

**A PROGRAMMING SYSTEM FOR
END-USER FUNCTIONAL PROGRAMMING**

ABU SALEH MOHAMMED MAHBUBUL ALAM

A thesis submitted to
The University of Gloucestershire
in accordance with the requirements of the degree of
Doctor of Philosophy
in the Faculty of Media, Arts and Technology

July, 2015

Abstract

This research involves the construction of a programming system, HASKEU, to support end-user programming in a purely functional programming language. An end-user programmer is someone who may program a computer to get their job done, but has no interest in becoming a computer programmer. A purely functional programming language is one that does not require the expression of statement sequencing or variable updating. The end-user is offered two views of their functional program. The primary view is a visual one, in which the program is presented as a collection of boxes (representing processes) and lines (representing dataflow). The secondary view is a textual one, in which the program is presented as a collection of written function definitions. It is expected that the end-user programmer will begin with the visual view, perhaps later moving on to the textual view. The task of the programming system is to ensure that the visual and textual views are kept consistent as the program is constructed. The foundation of the programming system is a implementation of the Model-View-Controller (MVC) design pattern as a reactive program using the elegant Functional Reactive Programming (FRP) framework. Human-Computer Interaction (HCI) principles and methods are considered in all design decisions. A usability study was made to find out the effectiveness of the new system.

Keywords. Functional Programming, Visual Programming, End-User Programming, Visual Dataflow Language, Usability, Human-Computer Interaction, Model-View-Controller, Functional Reactive Programming, Programming Systems, HASKEU

Author's Declaration

I declare that the work in this thesis was carried out in accordance with the regulations of the University of Gloucestershire and is original except where indicated by specific reference in the text. No part of the thesis has been submitted as part of any other academic award. The thesis has not been presented to any other education institution in the United Kingdom or overseas. Any views expressed in the thesis are those of the author and in no way represent those of the University.

SignedDate

Acknowledgements

I have been very fortunate to have both Dr. David Wakeling and Dr. Vicky Bush as my first supervisor. Their guidance and advice throughout this research has been invaluable.

I would like to express my heartfelt appreciation and thanks to my supervisor Dr. David Wakeling who has been a tremendous mentor for me. I would like to thank him for his insight and guidance throughout my Ph.D. work and for encouraging my research and allowing me to grow as a research scientist. He helped me with any concern related to my study. Many thanks. It was sad and heartbreaking for me when he had to leave the University after providing a three and half years of supervision. He has continued to support me by reading and commenting on this thesis.

I am also very thankful for the feedback and support of Dr. Vicky Bush who became my first supervisor and provided my immediate support after Dr. Wakeling left. I would like to thank her for her brilliant comments and suggestions. Her questions and comments and her careful reading of this thesis and her vast knowledge have been very valuable. I am very grateful for the many hours that she spent discussing and critiquing my thesis. This thesis would not have been possible without her help, support and patience. Outside of this research, I have also been learning the qualities of a good academic from her and I wish I had a little pinch of her excellence. A heartfelt thanks go to her too.

I would like to thank my second supervisor Dr. Shujun Zhang for his support and encouragement. At many stages in the course of this research I benefited from his advice.

A very special thanks to a special person, outside the faculty, Dr. Robin Reeves, for reading and commenting, and helping in the many phases of my writing. A tireless, sporting and a man of great knowledge, who read the thesis patiently, and his comments were particularly useful for the end-user section. It would not have been possible to write this thesis without the help and support of this kind person.

Many thanks to all of the participants in my usability study. Thanks to many additional friends who have helped me in various ways.

Especially, I would like to express my gratitude to all my family members for their support and encouragement. Words cannot express how grateful I am to them for all of their sacrifices that they have made on my behalf. I dedicate this dissertation to all my family members.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Research Statement	2
1.3	Who are the End-Users?	3
1.4	What are End-Users' Problems?	3
1.5	Choice of Functional Language	10
1.6	Design Inspiration	11
1.7	Research Aims	14
1.8	Research Methodology	15
1.9	Design, Implementation and Testing Approach	17
1.10	Contribution	22
2	Background and Literature Review	24
2.1	End-user programmers	24
2.2	Functional Programming Languages	27
2.2.1	Motivation	27
2.2.2	A Functional Language : Haskell	27

2.2.3	An Example Haskell Program	32
2.2.4	Other Functional Programming Languages	34
2.3	Support for Language Learning: Syntax-Directed Editor	35
2.4	Support for Language Learning: Visual Programming	39
2.4.1	Motivation	39
2.4.2	Visual Programming: Overview	40
2.4.3	Visual Programming: Classification	43
2.4.4	An Early Visual Language : Prograph	50
2.4.5	An Example Prograph Program	52
2.4.6	A Useful Visual Language : LabVIEW	54
2.4.7	An Example LabVIEW Program	57
2.4.8	Other Visual Programming Languages	59
2.5	Visual Functional Programming Languages	60
2.5.1	Motivation	60
2.5.2	A Visual Functional Programming Language : Visual Haskell	61
2.5.3	An Example Visual Haskell Program	65
2.5.4	Other Visual Functional Programming Languages	66
2.6	Conclusion	68
3	The Model View Controller as a Functional Reactive Program	70
3.1	Introduction	70
3.2	The MVC Design Pattern	72
3.3	The FRP Framework	74

3.3.1	Monad and Monadic IO	75
3.3.2	The FRP	83
3.4	An implementation of the “MVC as FRP” framework	88
3.5	An Example	96
3.6	Related Work	100
3.7	Conclusion	101
4	The Design of HASKEU	102
4.1	Visual Programming	104
4.1.1	Organization	104
4.1.2	Content	107
4.1.3	Dataflow and Scope	113
4.1.4	Direct-Manipulation	114
4.1.5	One Function Per Page	118
4.2	Exploratory Programming	118
4.2.1	Error Reporting	119
4.2.2	Infinite Undo	126
4.3	Textual Programming	129
4.4	Design of Concepts	130
4.4.1	Synchronization between Textual and Visual view	132
4.4.2	The choice of a tree structure	136
4.4.3	Higher-order functions in visual view	136
4.5	Conclusion	138

5	Implementation	141
5.1	The Model	143
5.2	The Controllers	149
5.2.1	Visual Layout	151
5.2.2	Type Management	157
5.2.3	Infinite Redo/Undo	161
5.2.4	To Select an Item	164
5.2.5	To Add a New Item	165
5.2.6	To Delete/Edit an Item	168
5.2.7	To Validate Syntax Error	170
5.2.8	To Adjust wxHaskell Events	171
5.3	The Views	173
5.3.1	To Draw the Shape Tree	173
5.3.2	To Show the Annotation Text Editor	175
5.3.3	To Calculate Row and Column of a Text Control from Insertion Point	176
5.3.4	To Enable/Disable Redo/Undo Buttons	177
5.3.5	To Add More wxHaskell Attributes	178
5.3.6	Difficulties and Achievements	178
6	Usability Experiment and Result	180
6.1	Experiment	180
6.1.1	User Manual	181
6.1.2	Selection of End-Users	181

6.1.3	To Devise a Programming Exercise	182
6.1.4	Instructions for the Exercise	183
6.1.5	Usability Goals	186
6.1.6	Experiment Process	187
6.2	Result	188
6.2.1	Performance Comparison Using Quantitative Data . . .	188
6.2.2	Suggestion from the Qualitative Data	193
6.3	Conclusion	194
7	Conclusions	196
7.1	Achievements	196
7.2	Limitations and Suggested Future Developments	198
	Appendices	203
A	Appropriateness of Using Software Analysis and Design, and UML Diagrams for Functional Programs	204
A.1	Appropriateness of using software analysis and design	205
A.2	Appropriateness of using UML	208
B	Spaghetti Code	217
C	User Manual	220
D	Usability Test - Questionnaire and Consent Form	260
D.1	Usability Test - Questionnaire	261
D.2	Usability Test - Consent Form	264

E	The Library API - MVC_WX	266
F	Source Code of Functor, Applicative and Monad	278

List of Figures

1.1	The spiral development.	19
2.1	An electrical circuit.	33
2.2	Example programs in HOPS.	37
2.3	Splicing of two graphs.	46
2.4	Example of a program in extended HI-VISUAL.	48
2.5	Find the names of employees who work in the toy department and earn more than 10000.	49
2.6	Some Prograph constructs.	51
2.7	Prograph - Reversing order of a list.	53
2.8	A non-empty list icon.	54
2.9	The LabVIEW control palette.	55
2.10	The LabVIEW function palette.	56
2.11	The LabVIEW tool palette.	56
2.12	Front panel for the series circuit example in Figure 2.1.	57
2.13	Block diagram for the series circuit example in Figure 2.1.	58
2.14	The Visual Haskell definition of <code>map</code>	62

2.15	Illustrating type annotation in Visual Haskell: i) (a,b); ii) [[a]]; iii) Stream (Vector a).	62
2.16	a) Source pads; b) Sink pads.	63
2.17	a) A data arc; b) A binding arc; c) Attached objects.	63
2.18	Shared object.	64
2.19	Currying.	65
2.20	Illustrating the series circuit (in Figure 2.1) implementation in Visual Haskell.	66
2.21	Commonly Used Arrow Combinators.	68
3.1	MVC Example - a volume controller system	73
3.2	The volume control system using the MVC-FRP architecture.	99
4.1	The visual display area.	106
4.2	Pictures of some data fields.	107
4.3	Compact view of a module.	109
4.4	Parameter icons.	110
4.5	Expression icons.	111
4.6	Data display of <code>reverse</code>	112
4.7	Showing dataflow of function <code>max</code>	113
4.8	Adding a new function.	115
4.9	Adding a new parameter.	116
4.10	Adding an argument.	117
4.11	Adding an annotation.	117
4.12	Extended window to show error textually.	121

4.13	Type representation in <code>map</code> application.	121
4.14	Overview of errors.	122
4.15	Error - type mismatch.	123
4.16	Showing all errors.	124
4.17	Error - unification.	125
4.18	Error - undefined function.	125
4.19	Error - unused argument.	126
4.20	Changes propagate from textual to visual.	129
4.21	Changes propagate from visual to textual.	130
4.22	System views before and after a syntax error.	134
4.23	User and system defined textual layout of a program.	135
4.24	Tree vs DAG.	137
4.25	Higher-order functions in HASKEU visual view.	139
5.1	General outline of the MVC layout in the system.	144
5.2	Tree structure and dataflow graph of expression <code>map ((+) a) b</code>	155
5.3	The redo/undo stack.	162
5.4	Heap profile to evaluate infinite redo/undo.	163
6.1	Performance of all participants - visually vs textually.	189
6.2	Transition rate from one system to the other of two groups.	191
6.3	Transition ability of the system.	192
6.4	Performance difference rate between two groups.	193
A.1	Class diagram of user login system.	213

A.2	Sequence diagram of user login action.	215
B.1	Spaghetti Code in LabVIEW (Carr, 2011)	218
B.2	A illustration of Spaghetti Code in early HASKEU.	219

List of Tables

6.2	The completion times (in minutes) by the participants using the both textual and visual systems	189
-----	--	-----

Chapter 1

Introduction

This thesis demonstrates an experiment to implement a novel programming system for end-user functional programming. This end-user programming system was developed to support both visual and textual programming, aiming to allow end-users 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.

1.1 Motivation

The aim of this thesis is to make it easier for end-users to learn how to use a general purpose functional programming system. The new visual system produced for this thesis supports the current textual syntax so that switching between the two will be easy. Hopefully, the new system will eliminate some of the harder syntax in functional programming and make learning the

Haskell type system easier for them at a later stage. There are some interesting areas of functional programming which may attract end-users to attempt to learn functional programming languages. Researchers have long maintained that computer programs will be developed and easily maintained in functional programming languages. Unlike traditional imperative programming languages, functional programming languages have no notion of sequence or state (Hughes, 1989). A function call computes its result without any side-effects, because variables, once given a value, can never be changed, which eliminates a major source of bugs. A functional programmer does not need to specify the flow of control and an expression can be evaluated at any time. While conventional programming languages place limits on the way that control theory, hybrid systems, vision, artificial intelligence, and human-computer interaction can be expressed, functional programming languages push back these limits (Hughes, 1989).

1.2 Research Statement

It is feasible to develop an end-user functional programming system that consists of a visual programming system and a textual programming system and for the end-user to have a smooth transition between the two, particularly as the end-users' programming expertise improves and increases. This end-user functional programming system can be implemented in a functional paradigm.

1.3 Who are the End-Users?

The target audience for this new programming system are end-users such as chemists, librarians, teachers, architects, and accountants, who need to use their computers for calculation purposes and who intend to use their computer seriously. They are not “casual”, “novice”, or “naive” computer users; and they don’t want to be a professional programmer (Nardi, 1993). The key difference between professional programmers and end-users is that professional programmers write programming as their primary occupation, and end-users write programs as just one of a range of tools to achieve a particular purpose. Many end-users use computers daily, at least for a certain period of intensive work on a project, and the development of this end-user programming system is targeted at these users. Instead of training large numbers of professional programmers, end-user computing is becoming the major trend. It has become obvious that a radical departure from traditional programming is necessary if programming is to be made more accessible to a large population (Shu, 1988).

1.4 What are End-Users’ Problems?

Programmers who are already experienced with general programming constructs such as types and recursion who have learnt them in imperative languages, still find it difficult to use them in functional languages (Chambers et al., 2012; Segal, 1994; Joosten et al., 2008; Ebrahimi, 1994; Lau et al., 1994).

The end-users have the same difficulties. As well as studies showing the main difficulties of learning a functional programming language, there have also been observational studies and investigations about what information sources people use when they encounter these difficulties. These studies have also shown how effective the different information sources are in enabling them to overcome problems. It was observed that people who were learning to program frequently referred to information sources that did not help them to successfully overcome their problems (Chambers et al., 2012). These results helped this research to analyze the learning barriers of functional programming. The following barriers to learning functional programming have been identified as well as the barriers to learning experienced by end-users. In this thesis, the opportunities of supporting the learning process in a more effective way by using visual programming techniques will be revealed and shown in more detail in Chapter 4.

- People commonly encounter several conceptual difficulties when learning functional languages. In particular, it was found in studies that they struggle to implement recursive functions (Segal, 1994), iteration (Joosten et al., 2008), and nested operations (Ebrahimi, 1994) while using functional languages. Recursive application, clauses, patterns and local functions are the features of functional languages which enable recursive functions, iteration and nested operations to be implemented. The visual system in this research allows these features to be illustrated

and can be used to assist with the editing (see sections 4.1.2 to 4.1.4) .

- While doing programming tasks, a novice programmer frequently referred to information sources, particularly for overcoming compiling errors encountered when passing functions as arguments to other functions. Learners can easily be overwhelmed by the concept of higher-order functions. This concept had therefore to be included after their introductory course (Chakravarty and Keller, 2004). The visual system of this research provides support for understanding and implementing higher order functions (see sections 4.2.1 and 4.4.3) .

- Functional languages also present challenges to learners such as the difficulty of understanding

(a) the type of functional expressions (Joosten et al., 2008). (To see how support is provided in representing types by the visual system of this research to represent types, see Section 4.2.1);

(b) the meaning of error messages (Joosten et al., 2008). (Go to Section 4.2.1 to see the support provided).

There is existing research which has focused on developing some innovative integrated development environments to assist learners to overcome these problems such as DrScheme for the functional programming language Scheme (Findler et al., 2002).

DrScheme is designed to catch typical student mistakes and explain them

in terms that the students understand. DrScheme is a graphics-enriched editor which includes a stepper, a context-sensitive syntax checker, and a static debugger. The stepper is useful for explaining the semantics of linguistic facilities and for studying the behaviour of small programs. The syntax checker changes the font of keywords and the text color in the place where the syntax error had occurred. The static debugger provides type inferences which are selectively overlaid on the program text.

The stepper enables students to make a program to calculate a value in a series of steps analogous to BODMAS calculations in secondary school algebra. The stepper tool has been found useful particularly for those students who prefer to learn by generalizing from examples, rather than working directly from an abstract model. DrScheme's syntax checker helps programmers to understand the syntactic and lexical structure of their programs and has also been found very useful by beginner programmers. The static debugger tool of DrScheme has been found very useful for programmers to perform type inference and to mark potential errors.

DrScheme has important features (syntax and semantic error checking, and error reporting for functional programming) and they have been implemented to support textual programming. These same features have been implemented in the visual system of this research. Some features such as a stepper may be implemented in future research.

- Apart from these barriers to learning functional languages, a study (Ko et al., 2004) has also found six barriers to end-users' learning programming systems. These are

1. Design barriers - the inherent cognitive complications of a programming problem, which occur in using the correct notation to represent a solution (i.e. words, diagrams, code). For example, a learner working on a program so that it sorts the names in a list into reverse-alphabetical order, might be unable to conceive a systematic way to sort the names. His/ her best solution might be to just keep moving the names until they look right! The declarative nature of functional programming and the ability to show program flow visually may help programmers to overcome design barriers (Burnett et al., 2001). The visual system in this research helps to illustrate the declarative function features and program flow (see Sections sections 4.1.2 and 4.1.3)

2. Selection barriers - the problem of finding what programming interfaces are available and what can be used to achieve a particular behaviour. For example, a learner working on designing an alarm clock may find difficulties in using a library function to get current time, to select how store alarm time (globally or locally) and then to compare it with the global/local alarm time variable. In functional programming languages, the alarm time needs to pass through functions as argument, and the visual system in this research shows a

function icon with its argument slots and it shows the type information of each argument. This is how visual functional programming can aid to overcome selection barrier.

3. Coordination barriers - the programming system's boundaries establish how a programming language's user interfaces and libraries can be connected to achieve complex behaviours. For example, a learner may correctly assume that there is inter-module communication involved in creating a new module by writing a program and accessing its data. However, he/she can make invalid assumptions about how to access data and how to try to "pull" values from the new module instead of "pushing" values to the old module. Again, as functional programming languages have no side-effects and variables can never be changed, so pulling values from a function is something a user does all the time and so the coordination overhead is simpler.
4. Understanding barriers - the properties of external behaviour such as compile and run-time errors that hide what a program did or did not do at compile or runtime. For example, when a learner writes a function without a '=', he/she receives the error message "expected: =". The learner needs to learn and understand where the '=' should be placed, and why it is "expected." The visual system in this research prevents users from making syntax errors. The direct-manipulation technique helps users to construct a program

without any prior knowledge of syntax. This research also implements visual error reporting which can help users to gain a better understanding and more ability to locate an error.

5. Use barriers — when learners know about the interface they want to use, but are misled by their difficulties in using them. For example, a learner can make invalid assumptions about how to use a method or what effects they would have, passing syntactically correct but semantically incorrect arguments (e.g., when a function can take two arguments, but it was given four arguments). The use of a strongly typed functional language in this research allows users to construct only semantically correct programs. Also, the visual system in this research shows standard library functions, non-standard library functions and user-defined functions as icons with a number of input slots and type information which can help users to choose a specific function.
6. Information barriers — the properties of an integrated development environment that make it hard to acquire information about a programs' internal behaviour such as a value of a variable or what calls what. For example, a learner might accidentally close a panel window/ toolbar and then can not be able to determine how to redisplay it. This is a problem caused by the user interface design of the IDE. In the end-user programming system developed in this research, the 8 golden rules of user interface design have been followed to resolve

this barrier.

The design of the end-user programming system developed in this research will aim to address, these barriers. It combines visual programming techniques with HCI techniques. The HCI techniques include rules of user interface design, data display design, icon design, and direct manipulation. The end-user programming system in this research aids end-users to overcome the barriers to learning a programming system and particularly to learning a functional one. For details, see Chapter 4.

1.5 Choice of Functional Language

The target domain of the new programming system in this research is a general purpose functional programming in Haskell. Haskell is a strongly typed functional language. The main difference between HASKELL and some other functional programming languages (for example, ML and Scheme) is in *strictness*. In Haskell, a function does not evaluate its arguments unless, and until their values are needed, which is known as *lazy evaluation*. The functional programming language, Haskell, has been tackling some of the interesting problems faced by computer scientists for more than 30 years. Nondeterminism, concurrency, state, time, efficiency, and decidability are all issues that functional languages address (Peyton Jones et al., 1996). Concurrent programming by using coroutines is a natural consequence of lazy evaluation, for example (Hughes, 1989). However, existing functional programs are textual with unfamiliar syntax, and involve unfamiliar concepts such as the use of higher-order

functions which have other functions as their arguments and/or their results. It is all too common for these programs to be surprisingly difficult to develop, especially when they are developed by end-user programmers (Nardi, 1993). The next section will go on to describe the inspiration for a functional programming development system that combines textual and visual elements to enable such languages to be more widely accessible.

1.6 Design Inspiration

It has been found that non-programmers can write quite complex programs in visual programming systems with little training (Halbert, 1984; Cypher, 1993). Different visual languages have developed many systems (such as signal processing, image processing and instrumentation) which perform their task satisfactorily using dataflow programming techniques (Bier et al., 1990; Rasure and Williams, 1991). This is an indication that a visual functional programming system may improve the learnability of functional languages by end-users. Visual representations aid understanding and memory retention, and may provide an incentive to learn how to program without language barriers. In more main-stream programming languages, representation of operations in different notations, such as visual versions of textual languages can ease program understanding (Green, 1990). The visual system discussed in this thesis can be seen as a support for programming using the Haskell textual language. Visual notations can be used effectively to give the programmer an

overview of a program's structure such as UML notations representing aspects of object-oriented languages (Larman, 2004). Similarly, relations, connectivity and type information may be grasped more easily through visual representations than through textual representations.

The use of Human-Computer Interaction (HCI) techniques to design visual programming systems is relevant to this investigation (Pane, 1998; Pane et al., 2002). One of the popular HCI styles is direct manipulation (Shneiderman and Plaisant, 2004) which is a continual depiction of the objects of interest, and involves fast, undo-able, and incremental actions and feedback. The benefits that can be gained by using direct-manipulation in the design of visual programming system are: control-display compatibility; less syntax leading to reduced error rates; help provided with language semantics; the avoidance of syntactic and semantic errors before compilation is attempted; errors are more preventable; faster learning and higher retention; encouragement to explore; the programmer is always kept aware of the result by the representation providing continual feedback; and the object of interest being immediately visible.

