and it can add more difficulty for expert users learning more complex tools [7]. In Hubbell's study, expert users found the syntax-directed

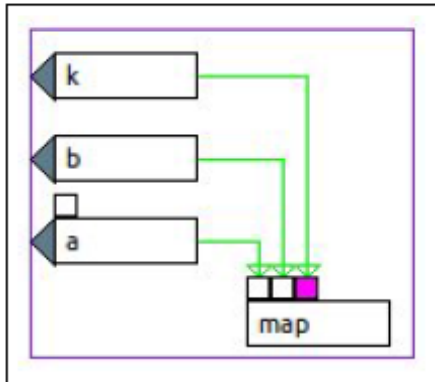


Fig. 8: Error - unused argument.

visual editor too laborious and irritating to use for inputting programs or for performing simple editing tasks. Many programmers find it easier to do some editing by entering text (e.g., inputting infix expressions, changing an if-statement to a while-statement). In a visual editor, such changes involve a sequence of operations (e.g., select a menu, select a tree node, restriction in editing for syntactically incorrect input, press enter when finished) which an expert can replace by typing the equivalent text very quickly. Also, in a complex tool, selecting and learning the right menu can take time.

Research has found that syntax-directed editors with some ability for “free typing” can be more effective than a strict syntax-directed editor [28]. The problem can be clearly understood by looking at all the steps necessary (given below) to create a small function `maxThree` visually in HASKEU, whereas textually a confident programmer can just type the program. However, once the code has been entered textually, in HASKEU it is possible to see the equivalent visual structure. This gives the programmer the opportunity to view the code in two formats, each of which can provide different insights.

Steps to create `maxthree` function visually:

- 1) Select the “Add new function” menu and click on the global pane to add a new function;
- 2) Edit the function name to change it to “`maxThree`”;
- 3) Select the “Pattern variable” menu and click three times on the global menu to add three pattern parameters;
- 4) Change all the three pattern parameter names to desired names;
- 5) Select the “Add new function application” menu and click on the body pane to add a new function application;
- 6) Edit the function application name to change it to “`max`”;
- 7) Drag and drop two pattern parameters to add them as an argument of application “`max`”;
- 8) Select the “Add new function application” menu and click on lower part of the previously added function application “`max`” in the body pane to add it as an argument of the new application;
- 9) Change the new function application name to “`max`” too;

10) Drag and drop the rest pattern parameter to add it as an argument of new “max”.

This approach may be supportive for learner programmers but frustrating for more expert programmers. However, within HASKEU, it is also possible to type in Haskell as text (see Figure

9)). The equivalent visual version will also be displayed if the code is syntactically correct. If not, there will be immediate feedback as no visual equivalent will be shown. This means that expert programmers can develop their programs textually but still take advantage of the visual feedback if they find that format useful later, as described in the next section.



Fig. 9: Typing in maxThree in textual format in HASKEU.

V. VISUAL IS GOOD FOR EXPERTS

Research has shown that visual programming is also good for expert programmers, not only for learners [34]. A visual representation of a program can provide a higher-level description of the desired actions (reducing syntax) and hence make the programming task easier, even for experienced programmers.

This is very true during debugging, where visuals can be used to represent more information about the program (such as the type information in the case of functional programming) than is possible with a purely textual interface. In HASKEU, the type information of an individual argument is shown with descriptions when the mouse pointer hovers over that argument slot and the whole type information of an application is shown when the mouse pointer is over the application box (see Figure 10).

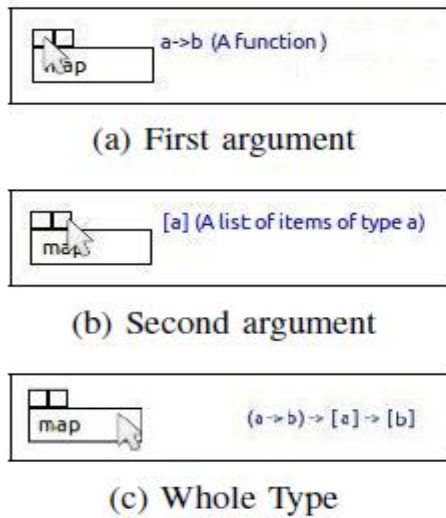


Fig. 10: Type representation in map application.

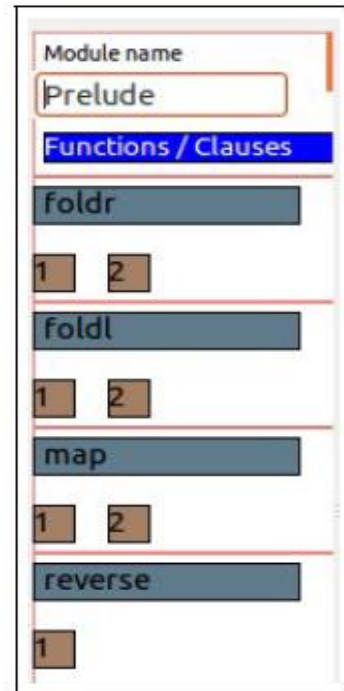


Fig. 11: Compact view of a module.

The purpose of visual programming is not only to help users to learn a new programming language but also to help them with the development and maintenance of programs. These processes just mentioned have been found to be timeconsuming activities in practice. While maintaining a huge program, a developer does not want to read the entire code.

Hence, they can benefit from the visualization of the program which can help them to get a rapid understanding of the source code and its structure [9]. In HASKEU, a compact view of a module can be seen and it can help programmers to maintain their program. Figure 11 shows a compact view of a small module including four functions `foldr`, `foldl`, `map` and `reverse`. Function clauses are numbered in order underneath the function name.

VI. USABILITY EXPERIMENT

A usability test was conducted to evaluate the system by testing it on end-users to obtain direct feedback. The test was designed as follows: a programming exercise was devised; instructions to complete this exercise were piloted; feedback enabled the instructions to be refined and four users undertook the exercise. Two were skilled programmers, though not with functional programming, and two had never programmed before. A questionnaire measured the time taken to complete the test, accuracy (correctness of resulting program), and emotional response (how the user felt whilst working in the program). The results were compared with the usability goals. They showed that the visual system was a good starting point for novice users to learn functional programming, and the textual system was found helpful too as their expertise increased. None of the participants found the textual error reporting very useful, but they appreciated seeing the errors visually. One of the programmers liked the prevention of syntax errors while editing in the visual system. Both liked having the option to use both the textual and visual systems together. Obviously, more extensive

testing is needed but these preliminary results are promising. It was noticed while the test was being conducted by one of the programmers that he was comfortable using the both textual and visual system simultaneously.

VII. RELATED WORK

Many programming systems such as Dreamweaver and Visual Basic use tandem program editing, but their visual system is only a GUI builder (to develop the GUI interface of the program being developed) not an actual visual programming system. LabVIEW has a pure visual programming environment, but there is no way to directly convert this to any textual language. The HOPS (Higher Object Programming System) is a text-based syntax-directed interactive term graph programming system for functional programming and is not a visual programming system by any means [35]. Visual Haskell (Reekie, 1994), a visual programming system for Haskell can be found in the literature and it was more of a visualization tool than a visual programming system. A recent, nominally visual, programming system to support Haskell is also called Visual Haskell [36]. It is a Haskell development system to support textual programs, rather than visual ones. No other systems have been discovered, particularly for the functional programming paradigm, which provide support for the parallel approach of textual and visual development of programs in tandem as HASKEU does.

VIII. FUTURE WORK

The visual programming system in HASKEU is at an early stage of development and further work is needed to produce a more comprehensive system. Haskell has a vast syntax and only a small portion is covered by the visual representation in this project. However, this small portion (function, clause, pattern matching, higher-order function, type representation, currying, primary control construct, partial application and local functions) can be seen as the building blocks of functional programming. HASKEU is intended so that the novice programmer will begin with the visual view, perhaps later moving on to the textual view. Preliminary tests have shown that novice programmers find the visual programming system in HASKEU a useful starting point upon the learning ladder of functional programming. Experienced programmers also had positive comments about the usefulness of having both visual and textual representations of a program. More comprehensive testing is needed to help to improve and refine the system. Future goals are to develop a more comprehensive representation of Haskell within the visual display, which may be appreciated by more expert Haskell programmers. This would involve designing generalized icons for programming constructs such as case, guard, class, instance, lambda expression in the HASKEU visual system.