This is not the first time the concept of a visual functional language programming system has been proposed. A previous attempt, called Visual Haskell (Reekie, 1994), a visual programming system for Haskell can be found in the literature. The Visual Haskell by Reekie was more of a visualization tool than a visual programming system. The end-user programming system in

this thesis makes use of ideas from visual programming in general, and Visual Haskell by Reekie in particular. The relevant aspects of the Visual Haskell by Reekie are described in Chapter 2. This thesis contributes to original research by creating an alternative programming system which has many advantages (see Section 1.10) beyond Visual Haskell by Reekie. Not many attempts have been made since that one, possibly because pure functional languages such as Haskell are not widely used and the challenge of creating such a system whilst being true to the functional paradigm. A recent, nominally visual, programming system to support Haskell is also called Visual Haskell (Angelov and Marlow, 2005), which is a Haskell development system to support textual programs, rather than visual ones.

This research also uses the idea of the Model-View-Controller (MVC) design pattern (Fowler, 2002) and Functional Reactive Programming (FRP) (Elliott and Hudak, 1997; Hudak, 2000; Peterson et al., 1999; Reid et al., 1999; Courtney and Elliott, 2001). MVC has been found to be a very useful and widely used design pattern for implementing user interfaces in object-oriented programming languages. The aim of functional reactive programming (FRP) is to provide a powerful way to describe reactive systems. So, it was worth investigating how easily MVC could be implemented in a functional programming language with the use of FRP (see Chapter 3), as the new reactive programming system in this research consists of two views (textual and visual) and many controllers.

The aim of this research was to develop a programming system for Haskell and the focus will be on solving the technical problems. Here, focusing on solving the technical problems means analyzing technological aspects such as Haskell’s syntax tree, Haskell’s type system, making use of FRP and laziness, making use of MVC, the use of HCI and finding a suitable solution to designing and implementing a supportive programming system for Haskell. Combining all these technologies to implement a new programming system in a pure functional paradigm is a challenge. A usability test was conducted with a range of end-users to evaluate the resulting system (see Chapter 6).

1.7 Research Aims

RESEARCH QUESTIONS:

The main research question “Is it possible to build a usable programming system for the end-user development of functional programming?” can be split into the following three relevant questions:

1. How suitable is textual programming for the end-user development of functional programming?
2. How suitable is visual programming for the end-user development of functional programming?
3. How can textual and visual programming be effectively combined in a

functional programming system?

RESEARCH OBJECTIVES:

These research questions will be answered by achieving the following objectives.

1. The development of a programming system based on textual functional programming, and a subsequent evaluation of the suitability of textual functional programming for end-user development.
2. The development of a programming system based on visual functional programming, and a subsequent evaluation of the suitability of this visual functional programming for end-user development.
3. The production of a single system that combines textual and visual programming using a Model-View-Controller (MVC) design pattern. This will allow end-users to program using both notations at the same time, and for them to compare their effectiveness.

1.8 Research Methodology

The research in this thesis is actually an implementation research rather than an action research. It refines the implementation after evaluation, at each step. Action research involves finding an answer to an instant issue, or it is a reflective continuous process, where finding an answer is led by people working with others in a group, or as part of a “community of practice”, to improve their ways of addressing and solving issues (McNiff, 1988). Implementation research is of great importance when satisfying challenges, and it gives a basis

for the context-specific, evidence-informed decision-making necessary to make what is possible in theory a reality in practice. Implementation research is a problem-focused, action-oriented research (Denicolo and Becker, 2012).

Data will be collected in user studies and by running tests. In the user studies both quantitative and qualitative data will be collected. The quantitative data will be a measure of the learning time and the accuracy of the program development. The qualitative data will be how well the program meets its specification. By running tests only quantitative data will be collected to understand behavioural characteristics of the system.

The following steps were taken as a research methodology:

1. The first step taken was to become familiar with the literature about functional programming, end-users, visual programming and Human-Computer Interaction (HCI). It was important to become conversant with the way that functional programs are developed using traditional textual notation. It included an examination of the typical functional programming styles in programming some `Prelude` functions. There followed an investigation into how GUI programming has been achieved in a functional programming language using standard libraries, and in particular how to implement the Model-View-Controller (MVC) design pattern in a functional programming language. This provided the springboard for research objectives 1, 2 and 3.

2. The next step was to create a textual programming system for functional programming, taking `Prelude` functions as an example domain. The textual representation of a program may be seen as one “view” of the underlying “model” that represents it. This model is, in fact, a syntax tree. This achieved research objective 1. A visual programming system for functional programming was then created, again taking `Prelude` functions as an example domain. The visual representation of the program produces another “view” of the underlying “model” that represents it. This achieved research objective 2.

3. Then, there was a further literature review to determine the criteria by which textual and visual programming systems would be fairly compared. A study was conducted of a small set of end-users recruited by invitation, but with no previous involvement with this research. These end-users performed the same programming tasks with both the textual and visual programming systems in order to provide a qualitative and quantitative comparison between them. The findings were recorded anonymously, consistent with the Handbook of Research Ethics. This achieved research objective 3.

It was decided to call the newly developed programming system “HASKEU” which stands for — Haskell for End-Users.

1.9 Design, Implementation and Testing Approach

To achieve the goal of this research and hence to design and implement HASKEU, a choice of a suitable software development life cycle (SDLC) model was important (Ruparelia, 2010). There are different SDLC models for various types of projects. The oldest SDLC model is the Waterfall model (Sommerville, 1995). Its main disadvantage is that it is not possible to make changes to the project as requirements change at a later stage. The Agile model is useful for projects with a large team where customer collaboration is also important (Beck et al., 2001). In Agile, requirements and solutions develop gradually through collaboration between the self-organizing and cross-functional teams. Agile supports flexible planning, progressive development, early delivery, ongoing advancement, and it supports fast and easy responses to change. SCRUM is one of the many iterative and incremental Agile methods. In SCRUM, the basic unit of development is called a sprint. Each sprint begins with a planning meeting. In the planning meeting, the tasks for the sprint are established and an approximate outcome is set. The Rational Unified Process (RUP) is another classification of waterfall model and it has a strict cut-off between phases (Sommerville, 1995). The Spiral model uses iterative methods and it allows for there to be as many changes as are needed (Boehm, 1988). There is no universal SDLC model that fits all projects. The development of the HASKEU system was a Spiral development. The main benefits gained of using spiral

development in HASKEU project are outlined below:

1. Because the project was developed in an iterative way, the complexities of the problems were discovered in stages along the way rather than one major problem at the beginning.
2. In each stage, a review was undertaken and this clarified the key things which needed to be done in the next stage.
3. It was possible to add extra functionality at a later date;

The spiral development consists of four phases: Planning, Risk Analysis, Engineering and Evaluation. In this iterative approach, a software project repeatedly passes through the four phases Figure 1.1.

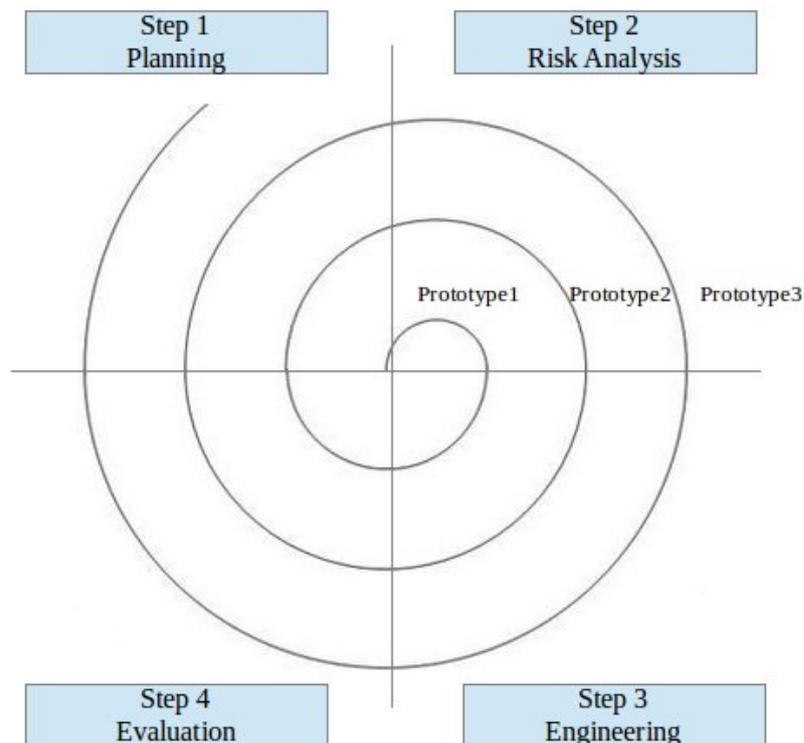


Figure 1.1: The spiral development.

The scheduling and development of different parts of the project was carefully planned and documented at each of the iterative stages. The documentation created at each stage could then be easily referred to and used to create the technical documentation of the final version of HASKEU. Two of the main parts of the HASKEU system are - the visual programming and textual programming. Although they are very interdependent and changes propagate between them, the functions to make these propagations possible were written separately and in a completely declarative way (see Chapter 5). The declarative nature of functional programming is very suitable for iterative development as is demonstrated in Spiral SDLC. The concept of the HASKELL syntax tree became clear by first analysing, designing and implementing the textual system and then the same stages of the visual system were realized. Similarly, to support the type checking and error reporting etc. in the visual system, it was important to know how to store the type information, how to check individual expressions etc.

The Design/Implementation/Testing was divided into the following steps:

- Prototype 01: Develop the textual programming system in HASKEU.
- Prototype 02: Develop the visual programming system in HASKEU.

In the evaluation phase, it was felt that the design was not simple enough for end-users to do programming and it could produce *spaghetti code* (Van Tassel, 1974). The term “Spaghetti code” refers to those codes which are intricate and poorly organized. It was first appeared when the

jump was made from assembler to structured programming. Now, for the same reason, many developers create visual spaghetti code (Whitley and Blackwell, 1997). Examples of visual spaghetti code are given in the Appendix B. The separation into pieces and automatic layout of a screen program have been implemented in some visual programming systems (such as IBM's VisualAge) for decreasing the complexity of a visual spaghetti code (Gibbons, 2002). In the first visual system of HASKEU, a module and all its functions were shown in one graphical window. Users could draw visual programs in which functions, nodes and lines could overlap each other. Changing any part of the module was likely to affect the functionality of other parts and caused unintended changes.

- Prototype 03: After a further literature review, stage three was an implementation of the refined visual programming system with support for “one function definition per page” (see Section 4.1.5) and “automatic layout” (see Section 4.1.3).
- Prototype 04: Production of the combined textual and visual programming system with the use of MVC and FRP. The “MVC as a FRP” theory was introduced (for more details, see Chapter 3).
- Prototype 05: Changes were made in the textual implementation. Previously the text editor did not need to be updated as it was the only system. When the textual and visual were combined, the text in the text editor was updated by the model value and the cursor position was lost. The syntax tree did not contain the cursor position, so then it was

decided to explicitly save the cursor position in the model.

- Prototype 06: Changes were made in the visual implementation to synchronize it with the textual system, as the visual system is syntax error free and the textual system can have syntax errors (for more details, see Section 4.4.1). Version 1 of HASKEU was released to support both visual and textual programming. This first version had no type checking facility.
- Prototype 07: Version 2 of HASKEU was designed to support type checking.
- Prototype 08: Reimplementation of the visual system so that types and type errors could be shown visually (see Section 4.2.1).
- Prototype 09: The system was exhaustively tested and the final version of HASKEU was released.

Although the Software Engineering Life Cycle (SDLC) gives a general overview of ordering different phases of software engineering, the implementation and documentation of the phases (from analysis to maintenance) depend on the underlying programming paradigm of the programming language on which the system will be developed. HASKEU was developed in a pure functional programming language (Haskell) and did not use a formal analysis and design method. Appendix A gives more explanation about appropriateness of using analysis and design for functional programs.

The implementation of the above mentioned prototypes of HASKEU demonstrates the research contribution outlined in the following section.

1.10 Contribution

As well as the design of a programming system, the implementation demonstrates the purely functional Model-View-Controller (MVC) design pattern as a reactive program using the elegant Functional Reactive Programming (FRP) framework. This implementation is the foundation of the programming system, HASKEU. This thesis has produced the following new features for end-users functional programming in Haskell:

1. Support for both visual and textual functional programming allowing for a smooth transition from one to the other as programming expertise increases;
2. The propagation of changes between the visual and textual interfaces, so that they are always consistent;
3. Extensive use of Human-Computer Interaction (HCI) techniques have been included.
4. No syntax errors can be created in the visual system;
5. The provision of an automatic layout to display the dataflow graph in the visual system;

6. The use of a block-based architecture to represent the scope of expressions in the visual system;
7. An on-time visual display of the textual notation of the type of a function application and its individual arguments;
8. The provision of visual error reporting;
9. Unnecessary and unused argument slots are clearly shown;
10. Guidelines for the display of data in visual systems are satisfied;

The remaining six chapters of this thesis cover the following areas: Chapter 2 shows a literature review and related research which influenced this study. It also includes a comparison of the programming of some typical instructions using a functional programming language, a visual programming language, and a visual functional programming language. Chapter 3 shows the implementation of the Model-View-Controller (MVC) design pattern as a reactive program. Chapter 4 describes the design of HASKEU - data display, icon design, use of HCI, dataflow of functional program, organization of the display, and error reporting. Chapter 5 shows the implementation of the system in the following order: the model, the controllers and the views. This chapter also shows the results of the functional test of the infinite redo/undo facility of HASKEU. Chapter 6 shows the usability test process and the results of this usability test. Chapter 7 presents conclusions containing the achievements, the limitations and suggestions for future developments.

Chapter 2

Background and Literature

Review

2.1 End-user programmers

As mentioned in the introduction, according to Nardi, *end-user programmers* are not “casual”, “novice”, or “naive” computer users; rather, they are people such as chemists, librarians, teachers, architects, and accountants, who want to make serious use of computers, but who are not interested in becoming professional programmers (Nardi, 1993). Such end-users may program computers daily, at least for a period of intensive work on a project. End-user systems should be targeted at them; others with infrequent computational needs can enlist contract or in-house programmers to write the few programs they need.

Although end-users represent a continuum of people with different techni-

cal skill levels, Nardi and Miller classified them into three discrete user groups — non-programmers, local developers and programmers (Nardi and Miller, 1990). Non-programmers have little or no programming education and lack an intrinsic interest in computers. Local developers have a good knowledge of specific programs. Programmers have a good education in the general use of computers and therefore have a broader technical knowledge than the other groups. If an appropriate design principle can be created for non-programmers, it is obvious that local developers and programmers can use the system as well. Also, the total number of experienced programmers and local developers is substantially smaller than the number of inexperienced non-programmers (Ko et al., 2011).

End-user programming (EUP) is defined as “programming to achieve the result of a program, rather than the program itself” (Ko et al., 2011). The developer’s goal in EUP is to use the program for a specific, personal purpose, whereas the goal in professional programming is to create a commercial program for other people to use. An end-user development can be an extension of an existing application, or it can be a new application, which runs separately from existing applications. Some popular EUP systems are spreadsheets, computer aided design systems and statistical packages. A key feature of these systems is that a useful subset of their functionality can be learned after no more than a few hours of instruction.

EUP does not mean using only simple languages. Many scientists use general-purpose languages like Java to analyze scientific research, with no intention of sharing the program for commercial use or polishing it for future use (Segal, 2007). An end-user programmer may use any of the wide range of languages, from task-specific languages to high-level general-purpose languages. SPSS (SPSS Inc., 2007) and Mathematica (Wolfram, 2003) are examples of task-specific language. The choice of language is important only to achieve the end-user's personal goal. However, task-specific languages lack the power of general-purpose programming languages. Also, it is expensive to build many different task-specific languages. Users would be forced to switch between many different languages and it is difficult to know how specific a task-specific system should be (Nardi, 1993).

Many end-users have serious difficulties learning general-purpose programming languages. One of the major barriers in learning such languages is becoming familiar with low-level programming primitives and assembling them into a functioning program (Lewis and Olson, 1987). Many end-user get discouraged at the amount of work needed to master a conventional programming language (Nardi, 1993). Novice programmers often have great difficulty understanding control constructs in programming languages (Lewis and Olson, 1987; Spohrer et al., 1985). End-users also face many of the same software engineering challenges as professional developers do. They need to choose APIs, libraries, and functions to use (Ko and Myers, 2004). Their programs contain

errors, and they also face the critical consequences of failure (Panko, 1998). Error rates increase when they work on large programs (Panko, 2000).

End-user capabilities can be summarised as follows:

- they may not be expert programmers (Nardi and Miller, 1990);
- they may be unskilled, and make false steps (Lewis and Olson, 1987);
- they may be uncertain, and make false starts (Ko and Myers, 2004);
- they find large programs daunting (Panko, 2000).

The next section will go on to look at properties of functional languages.

2.2 Functional Programming Languages

2.2.1 Motivation

For many years, researchers have argued that computer programs would be easier to develop and maintain if they were written in *functional programming languages* (Backus, 1978; Darlington et al., 1982; Hughes, 1989). Unlike traditional imperative programming languages, functional programming languages do not allow functions to have any *side-effect* — they compute only results. This constrains the use of *sequence* and *state*, which often cause programmers, and especially end-user programmers, to make mistakes.

2.2.2 A Functional Language : Haskell

This section describes a modern functional programming language, Haskell (Peyton Jones, 2002) which is proposed as a suitable language for EUP.

Functions

A function determines a *result*, which depends on one or more *arguments*. It may be defined by one or more *equations*, which may be *recursive*; that is, a function defined in terms of itself. For example, the factorial function may be defined 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`.

Types

Types describe values. Among the basic types 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. 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
    | Branch (Tree a) (Tree a)
```

Here, the identifiers `Leaf` and `Branch` are the *constructors* of the type `Tree`.

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, `(t1, t2, ..., tn)` 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.

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
```

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* (Bird and Wadler, 1988).

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
```

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 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 coding is the instructions are:

```
do
    s <- hGetStr
    putStr s
```

2.2.3 An Example Haskell Program

Below is an example functional program, which shows how Haskell may be used to calculate the voltage in the electrical circuit shown in Figure 2.1.

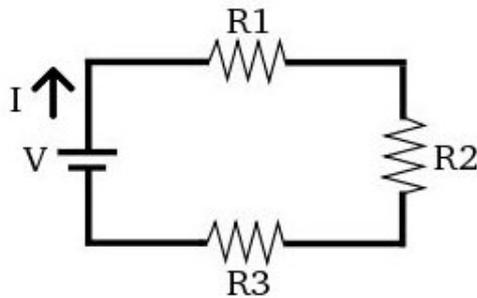


Figure 2.1: An electrical circuit.

Suppose the values of the resistors in this circuit are: $R1 = 8.0 \Omega$, $R2 = 6.0 \Omega$, $R3 = 4.0 \Omega$, and that the current through each resistor is: $I = 0.5 \text{ A}$. By Ohm's law, the total voltage of the circuit can be found by simply adding up the voltages at each resistor, where each is calculated as $V = IR$.

In Haskell, the calculation is implemented as follows:

```
totalVoltage :: Double
totalVoltage = sum (map (* 0.5) [8.0,6.0,4.0])
```

The function `sum` adds up all items in a list passed as an argument. The list that results from applying `map` is `[4.0, 3.0, 2.0]`, and the value that results from applying `sum` is `9.0`.

The next section explains the main difference between Haskell and other functional programming languages.

2.2.4 Other Functional Programming Languages

Other commonly used functional programming languages are Standard ML (Milner, 1984) and Scheme, a dialect of LISP (Steele Jr. and Sussman, 1978). The main difference between these languages and Haskell is in *strictness*. In ML and Scheme, a function call of the form

$$f \ (e_1, e_2, \dots, e_n)$$

causes the argument expressions, $e_1 \dots e_n$, to be evaluated before the body of the function f is executed. In Haskell, however, a function does not evaluate its arguments unless, and until their values are needed, which is also known as *lazy evaluation*. The main benefits of lazy evaluation are: it increases performance by avoiding unnecessary calculations and it is possible to construct infinite lists (Hudak, 1989). Haskell was chosen as a target domain for this research because of its laziness so that end-users do not need to put much effort into thinking about the performance of the program they develop. For the same reason, it was chosen as the development language for HASKEU.

Existing functional languages usually have an unfamiliar textual syntax, and programming in them involves unfamiliar concepts such as the use of

higher-order functions (functions having other functions as their arguments and/or their results). Hence, the development of these programs is “surprisingly difficult”, especially by end-user programmers (Nardi, 1993). In the next sections, various ways to support language learning will be discussed.

2.3 Support for Language Learning: Syntax-Directed Editor

To make textual programming easier, syntax-directed editing is particularly useful (Bai, 2003). A syntax-directed editor provides alternative ways for manipulating programs by creating or modifying programs in such a way that correct syntax is always produced. In most syntax directed editors, the display cursor indicates where to enter the program text, and the program display on the screen is automatically updated. A syntax-directed editor differs from a text editor as a text editor only updates the text, whereas a syntax directed editors updates the syntactic structure of the program as well as the text. When a user enters text, the syntax-directed editor immediately checks for errors and displays appropriate messages to the user. This prevents the user from writing syntactically incorrect programs. Program text is entered by making a template or a skeleton of a syntactic construct, and filling in the detail later. Novice users can benefit by using syntax-directed editing because these editors are considered to be a good medium to learn a new language and its construct. Advanced users benefit from using a syntax-directed editor for program main-

tenance (Horwitz and Teitelbaum, 1986; Reiss, 1985; Hubbell et al., 2006). A syntax-directed editor can allow more rapid program construction by automatically managing many of the syntactic structures, such as keywords and syntactic sugar. In spite of these advantages, the use of a syntax-directed editor for trivial tasks can be expensive and it can add more difficulty for users learning more complex tools (Hubbell et al., 2006). It was found that the syntax directed editor is too complicated and irritating for inputting programs or for performing simple editing tasks as it involves a sequence of operations to be performed (e.g., select a menu, select a tree node, restriction in editing for syntactically incorrect input, press enter when finished). Also, in a complex tool, selecting and learning the right menu can be a complex task. Research has found that syntax-directed editors with some ability for “free typing” can be more effective (Waters, 1984). 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).

As with other syntax-directed editors, the HOPS (Higher Object Programming System) is a graphically interactive term graph programming system which is designed for transformational program development (West and Kahl, 2009). Term graphs are closely related to the terms that are encoded in the textual representation of conventional programs, but their visual appearance is different. In HOPS, with the use of an extended syntax directed editor, only syntactically correct and well-typed programs can be constructed. It makes the

program's structure and program's typing structure explicit and interactively accessible in the shape of term graphs. A visualisation of program execution can be achieved by displaying all intermediate graphs of an automatic transformation and it has been found useful for debugging complex purely functional programs. Below are some example programs in HOPS (see Figure 2.2). The black arrows indicates successor edges and their sequence is indicated by the left-to-right order of their attachment to their source. The thick, usually curved arrows indicate binding edges. The Figure 2.2a corresponds to the terms $5 * 5 + 5 * 5 * 2$ in arithmetic. The Figure 2.2b corresponds to the terms $\lambda x f = f x$ in lambda-calculus. In Figure 2.2b, it should be noticed that in the linear notation the two different bound variables need different names f and x since they occur in the same scope, but in the term graph of HOPS, different nodes are labelled x because they represent different binding variables.

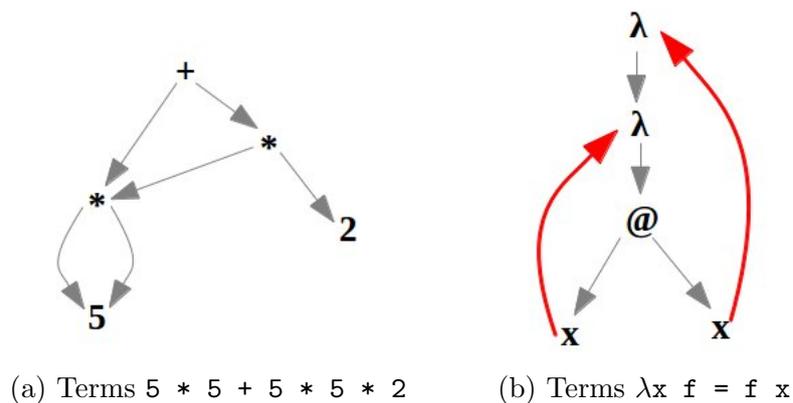


Figure 2.2: Example programs in HOPS.

HOPS is remarkable because of its support of the program transforma-

tion feature and proof assistant facility for a Haskell-like functional program by showing the program as a term graph instead of a textual representation (Vangheluwe and de Lara, 2003). The interactive program transformation and editing environment in HOPS ensures that it will always construct syntactically correct and well-typed programs (West and Kahl, 2009). The HASKEU system developed in this thesis has a different purpose from that of HOPS:

- HOPS is a text-based term-graph programming system and is not a visual programming system by any means, so it was not intended to be used to support learners. HASKEU is a hybrid textual and visual programming system and is intended to help end-users to learn functional programming.
- HOPS has an extended textually based syntax-directed editor that does not allow any syntax errors. This thesis analyses the pros and cons of a syntax directed-editor at the beginning of this section and has concluded that a program editor with the support of both a syntax directed-editor and free-typing can be helpful for both learners and expert programmers. The HASKEU system use free-typing in its textual editor and syntax-directed editing in its visual editor.
- HOPS does not allow any type errors. This research argues that highlighting type errors in a visual way can help end-users to learn functional programs. To support this, the HASKEU system allows end-users to make type errors in the visual system and gives feedback about the type

errors in such a way that end-users can understand and correct their errors efficiently (See Section 4.2.1).

Some features in HOPS such as debugging support can be useful to carrying out functional programming tasks, and would be a useful feature if implemented in a future version of HASKEU.

This thesis explores the use of a visual representation as well as textual notation. The visual system incorporates the ideas gained from syntax-directed editing to allow the user to construct only a syntactically correct program. At the same time, the textual system has the flexibility of “free typing”. The next section examines the characteristics of visual programming languages.

2.4 Support for Language Learning: Visual Programming

2.4.1 Motivation

Learning to program is often a time-consuming and frustrating endeavor. Moreover, even after the skill is learned, writing and testing programs may be a laborious activity. Conventional programming exploits our ability to think analytically, logically, and verbally, whereas *visual programming* (Zhang, 2007; Cox and Nicholson, 2008; Cox and Gauvin, 2011) — defined as “the use of meaningful graphical representations in the process of programming” — exploits our ability to think nonverbally (Shu, 1988). Visual representations aid

understanding and memory retention, and may provide an incentive to learn to program without language barriers. In generic programming, notations, such as visual versions of textual languages, can ease program understanding (Green, 1990).

2.4.2 Visual Programming: Overview

Computer programming is a very difficult and challenging area and it places a very cognitive load on learners. Still after a few years of learning, novice programmers struggle to be skilled (Mow, 2008). The modern form of textual programming has been around for over 60 years, and it has been taught in universities since the 1960s. Since then, many different techniques and tools have been developed to aid in learning computer programming (Newby and Nguyen, 2010). The visualization of a program provides valuable information to the programmer while programming and thus it has the potential to result in a better and faster understanding of the program's functionality, and can save time while programming is being carried out. As a program consists of huge amount of information, visualisation of the program is not very easy. So, it is very important to consider the human perception of information and cognitive factors required to visualize a program and hence the program understanding will be improved with the aid of the visualizations (Caserta and Zendra, 2011). The principal goal of the visualization of a program to be visualized is to increase code comprehension. (Marques et al., 2012). The purpose

of visual programming is not only to help users to learn a new program but also to help them with the development and maintenance of programs. These processes just mentioned have been found to be time-consuming activities in practice. Especially while maintain a program, a developer does not want to read the entire code of a huge program. Hence, they can benefit from the visualization of the program which can help them to get a rapid understanding of the source code (Haiduc et al., 2010). Now-a-days, end-user programming has become the most popular form of programming. This trend has led to the situation that there are now now more end-user programmers than professional programmers (Hofer et al., 2013). Though the growth of the end-user programming is rapid, not much research has been conducted about how to bring the advantages of visual programming to the awareness of end-users, especially in the functional programming area.

The following section describe the differences between two common terms (“Visual programming” and “program visualizations”) used in computer science research to improve programming support.

Visual Programming vs Program Visualizations

These two terms “Visual programming” and “program visualizations” are commonly used when there is a need to improve programming support in computer science research (Bentrad and Meslati, 2011). As there are significant differences between the two systems, it would be better to give some short definitions of these two to avoid any confusions. Program visualization tools can be easily

marked incorrectly as visual programming (Myers, 1990).

As a broad definition, a system can be called a visual programming system if it allows the user to build a program in a two or more dimensions way. Textual program are one dimensional as the system treats them as a long one dimensional stream. A software which is used to define pictures, such as X-II Window Manager Toolkit (McCormack and Asente, 1988), is not a visual programming language. Similarly, drawing packages, such as Adobe Photoshop (Aaland, 1998), are not visual programming language either as they cannot make programs.

On the other hand program visualization is a completely different approach to visual programming. In visual programming, the program itself can be built using various graphics but in program visualization, the program is actually a textual program and graphics are used to visualize some features of the program such as the run-time values. Program visualization is mainly used in research and training to improve the understanding of the structure, the execution mechanism and the gradual development of the software.

The main goal of both visual programming and program visualisation is to use graphics to show the program in a multi - dimensional visual way as human eye-sight and visual information processing are clearly optimized for multi-dimensional data. A conventional program shown in one-dimensional

textual form does not help to utilize the full power of the brain. It has been known for a long time that some two-dimensional visual forms of programs, such as flowcharts or even the indentations of a textual program are useful aids in program understanding (Smith, 1993). The format used in visual programming is very similar to a user's mental representation of problems (Petre and Blackwell, 1999). Users, especially end-users find a visual style of programming easier to understand. Also, visual programming can make programming tasks easier even for expert programmers by showing them in graphics, better descriptions of the desired actions which are to take place. The use of direct manipulation, where items in a software interface can be pointed and manipulated with the use of a mouse, can contribute to showing such better descriptions (Ohshima et al., 2013).

It is generally believed that learnability is a most fundamental attribute of usability (Grossman et al., 2009). Although a visual program can be very useful in aiding learnability, there are others issues to consider such as HCI while designing the interface. Recent research (Lazar et al., 2006) shows that a user can have frustrating experiences which can lead to up to 40% wastage of the user's time while learning the interface. This can be because if useful features are missing or they are hard to find or there are unusable features in the interface. In this thesis HCI issues have been carefully considered in order to make the learning of the user interface easier and while depicting visualization of a program. Also, direct-manipulation has been carefully implemented

to help users to perform their desired actions (Mow, 2008).

The following section discusses the classification of visual programming languages.

2.4.3 Visual Programming: Classification

In a visual programming language, a visual representation is used instead of words and numbers displayed in a traditional one-dimensional programming representation. It is of no real importance that the object being operated on or being displayed in the visual language is textual, numeric, pictorial, or even audio. The visual representation must convey the meaning what the program is intended to do (Reiss, 1986; Chang, 1987; Shu, 1988; Burnett and Ambler, 1993). Shu (Shu, 1988) has classified visual programming languages into Diagrammatic Systems, Iconic Systems, and Table / Form-Based Systems.

Diagrammatic Systems (Visual Dataflow Programming)

Using diagrams in connection with programming is a technique almost as old as programming itself (Shu, 1988). Control flow diagrams consider the flow of control but ignore the flow of data. In these diagrams, nodes represent statements or decision points and arcs represent the transfer of control between them (Cox and Nicholson, 2008)— for example, Nassi-Shneiderman diagrams (Nassi and Shneiderman, 1973). Dataflow diagrams consider the flow of data but ignore the flow of control (Cox and Nicholson, 2008). In these diagrams,

nodes represent either primitives, predicates or procedures, and arcs represent the movement of data between them — for example, programs for the Manchester dataflow machine (Gurd et al., 1985).

A dataflow model does not have either global updatable memory or a single program counter. It only deals with values. A function can be enabled when all its required input values have been computed. Then, the function is applied to the computed input values and results produced, which are sent to other functions that need these values. A dataflow model does not introduce sequencing constraints other than ones imposed by data dependencies. Dataflow programming languages share some features with functional languages (Uustalu and Vene, 2006). A function call in a functional language is similar to a node call in a dataflow language. The immutable way of passing data between functions in functional languages is identical to the way of flowing data between nodes in dataflow programming languages.

The reasons for describing a dataflow visually are the following:

- (a) 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.

- (b) Smaller dataflow programs can be easily combined into larger programs (see Figure 2.3).
- (c) Graphs presents a natural view of the execution of a program.
- (d) Using graphs, a formal meaning can be attributed to components of a program.

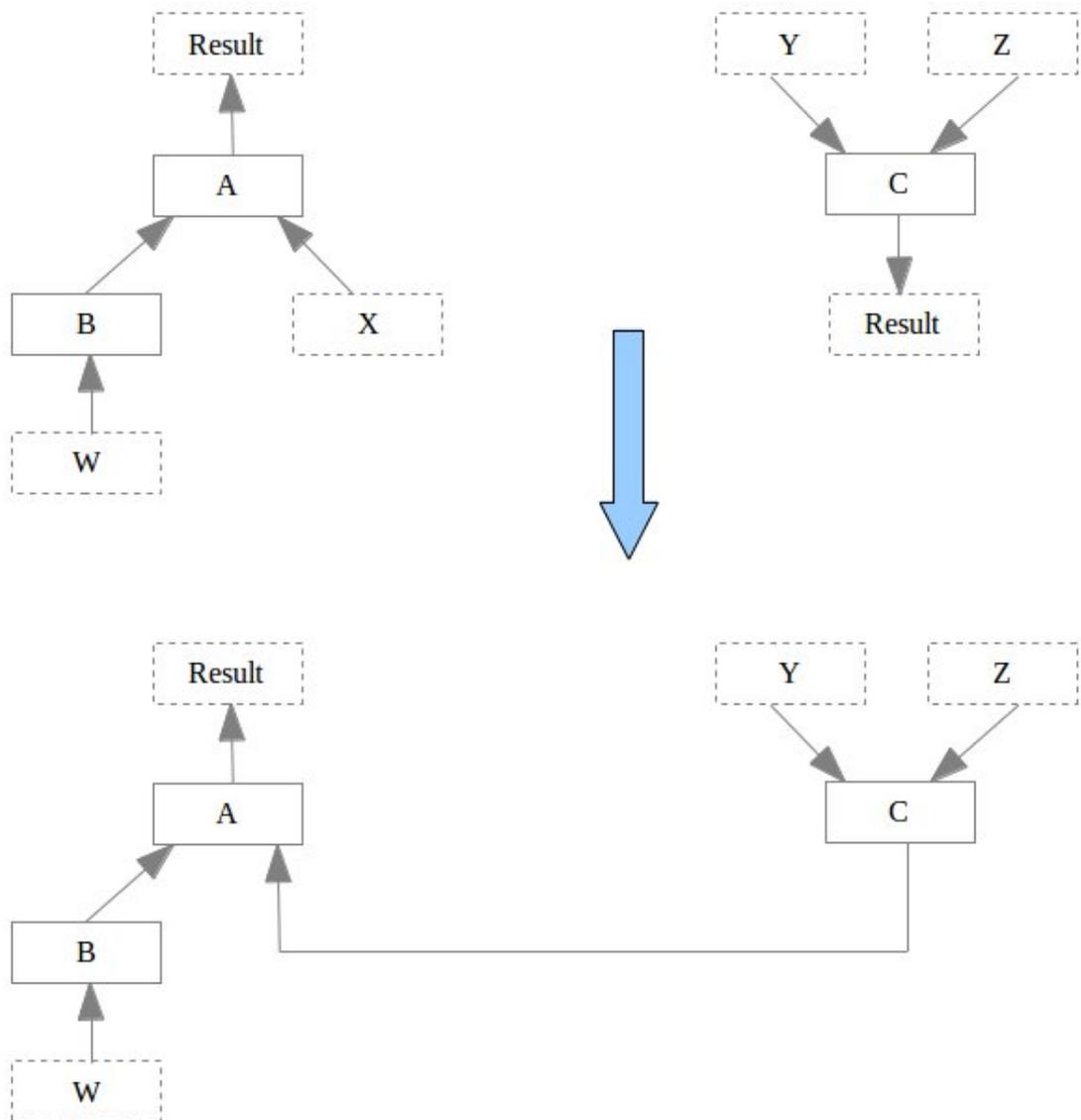


Figure 2.3: Splicing of two graphs.

The visual programming system in this research used the concept of visual

dataflow programming and then functional programming concepts were added (e.g. to show higher-order functions in a dataflow graph, larger programs to be split into smaller graphs). For more details see Chapter 4.

Iconic Systems

Incorporating graphics or pictures into the programming process adds an interesting and useful dimension (Shu, 1988). A significant number of iconic languages have been reported in the literature. In general, they have basically the same goal: to use icons as programming language constructs. In Tinkertoy, programs are built out of icons and interconnections that can be snapped together (Gittins, 1986). The icons have input and output sites through which they can be connected to form structures. HI-VISUAL was originally reported as a language supporting visual interaction in programming (Reiss, 1986). It is now being extended to be an iconic programming language in image processing. Figure 2.4 shows an example of a program in extended HI-VISUAL. Another example of an iconic system is Prograph (Cox and Mulligan, 1985) which is a functional, dataflow oriented language expressed graphically (more details of Prograph are given in Section 2.4.4). Iconic systems have some disadvantages such as icons can be inherently ambiguous, where some icons can be interpreted within a certain context (Lodding, 1983). As there are no universally accepted icons, evolving icons may take much time (Korfhage, 1984). Hence, visual iconic languages can be designed based on the concept of generalized icons. In such concept, object icons consist of a logical part and a physical part. Each icon has an image, which is the physical part of the

icon. Each icon also has a name and some additional attributes, which are the logical part of the icon. The HASKEU visual system was built upon the concept of generalized icons (see Section 4.1.2).

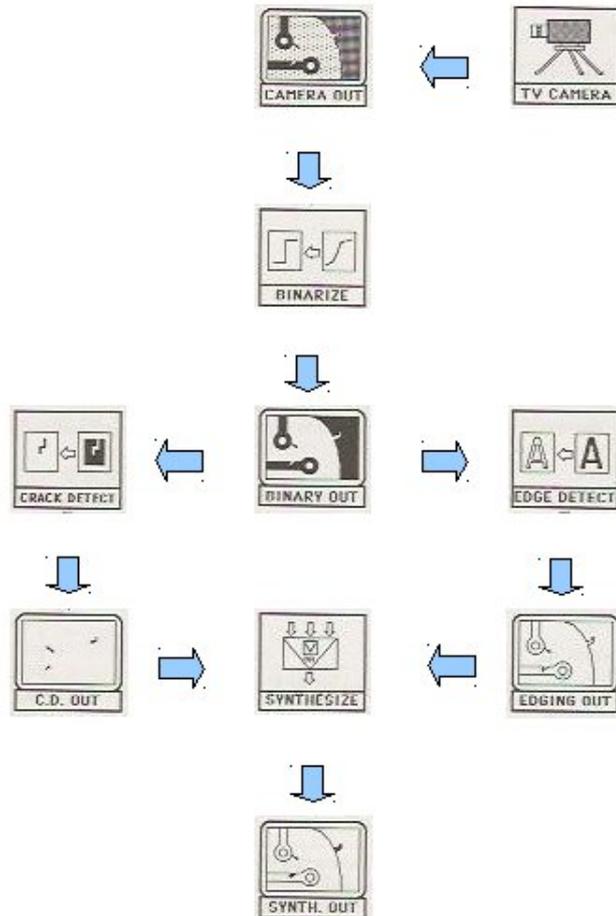


Figure 2.4: Example of a program in extended HI-VISUAL.

Table / Form-Based Systems

In 1979, the CODASYL End-User Facilities Committee (EUFC) stated that “The forms approach was considered the most natural interface between an end-user and data because a large number of end-users employ forms (e.g., purchase order forms, expense report forms, etc.) or versions of forms (e.g. reports, memos, etc.) in their daily work activities as well as their personal

life (e.g tax forms, employment application forms, etc.)” (Lammers, 1986). Figure 2.5 shows a typical QBE system (a database query language) of performing a simple query operation. The table/ form - oriented approach includes electronic spreadsheet systems as a subclass. Although table/ form based approach can be found very appealing and simple by end-users, research on experienced spreadsheet users has found that 44 percent of them tend to create user-generated faults in their spreadsheets (Brown and Gould, 1987). The table/ form - based approach is actually task-specific and it has high maintenance costs for end-users, so is not relevant to this research.

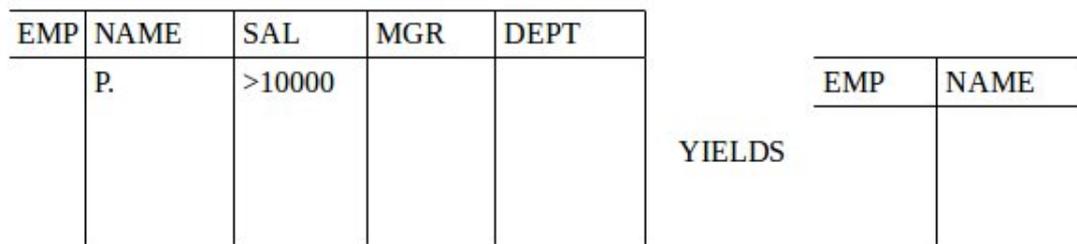


Figure 2.5: Find the names of employees who work in the toy department and earn more than 10000.

The following sections describe examples of some visual languages, an assessment of their value and lessons incorporated into HASKEU.

2.4.4 An Early Visual Language : Prograph

In the imperative programming world during the Prograph development period of the 1980s, when a programmer was given a problem, he/she often determined a sort of flow chart for the program’s functionality. Then, the programmer translated the visual concept into textual code. This translation was difficult because of “the translation of a multi-dimensional process to a

one-dimensional textual form” (Cox et al., 2012; Cox and Mulligan, 1985). The visual programming language, Prograph, was invented as a solution to this problem. Prograph showed visually the flow of data movement and it was significantly different from control-flow languages. When the object-oriented concept became more popular and later became the accepted programming method, Prograph’s modularity and object abilities were announced. Prograph was successful and was used for various applications, mainly on the Macintosh.

Icons

There were 29 different constructs in Prograph, where 4 were main constructs, 9 were basic constructs, 6 were external constructs, and 10 were control constructs. The 4 main constructs were section, universal, class, and persistent. A section was made up of one or more universal, class, and persistent constructs. Universals constructs were functions, procedures, and subprograms. Classes are consisted of methods and fields as in object-oriented programming. Persistents were same as global variables in many programming languages. Figure 2.6 shows some of the constructs used to build programs in Prograph.

Dataflow

The fundamental construct of Prograph was the definition frame, the body of which consisted of a network of method boxes and nodes connected by lines. A method box performed the function indicated by the name or symbol inside it. The flow of information was represented by lines. Each method box had nodes

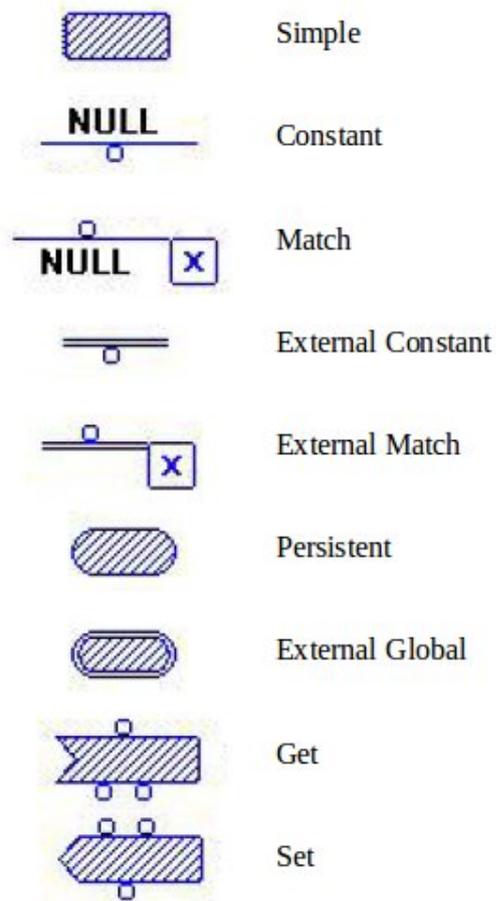


Figure 2.6: Some Prograph constructs.

for inputs and outputs. The methods showed the input to output mapping from one or more inputs to one or more outputs.

2.4.5 An Example Prograph Program

The Figure 2.7 shows a definition of the reverse function in Prograph which reverses the order of a list of elements. The IF-THEN box has a logical and a transformational compartment. The banner portion labelled by IF is the logical compartment. The result of the operation in the oval-shaped box is Boolean. The banner portion labelled by THEN or ELSE is the transformational compartment and it specifies what operations are to be executed when appropriate conditions are met. The THEN compartment in the REVERSE definition uses two system operations which deal specifically with lists. FIRSTREST can take a single input which is a list. It has two outputs - the first element of the list is the left output and the remainder of the list is the right output. APPEND can take two inputs. The data from the right input is added as the last element to the list from the left input and passed to the output.

It was time consuming to learn how to use Prograph. Once the visual nature of the language had been learnt, the programmer had to face the challenge of learning a large number of constructs. Some constructs had their own Prograph version that required a lot of attention to detail. For example, the

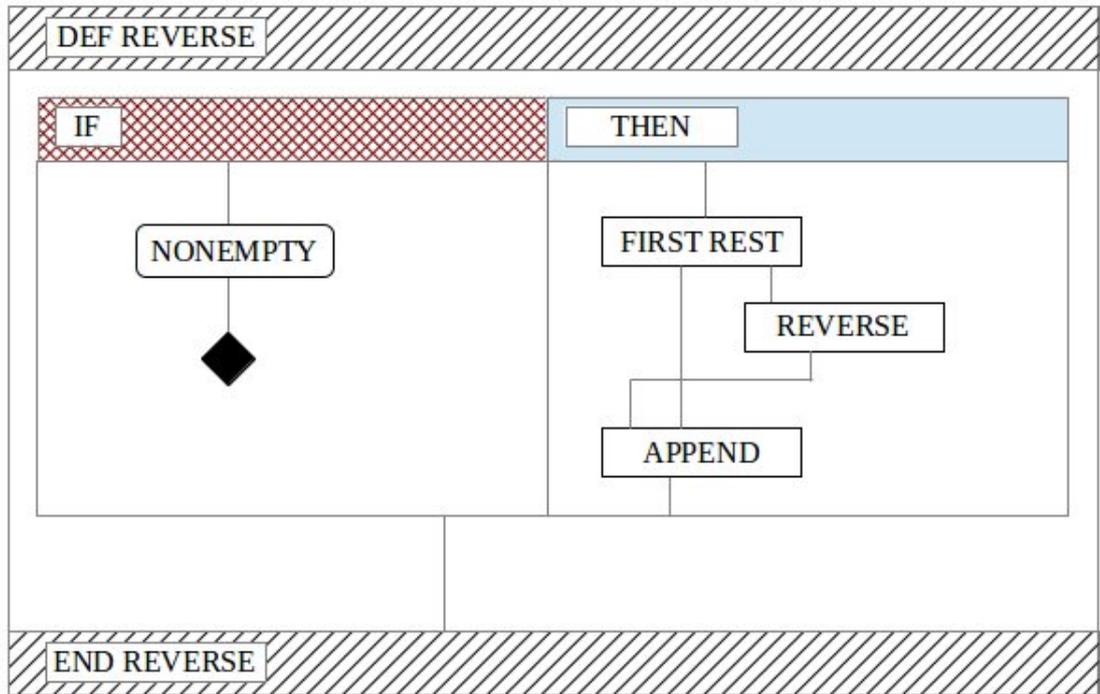


Figure 2.7: Prograph - Reversing order of a list.

conditional construct had 16 different implementations. Unfortunately Prograph, in common with some other visual programming languages, had the big disadvantage that it produced spaghetti code. As mentioned, Prograph had some drawbacks but it's use of diagrams and icons was effective. HASKEU has used these visual ideas from Prograph but taken them in an entirely different direction:

- HASKEU supports functional programming whereas Prograph was aimed at object-oriented programming. These two programming styles are significantly different in their approach to mutable states, program sequences and higher-order functions.
- Prograph used many different constructs. For example, even a condi-

tional construct had 16 different implementations. In HASKEU, the conditional construct is generalized as a function application (See Section 4.1.2).

- HASKEU aimed to avoid spaghetti code in its dataflow-graph. The use of symbols and automatic layout have avoided the appearance of any line-crossings (see Section 4.1.3).
- HASKEU uses HCI concepts in its icon design; hence HASKEU icons convey more meaning than Prograph icons do. For example, a non-empty list in HASKEU is displayed as in Figure 2.8a and in Prograph as in Figure 2.8b.

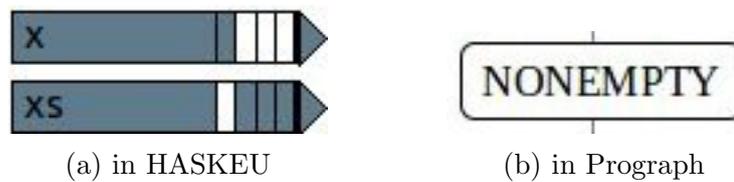


Figure 2.8: A non-empty list icon.

- Functions in HASKEU show their argument slots in a very clear way and also the type information of each argument. This is not displayed in Prograph.
- HASKEU shows visual error reporting which is a very useful tool to enable an end-user to understand errors (see Section 4.2.1). Again, this is another feature which is not available in Prograph.

2.4.6 A Useful Visual Language : LabVIEW

This section describes a useful visual programming language, the Laboratory Virtual Instrumentation Engineering Workbench (LabVIEW) (Johnson, 1997). LabVIEW programming involves constructing *graphs* consisting of *nodes* and *arcs* that can be compiled into executable code. In these graphs, nodes represent iconic *virtual instruments (VIs)*, and arcs represent *dataflow* between them.

Icons

In LabVIEW, icons are selected from the control or function palette (see Figures B.1 and B.2), and are placed on either the *front panel* screen, where controls and indicators show the input and output parameters, or on the *block diagram* screen, where the detailed graphical representations show the wire connections needed to form a VI. Among the advantages of this scheme are quick identification of dataflow, a logical layout and easy editing/troubleshooting by the developer or others. Among the disadvantages of this scheme are the effort required to learn the meanings of icons or images, and to manage the (often large) program graph.

Dataflow

LabVIEW uses dataflow diagrams to connect virtual instruments with “wires” using the tool palette. See Figure 2.11.

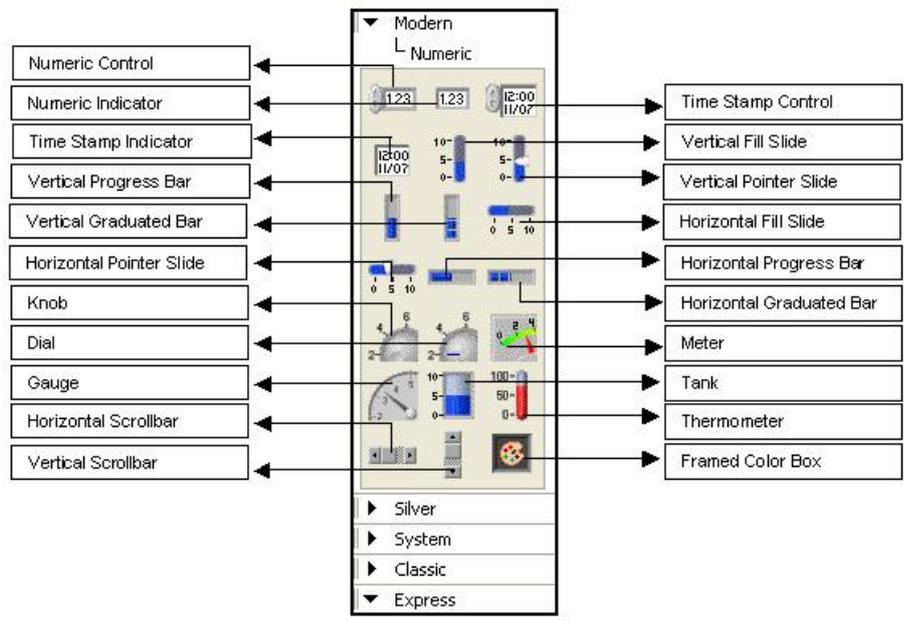


Figure 2.9: The LabVIEW control palette.

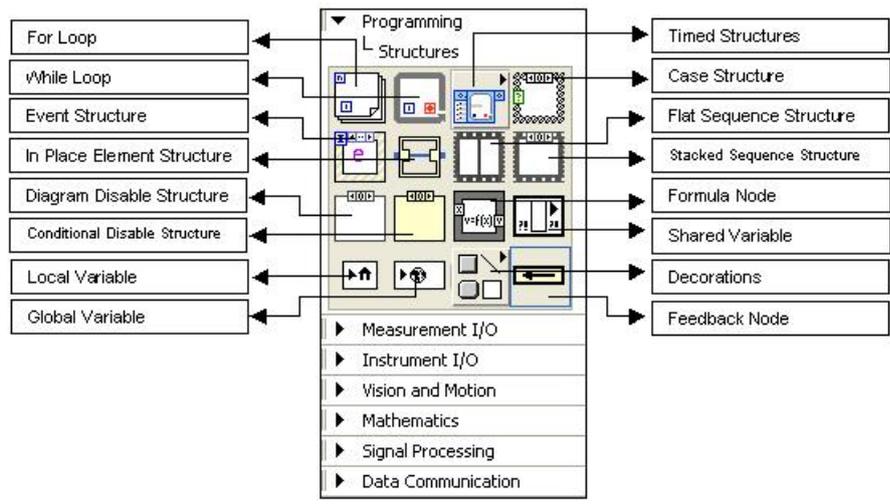


Figure 2.10: The LabVIEW function palette.

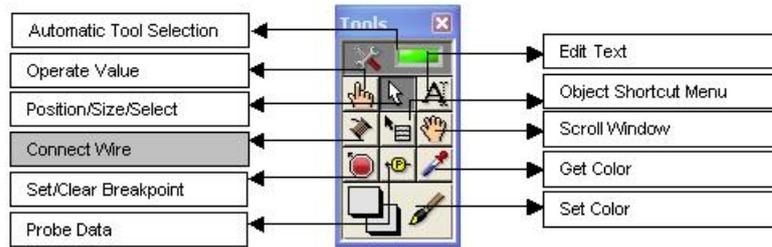


Figure 2.11: The LabVIEW tool palette.

A virtual instrument (VI) executes when it receives all of the required inputs, producing output data that is passed to the next VI in the dataflow path. The movement of data through VIs is all that determines their execution order.

2.4.7 An Example LabVIEW Program

Using LabVIEW, the series circuit of Figure 2.1 can be implemented as follows. In the Front Panel, five numeric controls are added by selecting “Numeric Control” from the “Control Palette” (see Figure 2.12 below). Four of these controls are “I”, “R1”, “R2” and “R3” (for the current and resistor values); the fifth is an indicator (for the total voltage value).

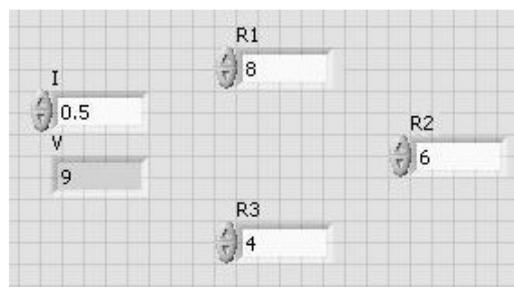


Figure 2.12: Front panel for the series circuit example in Figure 2.1.

In the block diagram screen, three “multiply” and two “add” functions from the “mathematics” option of the “Function Palette” are selected and dragged on to the block diagram as shown in Figure 2.13. This figure also shows all the wirings of the program. Putting all the input values in the the front panel window, and then hitting the run button in the tool bar causes the result (i.e. the total voltage) to be displayed as in Figure 2.12.

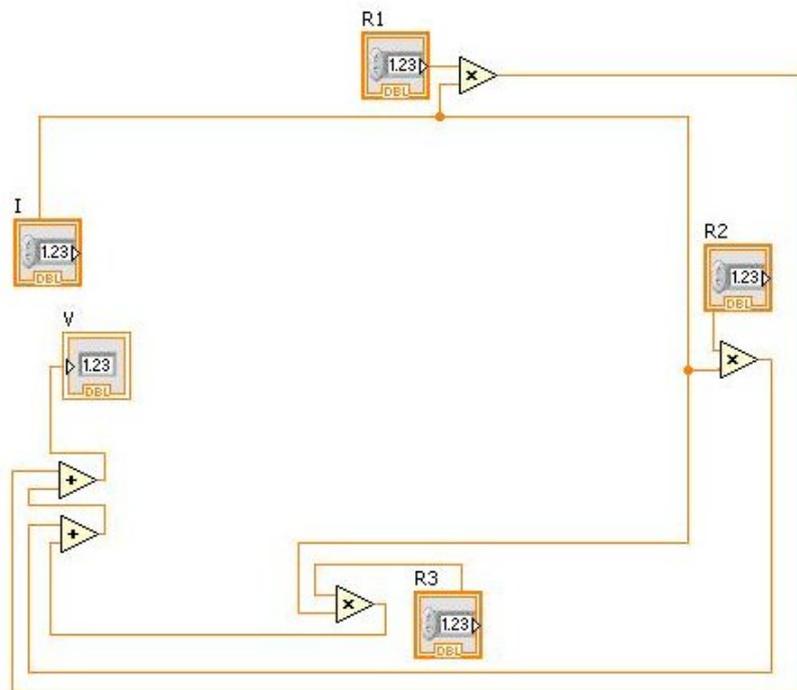


Figure 2.13: Block diagram for the series circuit example in Figure 2.1.

Comparing this example with one in Haskell (see Section 2.2.3), it clearly demonstrates that visual programming can improve program understanding by providing a visualisation of the program and that program development can be made easier by providing the support of different tools and palletes. LabVIEW is aimed at the particular domains of measurement and control sys-

tems and for these areas it is ideal. The HASKEU system took the dataflow iconic programming idea from LabView. Then the key concept of functional programming was added to it. In LabView, a graph can be saved and can be used it on the block diagram of another VI to make a modular program. In the HASKEU visual system, functions and local functions are displayed in separate windows and it also uses a block-based architecture to clearly represent the scope of expressions (see Section 4.1.3). The idea of using a different palette in LabVIEW, was also used in the HASKEU user interface design.

2.4.8 Other Visual Programming Languages

A large number of iconic languages have been described in the literature. In VennLISP (Tanimoto and Glinert, 1990), visual objects are used to direct computations, and the results of the computations are also visual objects. VennLISP is an example of having *executable graphics* based on Lisp. In Tinkertoy (Gittins, 1986), programs are built out of icons and interconnections that can be snapped together. The icons have input and output sites through which they can be connected to form structures. In extended HI-VISUAL (Reiss, 1987), there are seven types of icons — 1) Data icon, 2) Data Class icon, 3) Primitive icon, 4) Panel icon, 5) Program icon, 6) Control icon, and 7) Command icon. It is an iconic programming language in image processing. In Show and Tell (Kimura et al., 1986), a *boxgraph* consists of one or more boxes connected by a set of arrows. A box may contain a data object, a pred-

icate or an operation, and can be nested, and arrows direct the flow of data from one box to another. Visual Programming Language (VPL) (Microsoft Corporation, n.d.) is a visual dataflow-based programming logic system. It allows robotics programs to be created and debugged by dragging and dropping service blocks and a collection of connected blocks can be reused as a single block elsewhere. In VPL, the program is represented as sequences of blocks with connected inputs and outputs, which looks more like a logic diagram rather than a program. Also, to do some significant work, knowledge of textual coding is also required.

Other nominally visual programming languages are *Visual Basic* (Patrick et al., 2006), *Oracle Application Express (APEX)* (Aust et al., 2011) and *Unified Modeling Language (UML)* (Roff, 2003). These, however, are designed to aid the development of textual programs, rather than being visual programming languages in their own right.

Though the area of visual programming languages has matured after long research, purely visual languages are still not generally used as everyday programming tools as software developers have a strong tendency to keep to well-established textual languages (Erwig and Meyer, 1995; Banyasad and Cox, 2013). In brief, hybrid visual languages which integrate visual languages with textual languages are more likely to meet the requirements of real-world software development than highly ambitious purely visual languages. In such hy-

brid languages, each notation supports the other where it is superior. Examples can be found in the domains of logical, functional and procedural languages. The next section will review the area of visual programming for functional languages.

2.5 Visual Functional Programming Languages

2.5.1 Motivation

Using dataflow programming techniques various visual languages have successfully developed many systems such as signal processing, image processing and instrumentation (Bier et al., 1990; Rasure and Williams, 1991). Visual programming languages speed up learning and help understanding of a new programming language for both end-users and experienced programmers as they attempt to extend or modify an existing system or to build a new one (Browne et al., 1995). This indicates that if visual techniques can be applied to a functional programming system (currently textual), they can offer end-users the possibility to learn functional languages more quickly.

2.5.2 A Visual Functional Programming Language : Visual Haskell

This section describes a visual functional programming language, Visual Haskell (Reekie, 1994). Visual Haskell is a visual programming system for Haskell. A recent, nominally visual, programming system to support Haskell is also called

Visual Haskell (Angelov and Marlow, 2005), which is a Haskell development system to support textual programs, rather than visual ones. No other attempts to create a visual programming system for Haskell have been found in any research of computer science literature. In Visual Haskell by Reekie, nodes represent function applications, arguments, operators, etc, and arcs represent dataflow between them.

Functions

The Visual Haskell definition of the standard `map` function is shown in Figure 2.14. The icon defined for the function is displayed next to the text “map”. Its two inputs and one output are indicated by the triangular pads on the outside of the box, and its two clauses are stacked one above the other. Each clause shows its argument patterns immediately inside the box. Annotations are used to aid understanding the program. The application of a function `f` is shown as a plain box labeled with its name, and list-carrying arcs are decorated with an asterisk.

Types

Arcs are optionally annotated with their types. List carrying arcs are decorated with an asterisk and streams (infinite lists) with circles (see Figure 2.15).

Two objects are connected by drawing an arc from a source pad of one object to a sink pad of the other. The pads are the triangular, rectangular and

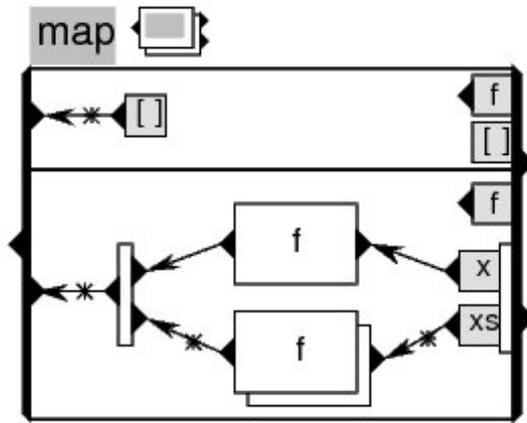


Figure 2.14: The Visual Haskell definition of map.

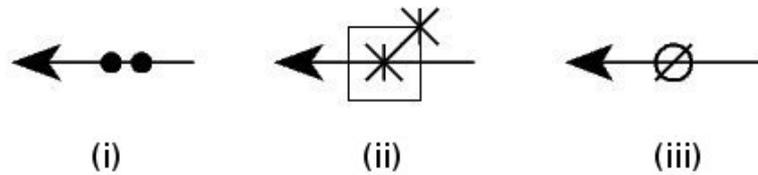


Figure 2.15: Illustrating type annotation in Visual Haskell: i) (a, b) ; ii) $[[a]]$; iii) Stream (Vector a).

half circle shapes on the outside of the main enclosing box. Figure 2.16 shows source and sink pads.



Figure 2.16: a) Source pads; b) Sink pads.

Data arcs, which are used for function application are drawn with an arrow (see Figure 2.17a); binding arcs, which are used in pattern matching parameters are drawn without an arrow (see Figure 2.17b). Two objects are attached if they are physically located so that at least one point of each is at the same

physical location (see Figure 2.17c).

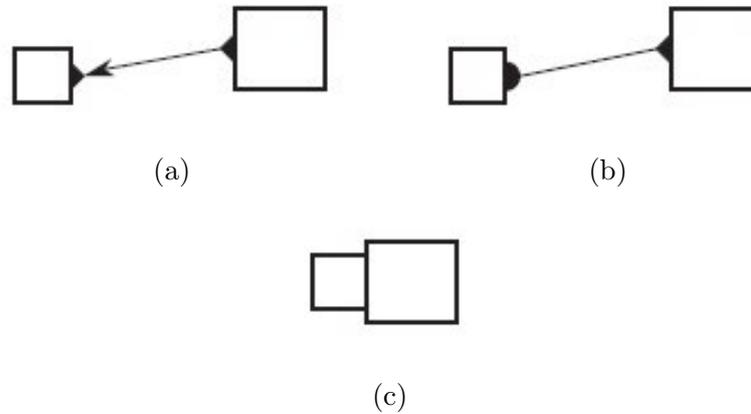


Figure 2.17: a) A data arc; b) A binding arc; c) Attached objects.

A variable or named pattern object can have more than one arc connected to its output pad. Such an object is called shared. For example, in Figure 2.18, the variable `x` is shared.



Figure 2.18: Shared object.

Higher-order functions

The above example of `map` (in Figure 2.14) shows how a functional argument is shown in the function definition. The application of a functional argument is displayed in the same way as any other function application in a plain box labeled with its name. Visual Haskell also helps the visualization of the *pattern of computation* of higher order functions.

Multiple arguments and currying

Any function application has one or more argument slots. If an argument is supplied to a pad which has a slot, and it is “able to fit into” the slot, then the argument is displayed in the box and the corresponding pad is not displayed. Annotation is used to indicate function values: the output pad of an expression producing a function-valued result is rectangular rather than triangular.

Figure 2.19a shows the application `map f` in the core syntax, and Figure 2.19b shows the application using the `map`’s icon and with the name of the argument, `f`, placed into the argument slot. Larger arguments can also fit (see Figure 2.19b).

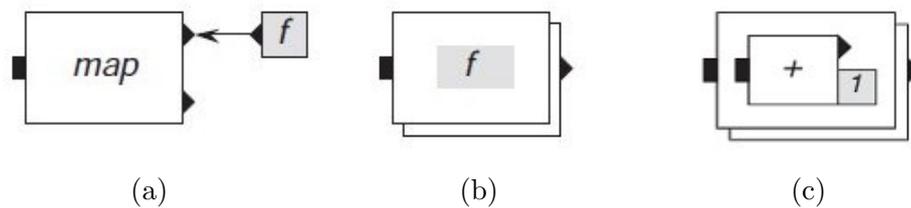


Figure 2.19: Currying.

Local definitions

A `let` expression is treated as an unguarded scoping expression with a null pattern (null pattern means it has no patterns). The use of a null pattern can be seen to be merely a mechanism to fit all Haskell constructs into a single visual construct.

2.5.3 An Example Visual Haskell Program

A Visual Haskell illustration of the Haskell implementation of the circuit example (see section 2.2.3) can be drawn as in Figure 2.20.

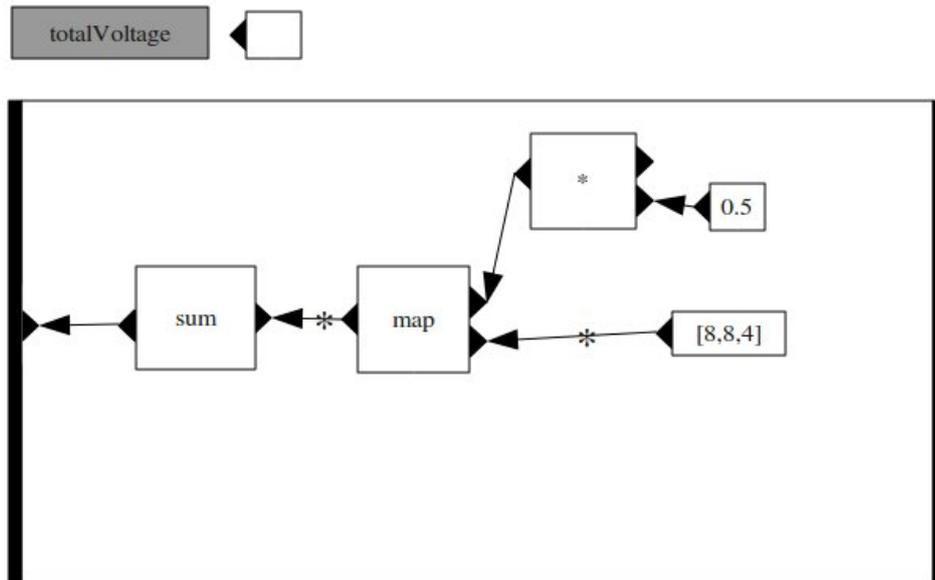


Figure 2.20: Illustrating the series circuit (in Figure 2.1) implementation in Visual Haskell.

Visual Haskell shows some ways to express Haskell syntax in a visual way. However, it lacks the use of HCI techniques in its design. The icons used to show types are not generalized and only a few types can be shown using icons. Higher-order functions need to be recognized without knowing the meaning of the specific icon. They are not easy to understand by looking at the dataflow graph. In the original version of Visual Haskell it was not specified how interaction could be carried out in visual programming. Visual Haskell is more of a visualization tool than a visual programming language. Another problem is that it does not display any error messages. HASKEU is an attempt to

overcome the limitations of Visual Haskell and HASKEU makes a functional programming system easier to learn for end-users.

2.5.4 Other Visual Functional Programming Languages

ML is a general-purpose textual functional programming language and one of the major languages in the ML family is CAML (Gordon, 2000). CAMLFLOW (S'erot, 2000) is a custom CAML to data-flow graph (DFG) compiler. It allows large and complex DFGs to be described in a textual and concise manner, using the facilities of the CAML LIGHT functional language. The main originality of CAMLFLOW lies in its ability to define higher-order polymorphic graph patterns. It offers powerful facilities for describing DFGs, and it has abstraction capabilities. Functional values can be used to define and manipulate sub-graphs. It offers the possibility to declare external types and functions in so-called interface files (.mli in CAML). It also offers multi-output function and product types, conditional sub-graphs, recursive values and data parallelism. This research has taken ideas from CAMLFLOW implementation, specifically how parsing and type-checking are performed to produce a type-annotated abstract syntax tree (see Section 5.2.2).

Arrowized Functional Reactive Programming (AFRP) is a form of FRP that uses the arrow combinators (Hughes, 2000) to solve the problems of time- and space-leaks in a radical way: the programmer may only build signal function using a certain signal functions that maps signals to signals.

`SF a b = Signal a -> Signal b`

The representation of the type `SF` is hidden (i.e. `SF` is abstract), so one cannot directly build signal functions or apply them to signals. Instead AFRP provides a set of primitive signal functions and combinators. There is a set of eight combinators that are often used in AFRP programming. Figure 2.21 shows the visual “wiring of arguments” of the five combinators. However, this is only a concept of how AFRP programs could be visualized, not a visual programming tool.

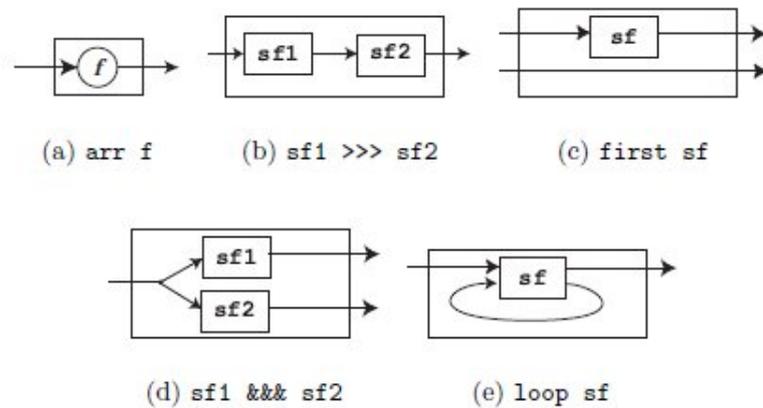


Figure 2.21: Commonly Used Arrow Combinators.

In the early research for this thesis, an attempt was made to develop a visual programming tool for this Arrowized Functional Reactive Programming, but it was decided to broaden the research to general purpose functional programming rather than a domain specific one.

2.6 Conclusion

This chapter began by showing the impact of visual languages on end-users who are learning programming languages. It has included an overview of the functional programming language, Haskell and looked at visual programs in different notations, including an early Haskell visual system, and shown an example program in different systems. So far, there has not been any suitable functional programming system in the literature that has targeted end-users. This chapter has contained an investigation of the suitability of both the existing textual programming approach on end-user development, and the suitability of a visual programming approach. In the next chapter the implementation will be shown which sets the stage to produce a single system that combines textual and visual programming using a Model-View-Controller (MVC) design pattern. This single system will allow end-users to program using both notations at the same time, and to compare the effectiveness of each method.

Chapter 3

The Model View Controller as a Functional Reactive Program

The Model-View-Controller (MVC) design pattern is very useful and widely used for implementing user interfaces in object-oriented programming languages. The Functional Reactive Programming (FRP) framework is an elegant one for implementing reactive systems (including user interfaces) in a purely functional manner. This chapter presents the fact that the MVC design pattern matches to the FRP framework very closely.

3.1 Introduction

The purpose of many modern computer systems is to manage the flow (retrieve and store) of information between the data store and the user interface, and a natural approach to implementing such systems is to tie these two pieces together to reduce the amount of coding and to improve application performance.

However, two significant problems of such a natural approach are — the user interface tends to change much more frequently than the data storage system, and business applications tend to incorporate business logic that goes far beyond data transmission. The Model-View-Controller (MVC) pattern separates a system into the modelling of the domain, the presentation, and the actions based on user input (Burbeck, 1987). The MVC is very popular for creating web applications or software because this MVC structure ensures efficiency and consistency. Many of the most popular frameworks use the MVC architecture, including ASP.NET (Freeman, 2012), CodeIgniter (Argudo, 2009), Zend (Allen, Rob and Lo, Nick and Brown, Steven, 2008), Django (Bennett, 2008), and Ruby on Rails (Tate and Hibbs, 2006).

A computer system that reacts to user actions is a reactive system, and it responds in a timely way to events in its environment. The aim of functional reactive programming (FRP) (Elliott and Hudak, 1997) is to provide a powerful way to describe reactive systems in a functional language. Among the systems based on FRP are: Functional Reactive Animation (Fran), a domain specific language (DSL) for graphics and animation (Elliott and Hudak, 1997); Functional Animation Language (FAL), a framework for drawing graphics and animations on the screen (Hudak, 2000); Functional ROBotics (FROB), a robot programming language embedded in the Haskell programming language (Peterson et al., 1999); Visual Tracking System (FVision), built using FRP to express interaction in a purely functional manner (Reid et al.,

1999); and Functional Reactive User Interface (Fruit), a graphical user interface library for Haskell based on a formal model of user interfaces (Courtney and Elliott, 2001).

This chapter presents how the MVC design pattern may naturally be implemented in the FRP framework, and how a new framework “MVC as FRP” has been implemented with the use of FRP types and combinators. Programmers brought up on object-oriented programming languages who also use the MVC design pattern may benefit from this “MVC as FRP” framework by being able to use an MVC design pattern in functional programming languages. Because it is implemented in a pure functional language, Haskell, this “MVC as FRP” framework has no side-effects. No reporting of a similar purely functional implementation of an MVC framework has been found in the literature. The chapter is organised as follows. Section 3.2 introduces the MVC design pattern, and Section 3.3 introduces the FRP framework. Section 3.4 presents the implementation of “MVC as FRP” framework, and Section 3.5 presents a worked example. Section 3.6 refers to a way of developing user interfaces in command oriented functional programs using the MVC framework, and Section 3.7 concludes.

3.2 The MVC Design Pattern

Originally, the *Model-View-Controller* (*MVC*) was developed as a design pattern for building user interfaces (Fowler, 2002). The key notions in the MVC

are the *model*, which provides a way to represent an object, the *view*, which provides a way to display it, and the *controller*, which provides a way to change it. Usually, there is a single model, with many views and many controllers (Krasner and Pope, 1988; Gamma et al., 1995). The “interface logic” functions allow the views to be developed independently, and the “business rules” functions allow the controllers to be developed independently. All these three key notions in the MVC design pattern are time-varying values. In the implementation of “MVC as FRP”, these key notions will be represented using fundamental FRP constructs and will be shown later in Section 3.4.

As an example of an MVC system, consider an interface that allows a *volume level* to be set using a *slider* or a *dial*. See Figure 3.1. Both the slider (a view/controller) and the dial (also a view/controller) may change the volume level (the model). Once the model is changed, both a view/controllers are updated with the changed model value.

3.3 The FRP Framework

A reactive system is one that responds in a timely way to events in its environment. Examples of such systems might be games that react to controller button presses by moving figures on the screen, music synthesizers that react to keyboard presses by sounding different notes and robots that react to collisions by changing direction. Developing software for reactive systems is a hard

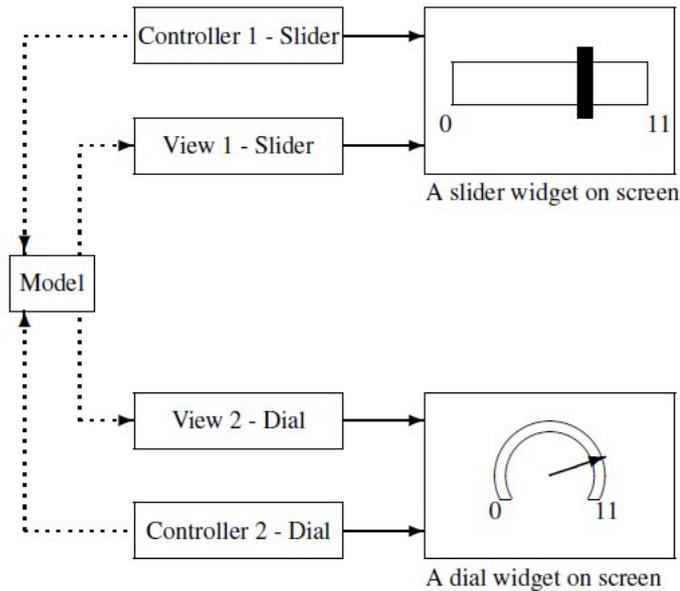


Figure 3.1: MVC Example - a volume controller system

task because it combines the already hard tasks of concurrent, embedded and real-time programming (Pembeci et al., 2002). Current approaches involve the explicit identification and management of concurrent processes by the programmer, often leading to undesirable non-deterministic behaviour, deadlock or livelock (Coulouris and Dollimore, 1988). To use a synchronous dataflow programming language, such as Signal (Le Guernic et al., 1991), Lustre (Caspi et al., 1987), or Lucid (Wadge and Ashcroft, 1985) was a common style to implementing reactive systems. Using lazy lists, Haskell programs can also be built in this style. In the synchronised dataflow model at the top level, a program is represented by a function that yields a stream of requests and accepts a stream of responses. In a lazy language a stream may be represented by a list. The problem with such an approach is one must take care to ensure that a request is always issued before the corresponding response is consumed and the modular program designs that are needed for large scale applications, cannot

always be produced. *Monads* hide this level of detail and are more modular than synchronous streams (Wadler, 1997). Hence the move was made from stream based to monadic IO (O’Sullivan et al., 2008). Monads are commonly used to order sequences of computations. Functional Reactive Programming (FRP) is designed to integrate reactivity in a direct way into the functional programming style and to hide the mechanism that controls time flow under an abstraction layer.

The following sub-sections first give a bit more explanation of Monad and monadic IO and then explain the FRP architecture.

3.3.1 Monad and Monadic IO

Haskell library includes the `Monad` class for working with data types with particular properties (monadic data types), as explained below. So, it will be better to introduce the Haskell data type before explaining `Monad`. The Haskell data type can be used to allow a value with an added context. For example, `"John"` is just a string value, whereas `Student "John"` has some added context. Another example, the integers `0 0 255` are just integer values, whereas `RGB 0 0 255` has some added context. A data declaration for `Colour` can be given as:

```
data Colour = RGB Int Int Int
           | Red
           | Green
           | Blue
           | Undefined
```

A data declaration consists of two parts - a type constructor and data constructor(s). The left hand side of the equal sign is the type constructor and the right hand side of the equal sign has the data constructor(s). Each data constructor is separated by a | sign. A type constructor can be used in a type signature and data constructors can be used where a value is expected, as shown in the following example for the `rgbToValue` function type signature and implementation.

```
rgbToColour    :: Colour -> Colour
rgbToColour (RGB 255 0 0)    = Red
rgbToColour (RGB 0 255 0)    = Green
rgbToColour (RGB 0 0 255)    = Blue
rgbToColour _                = Undefined
```

Type parameters can also be used in a data type declaration. For example, the type parameter `a` has been used in the following tree type declaration:

```
data Tree a    = Leaf a
               | Branch (Tree a) (Tree a)
```

So, if `Tree String` is used in a type signature, it will denote a tree with string elements. Again, here `Tree`, `Leaf` and `Branch` are the contexts and `a` is the type of values taken.

In Haskell, it is possible to implement polymorphic functions that can work on different data types. Haskell *type classes* can be used to implement such a

polymorphism at a higher level than possible in other languages. The purpose of type classes is to make sure that certain operations are available for values of chosen data types. For example, the `Functor` type class is shown below, where `fmap` is the operation which can be used to map a function `(a -> b)` to any data type with context `f` and value of type `a`.

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

So, the `fmap` function for the `Tree` data type can be implemented as follows:

```
instance Functor Tree where
    fmap f (Leaf x)           = Leaf (f x)
    fmap f (Branch left right) = Branch (fmap f left) (fmap f right)
```

Note that without type classes it was necessary to implement different functions for different data types (e.g. `fmapTree` to map over `Tree`, `fmapGraph` to map over `Graph` etc.).

Similar to `Functor`, `Applicative` and `Monad` are type classes which contain some operations/functions that can be used with data types defined as `Applicative` and `Monad` below:

```
class (Functor f) => Applicative f where
    pure      :: a -> f a
    (<*>)     :: f (a -> b) -> f a -> f b
```

In the above three lines, the code `(Functor f) =>` in the first line introduces a class constraint which means that to make a type constructor an instance of

the `Applicative` type class, it has to be in `Functor` first. The `Applicative` class is used to add strength to the `Functor` class. For example, the result of this expression `fmap (*5) (Leaf 3)` is `Leaf 15` which can be used as a second argument of the `fmap`. However, the expression `fmap (*) (Leaf 3)` gives a result of type `Leaf (Int -> Int)` which cannot be used with `fmap`. This is a typical scenario when `Applicative` is very useful and `Leaf (Int -> Int)` can be used as a first argument of `<*>`.

The applicative library in Haskell also defines `<$>` which is a synonym of `fmap` in the `Functor` class:

```
(<$>) :: Functor f => (a -> b) -> f a -> f b
```

Another operator `<$` in the applicative library is used to replace a value in a context:

```
(<$) :: Functor f => a -> f b -> f a
```

The function `liftA2` in the applicative library is used to lift a binary function to actions:

```
liftA2 :: Applicative f => (a -> b -> c) -> f a -> f b -> f c
```

The following is the class definition of `Monad`:

```

class Monad m where
    (>>=)  :: m a -> (a -> m b) -> m b
    (>>)   :: m a -> m b -> m b
    return :: a -> m a
    fail   :: String -> m a

```

Monad `return`, like `Applicative pure`, takes a value from a plain type and puts it in a context (e.g. `return "John" => Student "John"` puts the `String` in the `Student` context).

The operators `>>=` and `>>` are used to sequence two functions. Using `>>=`, if a function (e.g. `f1`) result type is `m a` (first argument of `>>=`), and the value of type `a` is fed to another function (e.g. `f2`) of type `(a -> m b)` (second argument of `>>=`), then these two functions (`f1` and `f2`) are sequenced (`f1 >>= f2`) in a functional way.

In the real world, input/output functions need to be sequenced and to bind input/output value with a context, the `IO a` type is used in Haskell. For example, to take an input from a console, the `getLine` function is used

```
getLine :: IO String
```

This means that `getLine` does some input interaction with the real-world and it returns a string value in the context of `IO`. Similarly, to display some value at a console, the `putStrLn` function is used

```
putStrLn :: String -> IO ()
```

This means that the `putStrLn` function takes a string value as its argument, and does some output interaction with the real-world and returns nothing (Haskell's unit data type `()` is used to denote a nothing value). However, sometimes an IO function has something to feed on to a subsequent IO function. For example, in `getLine >>= putStrLn`, the `getLine` passes its result as an argument to the next function `putStrLn` which will generate an output.

But in many cases an IO function does not need to pass an argument to the next function in the queue. This is true for the four functions below which need to be sequenced in order:

1. `putStrLn "Please enter the user name"`
2. `getLine`
3. `putStrLn "Please enter the password"`
4. `getLine`

None of the above functions pass a value to the next function. In order to sequence these functions in a functional way, the Haskell IO type has a hidden value which is passed to the next function. The type synonym IO is defined (using a `RealWorld` type for the hidden value) in the following way:

```
type IO a = RealWorld -> (a, RealWorld)
```

The value is kept hidden, rather than having an extra data constructor, otherwise passing it between functions would look unsightly and confusing to the

programmer. The >> operator implicitly passes the hidden values between functions (IO actions) and the definition of >> can be thought as below:

```
action1 >> action2 =
    action
  where
    action world0 =      let    (a, world1)    = action1 world0
                           (b, world2)    = action2 world1
                           in    (b, world2)
```

Using >>, the above four functions can be sequenced as shown below.

```
putStrLn "Please enter the user name" >>
getLine >>
putStrLn "Please enter the password" >>
getLine
```

If only the hidden value needs to be passed, the >> operator is used . If both the hidden value and the actual return value of a IO function needs to be passed, then the >>= operator is used. The do notation is a sugared version of monad operators enabling the above four functions to be sequenced as shown below

```
do
  putStrLn "Please enter the user name"
  getLine
  putStrLn "Please enter the password"
  getLine
```

The `<-` operator can be used within a `do` block to unpack an `IO` value from the context (e.g. `a <- getLine` where `a` is a value without context).

The value within an `IO Monad` can be accessed only by using the `Monad` operators/functions. Also, any function which uses an `IO` function has to be an `IO` function too. According to monad law, all the four functions as defined in the `Monad` class have the result type either `m a` or `m b`. This way, Haskell forces a function to be either pure or impure (a pure function is a function that has no side-effects, the return value is only determined by its input arguments). Once a function has worked with an impure value (a real-world value i.e. not an argument value), then it becomes an impure function.

`Functor`, `Monad` and `Applicative` are all used in the definition of FRP. The full list of all functions of these classes and their default instances are shown in the Appendix F. The following section gives an overview of the FRP framework. The core of FRP is that, as a reactive system responds in a timely way to events in its environment, a time value needs to be passed between functions to implement such an environment. FRP hides these details of passing the time from the programmer (just as the `>>` and `>>=` operators hide the details of sequencing in an `IO monad`). While using `Monad`, the perception of the programmer is that they are programming in a imperative way, the monadic `do` blocks especially look like chunks of imperative code. FRP brings a declarative feel back to the programmer.

3.3.2 The FRP

Originally, *Functional Reactive Programming (FRP)* was developed as a framework for building reactive multimedia graphics and animations (Elliott and Hudak, 1997). Now it is applied more generally for robot programming, visual tracking, user interfaces and music synthesizers (Peterson et al., 1999; Reid et al., 1999; Courtney and Elliott, 2001; Giorgidze and Nilsson, 2008). One recent FRP implementation is *reactive-banana* (Apfelmus, 2012) which is used here in the implementation of MVC. There is a rich set of operators (combinators) in this FRP library but only the operators relevant to this chapter are briefly described here. The full list of *reactive-banana* operators (combinators) can be found on the Haskell website (Haskell Website - reactive-banana, n.d.). The key notions in FRP are *behaviours* and *events* (Wan and Hudak, 2000). The type `Behavior`, which can be thought of as

```
type Behavior t a = Time -> a
```

is that of behaviours — time-varying values of type `a`. The `Time` can be thought of as synonym of integer to represent time. Some examples of behaviours might be `animation`, represented by the value `animation :: Behavior t Picture`, and the speed of a car in a racing game, of type `speed :: Behavior t Int`. The type `Event`, which can be thought of as

```
type Event t a = [(Time,a)]
```

is that of events — a time-ordered sequence of event occurrences of type `a`. Basic events might be left mouse button depressions of type `lbp :: Event t ()`

and key strokes, represented by the value `key :: Event t Char`. A basic event like `lbp` (or `key`) should be thought of as an event sequence containing all of the left button depressions (or key presses), not just the last one.

A rich set of operators (combinators) can be used to create new behaviours and events. For example, new behaviours and events can be created by trimming existing ones by start time (Elliott, 2008). Without trimming, a new phase would begin by responding to all the old input instead of from that new input directly after the old, which would result in incorrect behaviour. Also, without trimming, a new phase would contain all the previous inputs, which would result in inefficient behaviour as these old inputs would need responding to before reacting to the new inputs. The trimming functions discard all event occurrences upto and including those occurring at a given start time. The relevant combinators to create new behaviours and events by trimming are:

```

trimB      ::      Behavior t a
           ->      Moment t (forall s. Moment s (Behavior s a))

trimE      ::      Event t a
           ->      Moment t (forall s. Moment s (Event s a))

```

where `Behaviour` or `Event` are tagged with a start time by wrapping it up with a `Moment` type. The `Moment` type can be thought of as

```
Moment t a = t -> (a, t)
```

which is similar to `IO a = RealWorld -> (a, RealWorld)`. In this way it can pass the `t` value around. As the `Moment` type has the parameter `t`, which

is the type of the time value, the parameters `t` in `Behaviour` and `Event` are superfluous in the *reactive-banana* library, though used internally. The occurrence of time in `Moment` types are given implicitly in *reactive-banana*. `Moment` is a monad so that it hides the “unsightly plumbing” (Wadler, 1995) necessary to thread time values through the computation, as described in the previous section. `Moment` is also an *applicative functor*, so that the `Moment` types can be mapped over.

The `forall` used in the above trimming functions is a keyword. It is a *GHC/Hugs* extension and can be used to explicitly bring type variables into scope. In formal logic, `forall` (or \forall) is a quantifier and it quantifies whatever comes after it. For example, $\forall x$ means that what follows is true for every `x` of that type, e.g. For example, $\forall x, x^2 \geq 0$. In Haskell, `forall` allows the programmer to specify the type variables in a signature definition should be scoped over the body of that definition. The examples below demonstrate a typical usage of `forall`. In the code below, the local functions `listSorted` and `listNubbed` do not have explicit type signatures. They use the library functions, `zip :: [a] -> [b] -> [(a,b)]` (takes two list and makes a new list of tuples containing elements of both lists occurring at the same position), `sort :: Ord a => [a] -> [a]` (implements a sorting algorithm) and `nub :: Eq a => [a] -> [a]` (removes duplicates elements from a list).

```
testForAll          :: Ord a => [a] -> [(a, a)]
testForAll aList    = zip listSorted listNubbed
```

```

where

listSorted = sort aList

listNubbed = nub aList

```

If the two local functions were given type signatures as below,

```

testForAll      :: Ord a => [a] -> [(a, a)]

testForAll aList = zip listSorted listNubbed

where

listSorted :: [a]

listSorted = sort aList

listNubbed :: [a]

listNubbed = nub aList

```

then the Haskell compiler will give an error message that “ ‘a’ is a rigid type variable bound by the type signature for `testForAll`”. This means that Haskell does not recognize that we want the `a`’s in the inner definitions to be the same as the `a` in the outer definition. The two occurrence of `a` in the inner definition are not recognized as the same type by the Haskell system, either. The `forall` construct enables `a` to be scoped over the whole definition, including the `where` clause, as follows:

```

testForAll      :: forall a. Ord a => [a] -> [(a, a)]

testForAll aList = zip listSorted listNubbed

where

listSorted :: [a]

```

```

listSorted = sort aList

listNubbed :: [a]

listNubbed = nub aList

```

A useful basic combinator in FRP is `accumB` which starts with an initial value and combines it with incoming events (Hudak, 2000). Using `accumB`, the function associated with each event is applied to the last value of the behaviour, to yield a new behaviour. For example, whenever the up key is pressed, the speedometer count is incremented to yield a new behaviour. The function `union` is used to concatenate streams of event mapping functions. For example, this behaviour `speedometer` in a car racing game:

```

speedometer =
    0 'accumB'
        ((+1) <$> upKeyPressed)
        'union'
        ((subtract 1) <$> downKeyPressed)

```

is, as the name implies, a speedometer; its value is initially 0 and increases by 1 each time the event up arrow key is pressed, and decreases by 1 whenever the down arrow key is pressed. The *applicative functor* `<$>` essentially maps a function over a stream of events. In this example it maps the function `(+1)` over the event `upKeyPressed`. Later in Section 3.4, the use of the combinator `accumB` to implement the `mvc` function to control the model, views and controllers in a MVC system will be demonstrated.

Another combinator `changes` enables explicit control over updates of behaviours. It can observe when a `Behavior` changes and create an event firing which can produce an output.

```
changes :: Frameworks t => Behavior t a -> Moment t (Event t a)
```

For example, to display the value of a `speedometer` (a behaviour) in a speed gauge, the explicit control of the speedometer is necessary. Every time a speedometer value is changed, to explicitly tell the system to change the gauge, the code below would be written:

```
display speedometer =  
    changeGauge <$> (changes speedometer)
```

3.4 An implementation of the “MVC as FRP” framework

This section will demonstrate that the MVC design pattern can be characterised by the FRP framework in functional programming, hence the “MVC as FRP” framework has been implemented with the use of the essence of FRP as below:

A model corresponds to a behaviour

```
type Model t m = Moment t (Behavior t m)
```

for time-varying values of type m , at times of type t . Consider the volume control example given in Section 3.2. A volume can be changed at different times, and the model of the volume control system is a behaviour. The volume level can take the value of any integer between 0 to 11, and the type of m can be an integer.

A view corresponds to a behaviour.

```
type View t v = Moment t (Behavior t v)
```

for displays of type v , at times of type t . In the volume control example, there would be two views — dial and slider. The view of a dial is a behaviour that changes the dial view at certain times, and the same is true for the slider. A view changes when the model changes and the view has a type IO (although a view is intended to have type IO , this is not enforced by this definition). The complete view (consisting of all individual views) has to be isomorphic to the model. It is a good software engineering practice for the complete view to be isomorphic to the model.

A controller corresponds to an event mapping function.

```
type Controller t m
    = Moment t (Event t (m -> m))
```

for event-triggered updaters of values of type `m`, at times of type `t`. In the volume control example, if any event on the dial (click on dial) changes the volume level, it should change the underlying model first, then both views have to be updated with the new model value. So, the controller of the dial is an event mapping function that changes the model value.

The business rule is the function that is mapped with an event in the controller and it changes the model value. A business rule function has the type

```
type BusinessRule e m
    = e -> m -> m
```

where `e` is the event value and `m` is the model value. To give an example, if a click is made on the dial on mark 5, it sets the event value `e` equal to 5, and the model value has to be replaced by 5. The same should happen for a slider. So, a common business rule for both the controller of the slider and the dial can be coded as below:

```
brSetVolume :: BusinessRule Int ModelType
brSetVolume e m = e
```

If the business rule needs to ensure that if the event value exceeds 11 the model value is not changed, then the function becomes:

```

brSetVolume :: BusinessRule Int ModelType

brSetVolume e m =    if (e > 11)

                    then m

                    else e

```

An interface logic function maps from one view to another and has the type

```

type InterfaceLogic v m

    = v -> m -> v

```

where *v* is the view value and *m* is the model value. So, if the model value in the volume control system is 5, then both the dial and slider view should both display 5. An interface logic for a dial widget can be coded as below:

```

ilSetVolumeDial :: InterfaceLogic ViewType ModelType

ilSetVolumeDial v m =

    v >>= (\v -> set (vDial v) [selection := m]) >> v

```

A portable and a native GUI library for Haskell is *wxHaskell* (Leijen, 2004). The *wxHaskell* library uses the `:=` operator to combine a value (*a*) with an attribute (denoted by the data type `Attr w a`, *w* is the widget type) and the combination of an attribute with a value is called a property (denoted by the data type `Prop w`)

```

(:=) :: Attr w a -> a -> Prop w

```

The `set` in *wxHaskell* is a function to assign a list of properties to a widget.

```

set :: w -> [Prop w] -> IO ()

```

If a dial colour also needs to be changed when its value is greater than 7, the interface logic would be as below:

```

ilSetVolumeDial :: InterfaceLogic ViewType ModelType
ilSetVolumeDial v m =
    if (m > 7)
    then
        v >>= (\v -> set (vDial v) [selection := m])    >>
        v >>= (\v -> set (vDial v) [color := red])      >> v
    else
        v >>= (\v -> set (vDial v) [selection := m])    >>
        v >>= (\v -> set (vDial v) [color := white])   >> v

```

In this functional MVC implementation, the following two functions `controller` and `view` do the decoupling of a user-control and separate it into a controller and a view. The function `controller` maps a business rule with a specific event and returns a `Controller` type. The `m` and `e` in the type declaration are the types of model value and event value respectively. `Frameworks t`, as defined in *reactive-banana*, is the class constraint on the type parameter `t` of the `Moment` monad. Having `Frameworks t` as a constraint on a function type indicates that any input and/or output operation within the function can be added to an event network.

```

controller          :: Frameworks t
                   => Event t e
                   -> (BusinessRule e m)
                   -> Controller t m
controller ev br    = return (br <$> ev)

```

The following function `unionController` can be used to concatenate two controllers. The function `liftA2` is used here to lift the `union` function to concatenate streams of event wrapped by `Moment` as in the `Controller` type.

```
unionController      :: Frameworks t
                    => Controller t m
                    -> Controller t m
                    -> Controller t m

unionController      = liftA2 union
```

The following function `mergeController` can be used to concatenate a list of controllers. The library function `foldr1 :: (a -> a -> a) -> [a] -> a` is used here to repeatedly apply a function to reduce a list to a value as follows: it takes the last two items of the list (second argument) and applies the function (first argument), then it takes the result and the third item from the end of the list, applies the function, and so on.

```
mergeController      :: Frameworks t
                    => [Controller t m]
                    -> Controller t m

mergeController = foldr1 unionController
```

The function `view` shown below maps the interface logic with view and model and returns a `View` type. The `v` and `m` are the types of the view value and the model value respectively.

```

view      :: Frameworks t
          => View t v
          -> Model t m
          -> InterfaceLogic v m
          -> View t v

view v m il =
    v >>= (\v -> (m >>= (\m -> (return $ il <$> v <*> m))))

```

The following function `unionView` can be used to sequence two views of type `IO`:

```

unionView :: Frameworks t
          => View t (IO v)
          -> View t (IO v)
          -> View t (IO v)

unionView v1 v2 =
    v1 >>= (\v1 -> changes v1)
      >>= (\ev -> reactimate $ (\v -> v >> return ()) <$> ev) >>
    v2 >>= (\v2 -> changes v2)
      >>= (\ev -> reactimate $ (\v -> v >> return ()) <$> ev) >>
    v2

```

Here, the `reactimate` function, as defined in *reactive-banana*, is an interface between event functions and the views (of a potentially impure external world), and it executes an `IO` action whenever an event occurs to display the new view at that moment.

The following function `mergeView` can be used to sequence a list of views of type `IO`:

```

mergeView      :: Frameworks t
                => [View t (IO v)]
                -> View t (IO v)

mergeView lstV =
    foldr1 unionView lstV

```

The `mvc` function below takes an initial model value, an initial view value, a concatenated view mapping function of all views and a concatenated controller of all controllers in the system, and then combines the model, the view and the controller. This is done in such a way that the user does not have to pay any attention to the individual controller or view in order to change the model value or to update a view, and it makes the whole MVC system operate successfully. With the use of the FRP combinator `accumB`, the model value is updated every time a controller is in action. More precisely, on the occurrence of any event, the business rule associated with that event is applied to the last value of the model, and the business rule returns an updated model value. Using another `accumB`, the view value is updated every time a model value is changed.

```

mvc      :: Frameworks t => m -> v
         -> (View t v -> Model t m -> View t v)
         -> Controller t m -> View t v

mvc minit vinit fnv c =
    let    m = c    >>= (\c      -> pure $ accumB minit $ c)
          v = m    >>= (\m      -> changes m)
          >>= (\ev   -> pure $ accumB vinit $ ((\v -> v) <$ ev))
    in    fnv v m

```

3.5 An Example

The volume control example given in Section 3.2 is implemented here using the “MVC as FRP” framework given in Section 3.4. The *wxHaskell* library is used to build the volume control example GUI. The full list of *wxHaskell* functions can be found on the Haskell website (Haskell Website - wxHaskell, n.d.).

In this volume control system, the volume level takes the integer values between 0 to 11. So, the following type of `ModelType` can be used for the type parameter `m` of the underlying `Model`:

```
type ModelType = Int
```

The `ViewType` is the complete view of the volume control system. It consists of wxHaskell widgets (a slider and a dial) and has the type `IO`:

```
type ViewType =  
    IO ViewOfWxWidgets  
  
data ViewOfWxWidgets =  
    ViewOfWxWidgets  
    {  
        vSlider :: Slider (),  
        vDial :: Dial ()  
    }
```

Events in the existing event-based framework, *wxHaskell*, need to be adjusted in order to be used in the “MVC as FRP” framework. In contrast to a *wxHaskell* event, an FRP event (see Section 3.3) is more declarative and is an event stream, not just a single event. The *wxHaskell* events can be represented as FRP events using the functions, `event0` and `event1` as below, and *wxHaskell* widget views are updated via the library function `sink`. The function `event0` represents any *wxHaskell* event with no parameter, and the function `event1` represents any *wxHaskell* event with one parameter as a FRP event. These three functions, `event0`, `event1` and `sink`, are defined in the *reactive-banana-wx* library. The *reactive-banana-wx* library provides functionalities to use *reactive-banana* FRP with *wxHaskell*.

The following coding of the `eventSelect` function shows how a selection event of any *wxHaskell* widget can be adjusted to an FRP event. The `Selecting` and `Selection` classes in the *wxHaskell* library are used here as type constraints. The widgets which are instances of the `Selecting` class fire a `select` event when an item is selected. The widgets which are instances of the `Selection` class have their values changed or retrieved by calling the `selection` attribute function. The operator `(<@) :: f a -> g b -> g a` in the *reactive-banana* `Apply` class is used to apply a time-varying behaviour to a stream of events.

```

eventSelect    ::      (Frameworks t, Selecting w, Selection w)
                  =>      w -> Moment t (Event t Int)
eventSelect w = do      eSelect <- event0 w select

```

```

    b <- (behavior w selection)
    return (b <@ eSelect)

```

The business logic for both the slider and dial controllers are the same, and they just replace the model value by the event value of the slider or dial

```

brSetVolume :: BusinessRule Int ModelType

brSetVolume e _ = e

```

The interface logic for both slider and dial change the display of slider and dial with the model value

```

ilSetVolumeSlider :: InterfaceLogic ViewType ModelType

ilSetVolumeSlider v m =
    v >>= (\v -> set (vSlider v) [selection := m]) >> v

ilSetVolumeDial :: InterfaceLogic ViewType ModelType

ilSetVolumeDial v m =
    v >>= (\v -> set (vDial v) [selection := m]) >> v

```

Below, instances of the slider and the dial widget are created, and then positioned in a window panel. The function `liftIONow`, as defined in *reactive-banana*, lifts an IO action to a `Moment` monad. The `hslider` function creates a horizontal slider in a window `win` with a specified minimum (0) and maximum (11). The second argument of `hslider` is set to `True` to show labels (minimum, maximum, and current value).

```

s          <- liftIONow
           $ hslider win True 0 11 [selection := 0]

d          <- liftIONow
           $ makeDial win 0 11 []

```

Then, the business logic and interface logic (created above) of the slider and dial are mapped to create the controllers and the views

```
evSldComm    <- eventCommand s
evDialSel    <- eventSelect d
let cSld     = controller evSldComm brSetVolume
let cDial    = controller evDialSel brSetVolume
let vSld     = \v m -> view v m ilSetVolumeSlider
let vDial    = \v m -> view v m ilSetVolumeDial
```

The two lists of controllers and views are merged to a single view and a single controller, and with the initial model value (e.g. 0) and the initial view value, are passed to the `mvc` function