IX. CONCLUSION

Visual programming may not completely replace textual programming but it can enrich the textual view. The two forms can support each other in the learning, development and maintenance activities of programming. It is expected that the learner will begin with the visual view, perhaps later moving on to the textual view as it allows them to perform some useful visual programs with a small investment of time and then go on to more advanced levels of understanding textually when they are ready. It was found that non-programmers can write quite complex programs in visual programming systems with little training. Also, experts can get help from visual system when it comes to debugging and maintaining the program.

Both learners and experts can get help from visual error reporting. Experts like textual systems too because they allow some editing operations to be carried out very efficiently. In conclusion, we recommend that a programming system with textual and visual representations in tandem can make program development easier and quicker for novices and experts alike.

ACKNOWLEDGMENT

The first author is very thankful to Dr David Wakeling for his invaluable guidance throughout the research.

REFERENCES

- [1] G. Harvey. (2015) Gnu emacs home page. 51 Franklin St, Fifth Floor, Boston, MA 02110, USA. [Online]. Available: <https://www.gnu.org/software/emacs/> accessed 6th October 2015
- [2] Oracle Corporation. (2015) Netbeans IDE. [Online]. Available: <https://netbeans.org/> accessed 6th October 2015
- [3] K. Zhang, Visual languages and applications. Netherlands: Springer, 2007.
- [4] P. T. Cox and P. K. Nicholson, "Unification of Arrays in Spreadsheets with Logic Programming," in PADL, ser. Lecture Notes in Computer Science, P. Hudak and D. S. Warren, Eds., vol. 4902. Netherlands: Springer, 2008, pp. 100–115.
- [5] P. T. Cox and S. Gauvin, "Controlled Dataflow Visual Programming Languages," in Proceedings of the 2011 Visual Information Communication - International Symposium, ser. VINCI '11. New York, NY, USA: ACM, 2011, pp. 9:1–9:10.
- [6] N. C. Shu, Ed., Visual programming. New York, NY, USA: Van Nostrand Reinhold Co., 1988.
- [7] T. J. Hubbell, D. D. Langan, and T. F. Hain, "A Voice-activated Syntaxdirected Editor for Manually Disabled Programmers," in Proceedings of the 8th International ACM SIGACCESS Conference on Computers and Accessibility, ser. Assets '06. New York, NY, USA: ACM, 2006, pp. 205–212.
- [8] B. R. C. Marques, S. P. Levitt, and K. J. Nixon, "Software visualisation through video games," in Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference, ser. SAICSIT '12. New York, NY, USA: ACM, 2012, pp. 206–215.
- [9] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in Proceedings of the 2010 17th Working Conference on Reverse Engineering, ser. WCRE '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 35–44.
- [10] A. Alam, "A Programming System for End-User Functional Programming," Ph.D. dissertation, University of Gloucestershire, Aug 2015.
- [11] T. R. G. Green, "Programming languages as information structures," in Psychology of programming, J.-M. Hoc, T. R. G. Green, R. Samurcay, and D. J. Gilmore, Eds. London: Academic Press, 1990.
- [12] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler, "A history of haskell: Being lazy with class," in Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages, ser. HOPL III. New York, NY, USA: ACM, 2007, pp. 12–1–12–55. [Online]. Available: <http://doi.acm.org/10.1145/1238844.1238856>
- [13] A. Church, "A set of postulates for the foundation of logic part ii," Annals of Mathematics, vol. 34, no. 2, pp. 839–864, 1933.
- [14] S. P. Jones and S. Singh, "A Tutorial on Parallel and Concurrent Programming in Haskell," in Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Verlag, 2008.

- [15] R. Warburton, *Java 8 Lambdas: Pragmatic Functional Programming*, 1st ed. Sebastopol, CA, USA: O'Reilly Media, Inc., 2014.
- [16] Microsoft. (2015) C# language specification. [Online]. Available: <https://msdn.microsoft.com/en-us/library/ms228593.aspx>, accessed 6th October 2015
- [17] R. Bird and P. Wadler, *An introduction to functional programming*. Hertfordshire, UK, UK: Prentice Hall International (UK) Ltd., 1988.
- [18] S. K. Card, A. Newell, and T. P. Moran, *The Psychology of Human-Computer Interaction*. Hillsdale, NJ, USA: L. Erlbaum Associates Inc., 1983.
- [19] B. Shneiderman and C. Plaisant, *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, 4th ed. Reading, MA, USA: Pearson Addison Wesley, 2004.
- [20] G. W. Johnson, *LabVIEW Graphical Programming: Practical Applications in Instrumentation and Control*, 2nd ed. Berkshire, UK: McGraw-Hill Education, 1997.
- [21] K. Lodding, "Iconic Interfacing," *IEEE Computer Graphics and Applications*, vol. 3, pp. 11–20, 1983.
- [22] R. Korfhage, "Query Enhancement by User Profiles," in *SIGIR*, 1984, pp. 111–121.
- [23] A. Marcus, *Graphic design for electronic documents and user interfaces*. New York, NY, USA: ACM, 1992.
- [24] C. D. Wickens and J. G. Hollands, *Engineering Psychology and Human Performance*, 3rd ed. Englewood Cliffs, NJ, USA: Prentice Hall, Sep. 1999.
- [25] I. Apple Computer, *Macintosh Human Interface Guidelines*. Boston, MA, USA: Addison-Wesley Publishing Company, 1992.
- [26] R. Burstall, "Christopher Strachey—Understanding Programming Languages," *Higher-Order and Symbolic Computation*, vol. 13, no. 1-2, pp. 51–55, Apr. 2000.
- [27] Y. Bai, "The design and realization of a common syntax-directed editing system." in *IRI*, 2003, pp. 85–92.
- [28] R. C. Waters, "The Programmer's Apprentice: Knowledge Based Program Editing," in *Interactive Programming Environments*, D. R. Barstow, H. E. Shrobe, and E. Sandewall, Eds. New York, NY, USA: McGraw-Hill, 1984, pp. 464–486.
- [29] F. Arefi, C. E. Hughes, and D. A. Workman, "Automatically Generating Visual Syntax-directed Editors," *Communications ACM*, vol. 33, no. 3, pp. 349–360, Mar. 1990.
- [30] G. Costagliola, A. D. Lucia, S. Orefice, and G. Polese, "A classification framework to support the design of visual languages." *Journal of Visual Languages and Computing*, vol. 13, no. 6, pp. 573–600, 2002.
- [31] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y. Kafai, "Scratch: Programming for all," *Communications ACM*, vol. 52, no. 11, pp. 60–67, Nov. 2009.

- [32] J. M. Smith and D. C. P. Smith, "Database abstractions: Aggregation and generalization." ACM Transactions Database Systems, no. 2, pp. 105–133, 1977.
- [33] B. Shneiderman, "Direct Manipulation: A Step Beyond Programming Languages," Computer, vol. 16, no. 8, pp. 57–69, Aug. 1983.
- [34] B. A. Myers, "Taxonomies of visual programming and program visualization," Journal of Visual Languages and Computing, vol. 1, no. 1, pp. 97–123, Mar. 1990.
- [35] S. West and W. Kahl, "A Generic Graph Transformation, Visualisation, and Editing Framework in Haskell," ECEASST, vol. 18, 2009.
- [36] K. Angelov and S. Marlow, "Visual Haskell: A Full-featured Haskell Development Environment," in Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell, ser. Haskell '05. New York, NY, USA: ACM, 2005, pp. 5–16.