```
let lstC     = [cSld, cDial]
let lstV     = [vSld, vDial]
let viewInit = return $ ViewOfWxWidgets s d
let c        = mergeController lstC
let fnv      = \v m -> mergeView [vw v m | vw <- lstV]
mvc modelInit viewInit fnv c
```

Now, the volume control reactive system is ready to use (see Figure 3.2).

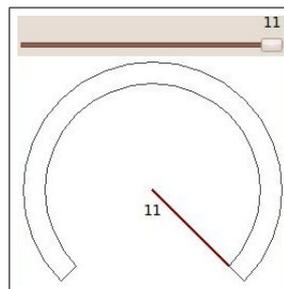


Figure 3.2: The volume control system using the MVC-FRP architecture.

3.6 Related Work

Alley Stoughton has devised a way of developing user interfaces in command-oriented functional programs using the Model-View-Controller framework (Stoughton, 2008). However, her design is not purely functional. When communicating with users, her design has side-effects, and the widgets of graphical views have a state. The MVC framework approached in this chapter has all the advantages of the FRP (safe programming, efficient programming and composability), and there are no side-effects.

Functional Reactive Animation (Fran) (Elliott and Hudak, 1997), Functional Animation Language (FAL) (Hudak, 2000), Functional Reactive User Interface (Fruit) (Courtney and Elliott, 2001), and *reactive-banana* (Apfelmus, 2012) are the systems based on FRP. They have been designed to aid GUI programming in pure functional languages. However, none of them say explicitly that their designs can serve MVC. The “MVC as FRP” design is an abstraction layer to FRP and has the defined types and functions necessary to show MVC explicitly. This design has also used higher-order functions and polymorphic types so that it is not dependent on any specific GUI library.

In this PhD work (Alam, 2014), the Haskell programming environment, HASKEU, has been implemented to aid end-users to learn Haskell programming using this “MVC as FRP”. The HASKEU programming system has been designed as an MVC system with both visual and textual interfaces, and hence

changes propagate between the visual and textual interfaces, so that they are always consistent.

3.7 Conclusion

This chapter described a software architecture — “MVC as FRP” — for functional programs which makes the development of reactive systems easier, as it has all the strengths of both the MVC and FRP frameworks. It was tested by implementing a complete program that was chosen to be complex enough to cover all eventualities in the Model-View-Controller architecture and in Functional Reactive Programming, but also simple enough to be suitable as an example for research and teaching. Building on the textual and visual display of Haskell programs in HASKEU, it is a natural step, using “MVC as FRP” to implement a GUI builder as provided in IDEs for languages such as Java or Visual Basic, and it is a future project. As is the case in imperative languages, such a GUI builder can be seen as an abstraction layer for the implemented “MVC as an FRP”, but now in a functional development environment.

Chapter 4

The Design of HASKEU

In Section 2.1, the capabilities of end-user programmers were listed as:

- (a) they may not be expert programmers;
- (b) they find large programs daunting;
- (c) they may be unskilled, and make false steps;
- (d) they may be uncertain, and make false starts.

HASKEU caters for end-user programmers by providing:

- support for visual and textual programming which helps with end-user capabilities (a) and (b) above;
- support for exploratory programming which helps with end-user capabilities (c) and (d) above.

The support contains the following topics which form the structure of this chapter:

- visual programming which helps with end-user capability (a) above;
- one function per page which helps with end-user capability (b) above;
- error reporting which helps with end-user capability (c) above;
- infinite undo which helps with end-user capability (c) and (d) above;
- textual programming which will make HASKEU a more useful system when end-user's expertise increases;

The design of HASKEU makes extensive use of *Human-Computer Interaction (HCI)* techniques (Card et al., 1983). The following 8 golden rules of user interface design (Shneiderman and Plaisant, 2004) are applied to the design of HASKEU in order to improve the usability of the system:

1. Strive for consistency — the user interface is consistent for all the operations.
2. Enable frequent users to use shortcuts — users do not have to work very hard to go from one point to another, especially for those users who use the interface regularly.
3. Offer informative feedback — for every action performed by the user, feedback should be provided by the system.
4. Design dialog to yield closure — complicated tasks are require to be split into several steps with a beginning, middle and end.
5. Offer simple error handling — this is to make sure the user does not make serious errors.

6. Permit easy reversal of actions — this provides the facility that the user can undo an error.
7. Support internal locus of control — this lets the experienced users feel that they are in control of the system.
8. Reduce short-term memory load — by designing screens where options are clearly visible, short term memory load can be reduced.

The usage of HCI will be discussed throughout this chapter. The following sections go on to discuss the design in detail.

4.1 Visual Programming

4.1.1 Organization

Five desirable attributes of any data display are — *consistency, efficient information assimilation, minimal memory load, compatibility of display with entry and flexibility of control* (Smith and Mosier, 1986). There are correspondence between these five desirable attributes and the above 8 golden rules. Even though these five desirable attributes pre-date 8 golden rules, there is a broad agreement: both emphasise consistency and reducing short-term memory load. The five desirable attributes emphasise flexibility of control which is in tune with the principles of allowing short-cuts, easy reversal and supporting internal locus of control. Some are not so clear-cut but not incompatible (e.g. Feedback in 8 golden rules and compatibility of display with entry in five desirable attributes). The golden rule of designing dialogue to yield closure supports ef-

efficient information assimilation, one of Smith and Mosier's desirable attributes.

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

$$\{\textit{function name}\} \{\textit{pattern parameters}\} = \{\textit{function body}\}$$

where

$$\{\textit{local functions}\}$$

Figure 4.1 shows the organization of the visual display area, which is split into five panes. The main properties of this design are: a similar mechanism is used to select and edit any item (consistency and flexibility of control); a compact module view as well as an elaborate function description (efficient information assimilation); function parameters, function body and local functions are displayed in separate panes (efficient information assimilation); tooltip text to show types, menus (minimal memory load); only one function description is displayable at a time (efficient information assimilation); every displayed item is editable (compatibility of display with entry); each pane is resizable and scrollable (flexibility of control).

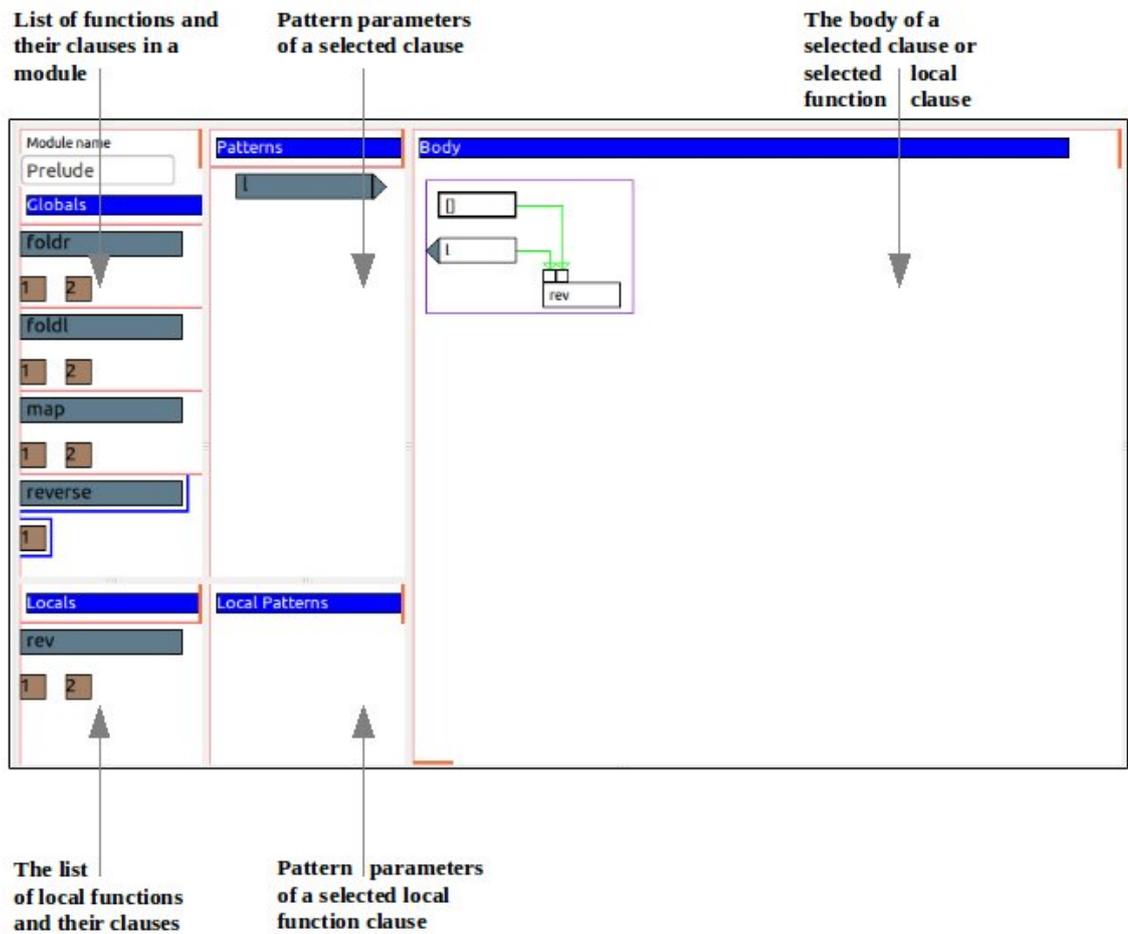


Figure 4.1: The visual display area.

4.1.2 Content

All items in this system are displayed as annotated icons (another example of data consistency). Different icons are designed by applying semiotics as a guide to five levels of icon design, where the first four levels were guided by Marcus (Marcus, 1992) and the fifth one was guided by Shneiderman (Shneiderman and Plaisant, 2004):

1. *Lexical* Machine-generated marks — colour, brightness
2. *Pragmatics* Identifiable, memorable, overall legibility
3. *Syntactics* Appearance and movement — modular parts, patterns, shape
4. *Semantics* Items 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

Figure 4.2 shows some icons used in this system.

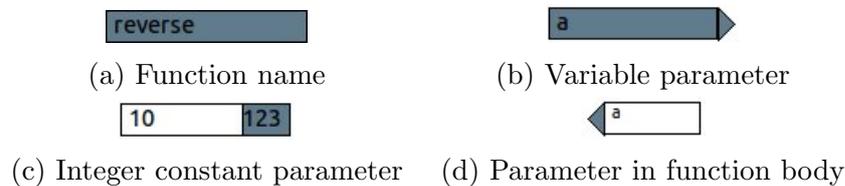


Figure 4.2: Pictures of some data fields.

Applying lexical semiotics, items on the left side of a function definition, which can be used in a function body, use a grey-blue rectangle (see Figure 4.2a and 4.2b) (for example, a function name or a parameter variable). All the

other items use a white rectangle (see Figure 4.2d and 4.4h) (for example, all items in a function body or a wild card parameter in a pattern). Applying pragmatic semiotics, parameter variables have a triangle on their right-hand side (see Figure 4.2b), to suggest to the user that they will be immediately used, and so that they can be spotted easily among other parameters (e.g., wild cards (see Figure 4.4h), constants (see Figure 4.2c)). Similarly, if an item used in the function body is a parameter, then it uses a triangle to its left (see Figure 4.2d). As syntactic semiotics, any item annotated with 123 on its right side, denotes an integer constant (see Figure 4.2c). In the same manner, abc denotes a string constant (see Figure 4.4c), $'c'$ denotes a character constant (see Figure 4.4b), T/F denotes a boolean constant (see Figure 4.4d). The visual programming system also facilitates the creation of local functions. Only one level of local definition is allowed. In informal observation (four libraries were checked, not including the prelude (which is a standard library accessible by all Haskell programs) — `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. End-users wish for simplicity, and so the initial system provides only simpler methods (Nardi and Miller, 1990; Nielsen, 1992, 1993).

Four standard colours and markings are used in this system to attract attention (Wickens and Hollands, 1999). They are: a blue rectangle to indicate focus on a selected item (an example of dynamic semiotics), a purple rectan-

gle to indicate focus on a group of items in a scope of an expression (another example of dynamic semiotics), green lines to show dataflow and magenta to denote any unused argument slot (an example of semantic semiotics).

Function Names and Clauses

Function clauses are numbered in order underneath the function name (another example of syntactic semiotics). This avoids repetition of function names. Figure 4.3 shows a compact view of a small part of the `Prelude` module. This module includes four functions `foldr`, `foldl`, `map` and `reverse`. The first three functions have two clauses each, and the last (`reverse`) has just one clause.

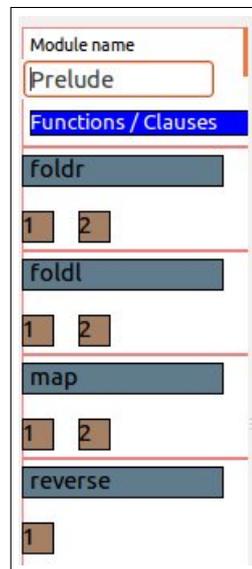


Figure 4.3: Compact view of a module.

Pattern Parameters

Eight kinds of pattern parameters are used in this design: constants (integers, characters, strings and booleans), variables, lists (empty and non-empty) and wild cards. This small set of parameters is sufficient to define simple functions such as those listed in this section. Figure 4.4 shows the icons chosen for these parameters. These icons are designed to be self-explanatory, and another example of using syntactic semiotics.

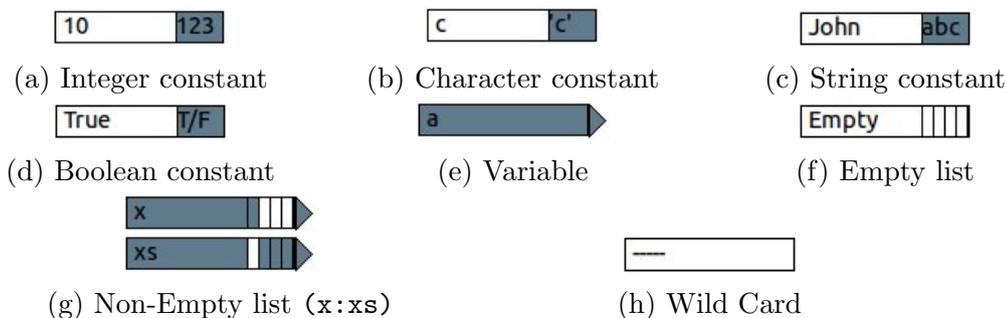


Figure 4.4: Parameter icons.

Function Bodies

Nine kinds of expression are used: constants (integers, characters, strings and booleans), function applications, operators, lists (empty and non-empty) and a conditional. The icons used for the constants in the function bodies are the same as those used for constant parameters. Figure 4.5 shows the non-constant expression icons. This small set of expressions is sufficient to define simple functions, as well as those listed in Section 4.1.2 (although perhaps clumsily). A function application icon contains argument slots on its upper left side as little boxes. An operator has two argument slots and a $+/-$ symbol

sets it apart from a function application (this symbol is a general indication of any operator). A rectangle with a bold outline denotes one of the two list constructors, `[]` or `:`. In functional programming, the primary control construct is a function (Burstall, 2000), and hence in HASKEU the `if-then-else` syntax is replaced by a predefined function, `cond :: Bool -> a -> a -> a`.

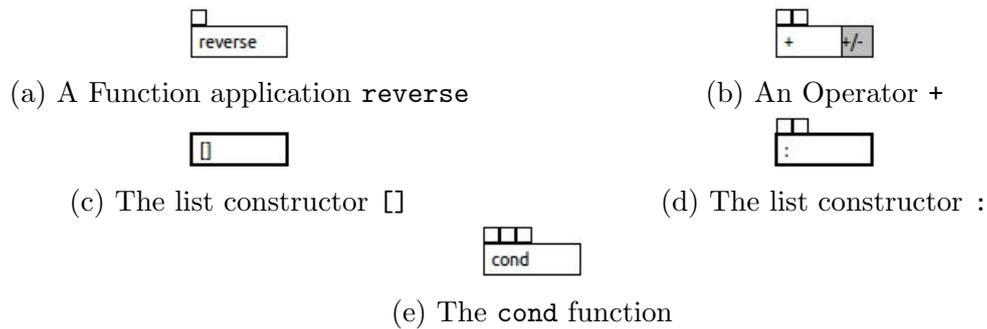


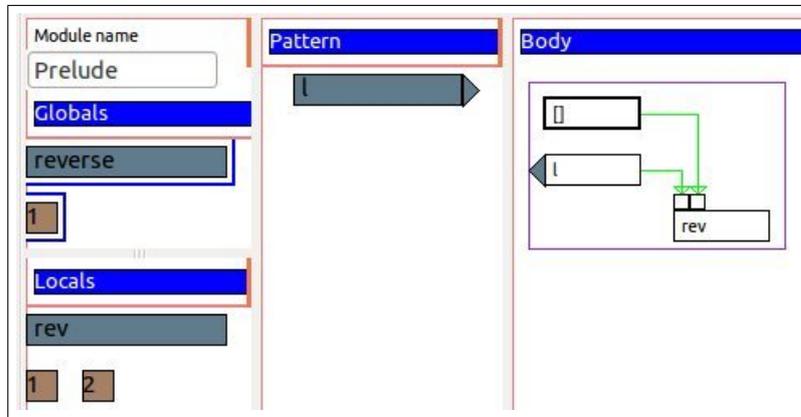
Figure 4.5: Expression icons.

Local Functions

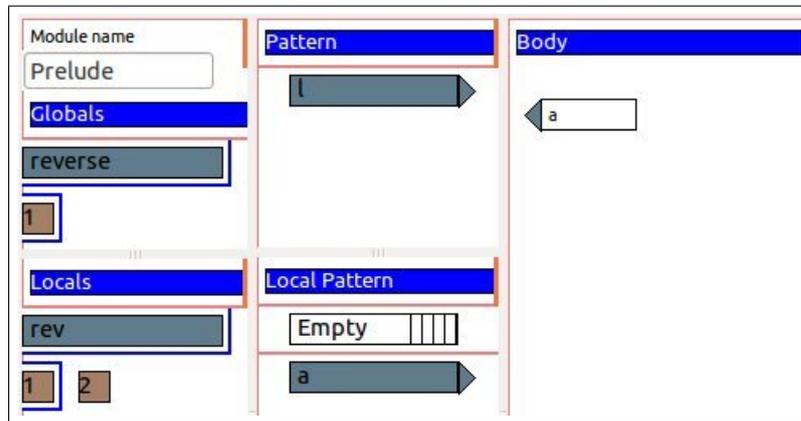
Local functions and their patterns are displayed in the same way as global functions and their patterns, but in separate areas. While working on a local function, a user can see the parameters of the global function along with the relevant local parameters (see Figure 4.6c). This is another way in which minimal memory load is achieved. Figure 4.6 shows the display of the `Prelude reverse` function, the textual equivalent of which is given below:

```
reverse l = rev l []

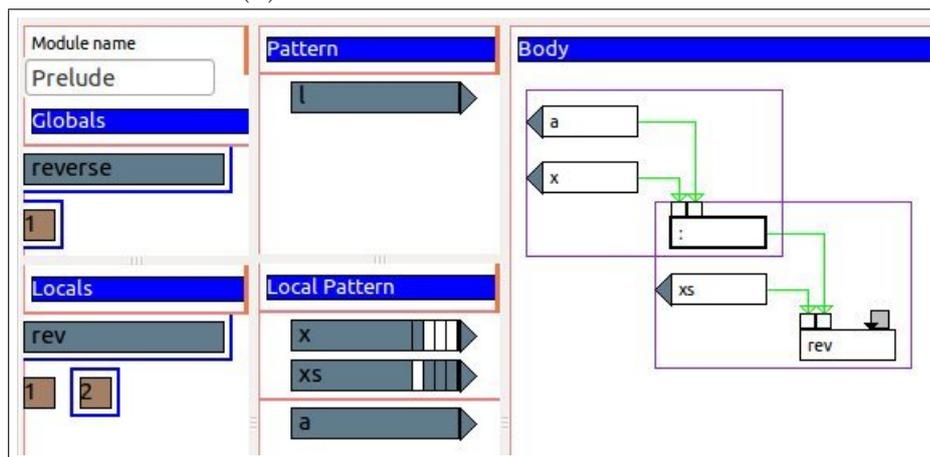
where   rev []      a = a
        rev (x:xs) a = rev xs ((:) x a)
```



(a) Function `reverse`



(b) First clause of local function `rev`



(c) Second clause of local function `rev`

Figure 4.6: Data display of `reverse`.

4.1.3 Dataflow and Scope

A function body is represented by a dataflow graph. An automatic layout algorithm is used to draw the dataflow graph based on the *grid standard* (Batini, 1986; Tamassia et al., 1988). The dataflow graph is embedded in a rectangular grid so that item boxes are placed in grid cells, and the edges follow horizontal and vertical tracks. Figure 4.7 shows the body of `max` function, where the flow of data, and scope of operators and applications can be clearly seen.

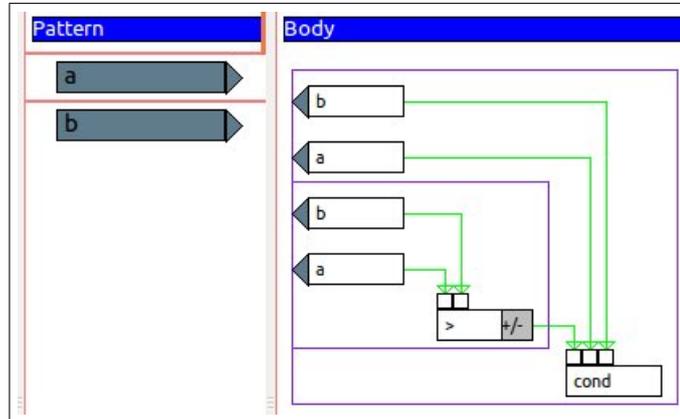


Figure 4.7: Showing dataflow of function `max`.

In HASKEU, the dataflow direction is left-to-right and top-to-bottom (see Figure 4.7). The reason for choosing left-to-right is that it is the same direction in which data flows in textual programs. The reason for choosing top-to-bottom is that it avoids edges crossing. Hence, argument slots are drawn at the top-left corner of an item and aligned horizontally, and the value edge comes out of the right-hand side of an item. The result of a function is that of the bottom-right-most item. The dataflow graph is redrawn after each editing action. This is another way in which data consistency is achieved.

Any item in a function body is represented as a box with/without argument slots and/or value edge. A block-based architecture is used to represent the scope of expressions (Bernini and Mosconi, 1994). A block consists of a function application or operator, with its arguments. Two items are linked by a directed edge (Trudeau, 1993).

Connector symbols are used to reduce the number of flow lines (Nassi and Shneiderman, 1973). For example, an arrowed arc is used on the right-top of a recursive application (see Figure 4.6c, the `rev` application in function body). To prevent the dataflow graph from having crossing lines, the same argument is included more than once in the the dataflow graph. A triangle symbol on the left-hand edge of these repeated items remind the end-users that they are parameters.

4.1.4 Direct-Manipulation

Program editing uses direct-manipulation. This can lower the barrier to learning the syntax of a new programming language by constraining syntax and providing concrete visual representations on which to operate (Shneiderman and Plaisant, 2004; Fekete and Beaudouin-Lafon, 1996; Hundhausen et al., 2006; Read, 1996). End-users of this system may not be aware of the syntax and semantics of the programming language (Minor, 1991). Direct manipu-

lation techniques are used in the five panes as follows:

Adding a New Item

The mouse cursor changes in the relevant panes to indicate the mode of operation. Figure 4.8 shows how a new function is added. Figure 4.8a shows a list of existing functions. When the user positions the mouse cursor over an existing function, a horizontal double line appears to indicate the position of the new function (see Figure 4.8b). When the user clicks on the mouse, a placeholder is inserted into the list as shown in Figure 4.8c. There is similar procedure for creating a new clause by showing vertical double lines.



(a) List of functions (b) Selecting new position (c) New function inserted

Figure 4.8: Adding a new function.

Adding a new parameter follows the same procedure as adding a new function. Figure 4.9 shows the steps of adding a new parameter variable, **a**, after

an existing parameter variable, `f`.

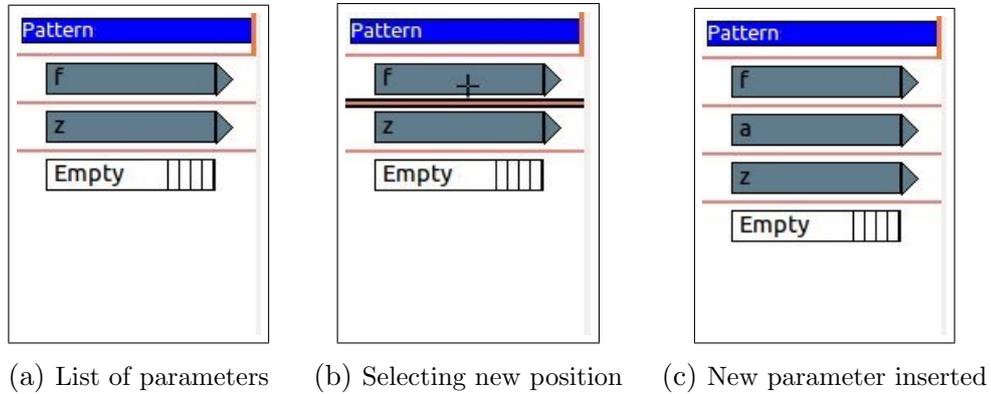
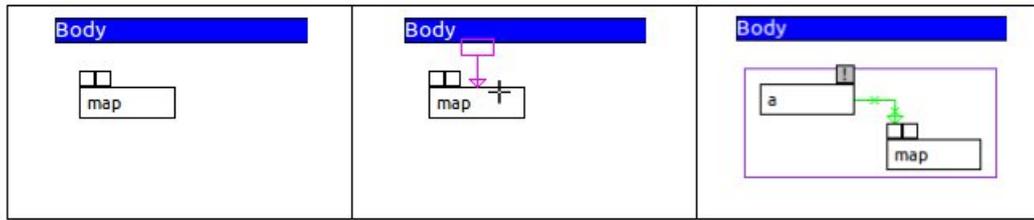


Figure 4.9: Adding a new parameter.

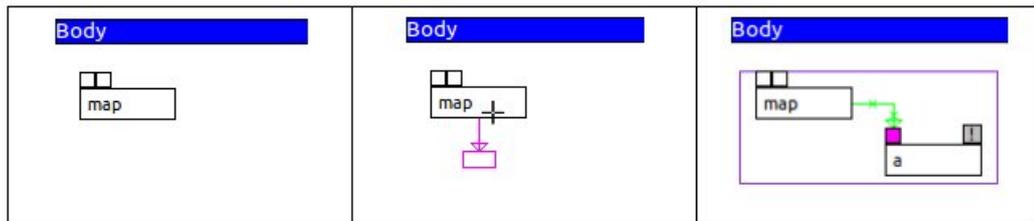
Figure 4.10 illustrates the procedure for adding an argument. A new argument can be added to an existing item and also an existing item can also be added as an argument to a new item. If the mouse cursor is positioned at the upper part of an item (in this example, `map`), then a symbol indicating “add argument” appears (see Figure 4.10a), and if it is positioned at the lower part of an item, then a symbol indicating “add as argument” appears (see Figure 4.10b). No symbol appears in the illegal case of applying a constant to an argument.

Selecting/Editing an Item

Item views change to indicate selection or editing. A blue outlined rectangle indicates selection and an annotation indicates an editing. Figure 4.11 shows the different item views as different keys are pressed.



(a)



(b)

Figure 4.10: Adding an argument.

		'f' key pressed; showing pattern variable shape;
		'o' key pressed; showing unobtrusive "!" symbol for being undefined function;
		'l' key pressed; showing unobtrusive "!" symbol for being undefined function;
		'd' key pressed; showing unobtrusive "!" symbol for being undefined function;
		'r' key pressed; showing input slots for <i>foldr</i> function;
		backspace key pressed; showing unobtrusive "!" symbol for being undefined function;
		'l' key pressed; showing input slots for <i>foldl</i> function and a recursion symbol;
Showing a pattern of <i>foldl</i> function	Editing an item in the function body	

Figure 4.11: Adding an annotation.

4.1.5 One Function Per Page

Typically, a user can focus well on only one task at a time (Medina, 2010; Crenshaw, 2008). Each time a person switches tasks, the brain runs through some processes to disconnect the neurons committed to one task and then connects them to the other task, and hence task switching creates delays and is inclined to produce errors. When a user wants to perform many tasks at one time, or to switch rapidly between them, it results in a very high rate of errors and it takes much longer (sometimes more than twice as long) to complete the tasks than if they were done alone sequentially (Wallis, 2006). This is because the human brain is compelled to restart and refocus. This system does not allow users to view two function bodies at a time in the same window. They need to select the function from the globals or locals pane to view its definition or to work with it. So, the user is not overwhelmed by the amount of work that needs to be done to complete a module or a single function and hence the perceived workload is also reduced (Carrier et al., 2009).

4.2 Exploratory Programming

This section aims to make the design of HASKEU suitable for the end-users who would like to explore the HASKEU system.

4.2.1 Error Reporting

Error reporting is an example of “offer simple error handling” in the 8 golden rules. Giving good error reports to novice users is very important (Norman, 1989; Shneiderman, 1986; Shneiderman and Plaisant, 2004). Since errors occur because of lack of knowledge, incorrect understanding, or inadvertent slips, users are likely to be confused, to feel inadequate or to be anxious. Error reports with an imperious tone that condemn users can heighten anxiety, making it more difficult to correct the error and increasing the chances of further errors. Messages that are too generic or obscure offer little assistance to end-users. Error reports should be understandable and state to the intended user what the problem is and how to solve it (Lewis and Norman, 1995). Best practices have been identified to produce better system with suitable error messages.

While the user is developing a program, HASKEU continually checks for errors and provides feedback. Unnecessarily hostile messages using violent terminology can disturb non-technical users (Shneiderman and Plaisant, 2004). HASKEU does not make use of violent language or colour to indicate errors, and hence the end-user will not be discouraged.

No syntactic errors are possible in their visual system — one cannot draw a broken dataflow graph or give it any invalid annotation, Semantic errors (type errors) are likely to happen all the time while end-users are developing a program. End-user will find it very difficult to understand the details of type

error messages, when it is hard even for experienced programmers to understand them (Chitil, 2001). Type error messages can be unsatisfactory if the meaning of these type error message, the meaning of the reported types and their relation to the program, is not well-defined. Furthermore, if the program position given in an error message is far from the source of the error, it makes it very hard for the end-users to locate the source of the error. The visual type error reporting in HASKEU is designed to aid end-users to understand and locate an error more precisely. HASKEU still allows users to see the type error details textually in an extended window (just below the visual data display area in Figure 4.12 outlined in green) in order to show the differences between visual and textual error reporting and also to allow the end-users prefer to work separately in the textual programming environment, where a visual report is not available, if they prefer.

Type Representation

The user's attention span was considered when designing the visual representation of the type system to provide good usability (Eberts, 1994). In order to give the user a minimal mental workload when starting with a function application or an operator, they initially see an annotated box with a number of argument slots. Then, when the user wants to know about the argument or result type of an application or operator, perhaps after a complaint was made by the system about a type error, the visual interface guides the user using concise tooltips. The type information of an individual argument is shown with

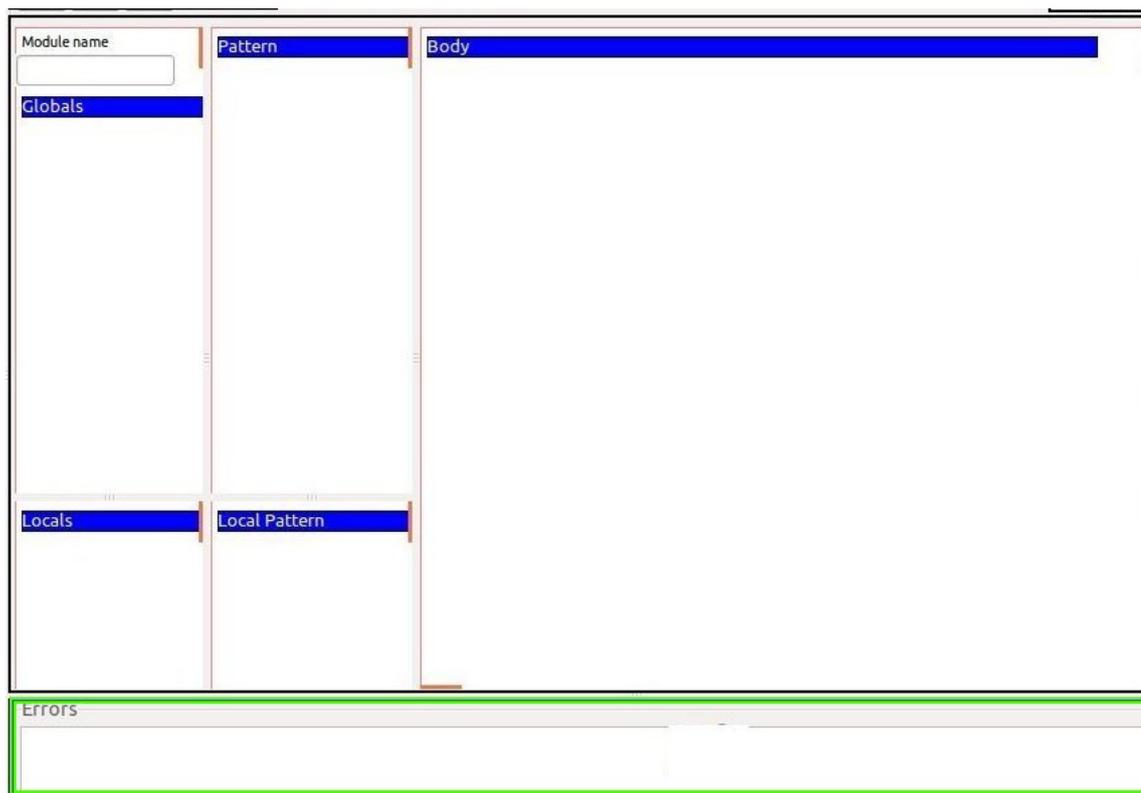


Figure 4.12: Extended window to show error textually.

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 4.13). During the program editing phase, the availability of type information for an application and its individual arguments and the visual dataflow display of applied and unapplied arguments can improve the user's understandibility about partial applications and higher order functions.

Visual vs Textual Error Reporting

An overview (a list) of all errors can be seen visually in the globals pane. Any function name and/or clause number with a cross mark indicates the existence



Figure 4.13: Type representation in `map` application.

of errors in it. Figure 4.14 denotes that the clause number “1” in function `foldl` and the clause number “2” in function `map` contain errors. Using this information, a user can track and trace all the type errors in the order of their existence in the code hierarchy.

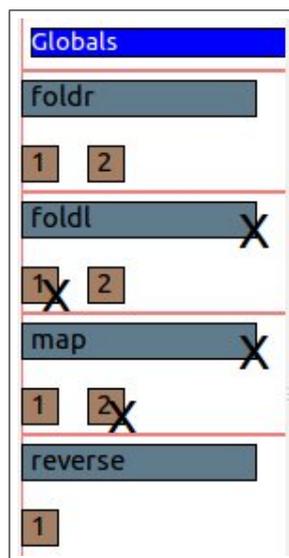


Figure 4.14: Overview of errors.

A user can see the detail of errors visually in the individual function body.

Consider the following small Haskell program:

```
f a = map 'c'
```

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 'c'

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

The visual error report in HASKEU can express the above error details in a single view (see Figure 4.15). Cross marks in the dataflow arc in function body indicate type errors and both end-points contain the type information as tooltip text.

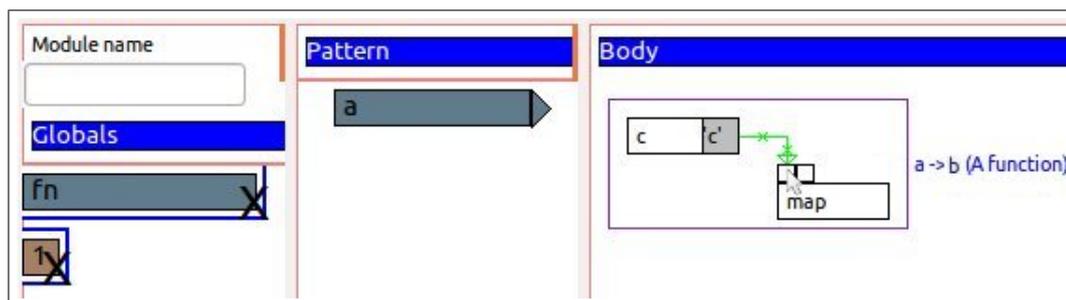


Figure 4.15: Error - type mismatch.

The function body shows all the type errors in the dataflow graph, not just the first one. Another small Haskell program is given below as an example:

```
fn a = map (map b)
```

As `b` is undefined, applying `map` to `b` is incorrect, and hence `(map b)` cannot produce anything, applying another `map` to this `(map b)` is also incorrect. The Glasgow Haskell compiler shows only the first error message, whereas the visual error reporting in this system shows both errors in the dataflow graph (see

Figure 4.16). In functional programming, a previous error may be the cause of some later errors and the ability to show them all in one view may improve the end-user's understanding of the functional program.

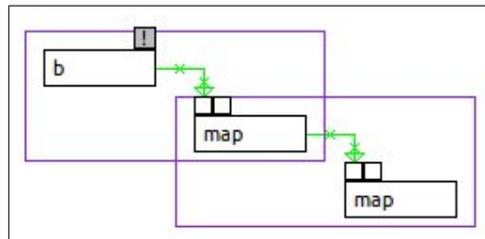


Figure 4.16: Showing all errors.

A type mismatch can happen during unification. Unification of two types means that they are assumed to be of the same type. In the case of type mismatch during unification, it is hard for the type checker to tell which wrong parameter makes the other parameters wrong, only the user will know. The following program highlights part of the problem:

```
fn a = map a a
```

The Glasgow Haskell compiler gives the following error message:

TestProg.hs:1:14:

Couldn't match expected type '[a0]' with actual type 'a0 -> b0'

In the second argument of 'map', namely 'a'

In the expression: map a a

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

Here, `map` is using the same parameter `a` in its two arguments where one is correct and the other is not. From the visual view of this function, the user

can see all the uses of `a` and how many of them have been used incorrectly (see Figure 4.17).

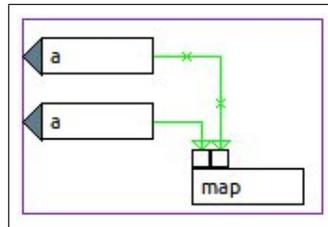


Figure 4.17: Error - unification.

This system allows users to work with undefined functions, because some users like to define those functions later, and because one could not define mutually recursive functions otherwise. Also while editing an application's annotation the annotated text can be turned into an undefined application many times. An unobtrusive “!” symbol is shown at the top-right corner of an undefined application and also a tooltip text “Function not defined” will be shown (see Figure 4.18).

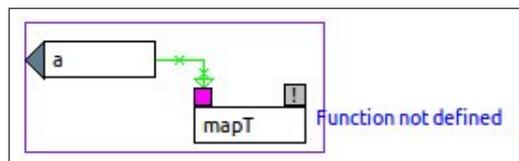


Figure 4.18: Error - undefined function.

Any unused argument slot is shown in the colour magenta (see Figure 4.19), so that the user will know an unnecessary argument has been used.

The The Glasgow Haskell compiler says after compilation:

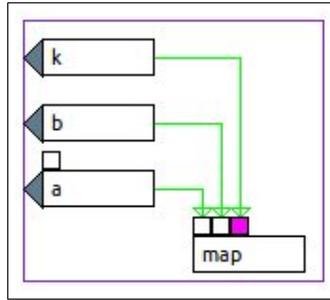


Figure 4.19: Error - unused argument.

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

Whereas this visual system in HASKEU shows a magenta argument slot,
as soon as any unnecessary argument has been used during program editing.

4.2.2 Infinite Undo

Infinite undo is an example of “permit easy reversal of actions” in the 8 golden rules. While programming, the user may frequently find that a change needs to be undone. The undo ability of any programming environment helps a user to be more adventurous while learning a new system. The ability to undo a long sequence of operations lets the user feel that the interface encourages exploration. While learning the interface, users can experiment with it, confident

that they aren't making irrevocable changes – even if they do something accidentally. This is true for users of all levels of skill, not just beginners. Undo is thus a primary tool for supporting exploration in software user interfaces. It allows the user to reverse one or more previous actions if they decide to change their mind. The significant benefits of undo are: It saves time/keystrokes and it reassures the user (Cooper et al., 2007).

In HASKEU, undo is a global, program-wide function, which can undo actions made by direct manipulation, by clicking a button or through a dialog box both in the textual interface and the visual interface. Hence, the undo button stays in the global toolbar with other global tools such as open, save, exit etc. Multiple undo functionality is used which can reverse more than one previous operation, in reverse temporal order. The HASKEU maintains a stack of operations, the depth of which is infinite, to allow users complete flexibility in the way they program. Some will use the undo facilities more frequently than others. HASKEU has also a redo function which can prevent the situation created by a multiple undo where if the user has undone something desirable, they can restore the last good action using redo. Program elements such as undo and redo allow the user to manipulate pieces of information needed in multiple places and within a particular task and relieve short-term memory (Mandel, 1997).

Some operations which need to be undoable have been chosen which keeps

it consistent with other development systems. Any action that might change a file, or anything that could be permanent, should be undoable, while transient or view-related states often are not. Useless undos often irritate the user by cluttering up the undo stack. Specifically, the followings actions are set as undoable and non-undoable in HASKEU:

Undoable Operations

- Adding a new function or function clause
- Adding a new parameter
- Adding an argument
- Adding an annotation
- Deleting items

Non-Undoable Operations

- Selecting
- Navigating between functions or clauses
- Moving the mouse cursor and text cursor locations
- Changing the scrollbar position
- Changing the window position and size

The next section will go on to describe how conventional textual programming fits into the design of HASKEU.

4.3 Textual Programming

This integrated Visual-Textual programming system has been designed to help end-user programmers to develop functional programming skills. As visual and textual program representations are both useful, integrating the two may give more strength in program development (Meyers, 1991; Scaffidi et al., 2012). The combination of a visual language and a textual language is intended to support the end-user in developing an understanding of functional programming concepts as well as the skills to use these concepts effectively. Any editing operations in the visual interface update their textual equivalent and vice-versa. Figure 4.20) shows textual changes reflected in the visual view after a function name changes (character `l` deleted), and Figure 4.21 shows visual changes reflected in the textual view after parameter `z` is deleted.



Figure 4.20: Changes propagate from textual to visual.

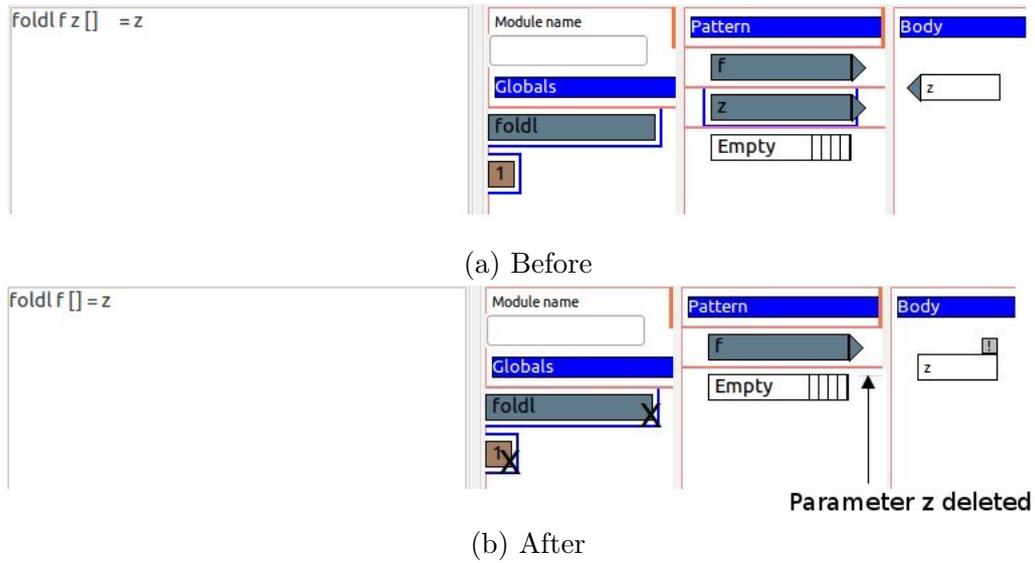


Figure 4.21: Changes propagate from visual to textual.

4.4 Design of Concepts

Design often involves diagrams, especially in the OOP paradigm (e.g. UML diagrams), whereas functional programming hardly uses diagrams to show the design of a program. The appropriateness of using UML diagrams for functional programs is discussed in Appendix A.

The overview of the design of the HASKEU implementation is described here very briefly by writing down function types to encapsulate the design relationship. The architecture of the HASKEU system is based on “The Model View Controller as a Functional Reactive Program” as given in Chapter 3. The global data structure of the the HASKEU system state is called `SystemState` which contains the textual state, visual state, source file information, type information and state changes history. The `SystemState` is also mentioned as a model in this thesis. HASKEU architecture has mainly two

function types — `BusinessRule` and `InterfaceLogic`. The business logic function is the prototype of the declarative functions that make changes in the model. It takes the model value `m` and the event value `e` as parameters and has the type

```
type BusinessRule e m
    = e -> m -> m
```

The interface logic is the prototype of the declarative functions that change the view of the system as system state change. It takes the model value `m` and the previous view value `v` as parameters and has the type

```
type InterfaceLogic v m
    = v -> m -> v
```

Below are the list of main business rule functions in the HASKEU system, a function name summarizes its purpose.

```
brTextEditorChanged      :: BusinessRule (String, Int) SystemState
brModuleNameChanged     :: BusinessRule (String, Int) SystemState
brFnNameChanged         :: BusinessRule (String, Int) SystemState
brLclFnNameChanged      :: BusinessRule (String, Int) SystemState
brFnPatNameChanged     :: BusinessRule (String, Int) SystemState
brLclFnPatNameChanged   :: BusinessRule (String, Int) SystemState
brExpNameChanged        :: BusinessRule (String, Int) SystemState
brListTxtErrSelected    :: BusinessRule Int SystemState
brMseLeftClickGblFn     :: BusinessRule EventMouse SystemState
brMseLeftClickFnArg     :: BusinessRule EventMouse SystemState
brMseLeftClickLclFnPat  :: BusinessRule EventMouse SystemState
brMseLeftClickExp       :: BusinessRule EventMouse SystemState
```

Below are the list of main interface logic functions in the HASKEU system.

```
ilSetWindowTitle           :: InterfaceLogic ViewType SystemState
ilSetTextEditorText       :: InterfaceLogic ViewType SystemState
ilSetSelectedTextualError :: InterfaceLogic ViewType SystemState
ilShowTextualErrors       :: InterfaceLogic ViewType SystemState
ilShowLineRowColStatus    :: InterfaceLogic ViewType SystemState
ilIsEndOfUndo             :: InterfaceLogic ViewType SystemState
ilIsEndOfRedo             :: InterfaceLogic ViewType SystemState
ilSetModuleName           :: InterfaceLogic ViewType SystemState
ilDrawFnView              :: InterfaceLogic ViewType SystemState
ilDrawLclFnView           :: InterfaceLogic ViewType SystemState
ilDrawFnArgView           :: InterfaceLogic ViewType SystemState
ilDrawLclFnPatView        :: InterfaceLogic ViewType SystemState
ilDrawFnBodyView          :: InterfaceLogic ViewType SystemState
ilShowMseCoordinateStatus :: InterfaceLogic ViewType SystemState
ilShowVisualErrors        :: InterfaceLogic ViewType SystemState
```

The following sections describe the concepts in the HASKEU system that were challenging to design and some novel solutions:

- (a) Synchronization between textual and visual view;
- (b) Tree structure in the dataflow;
- (c) Higher-order functions in visual view;

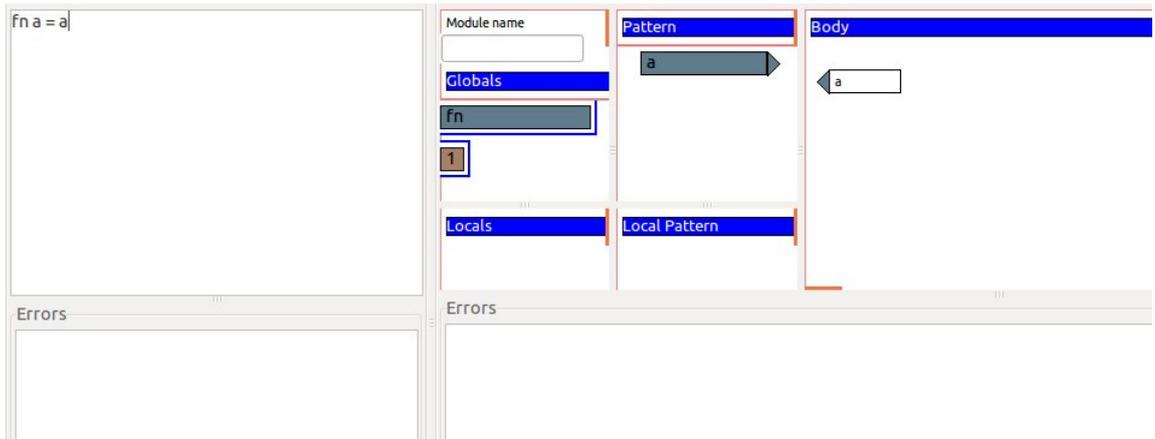
4.4.1 Synchronization between Textual and Visual view

Any editing operations in the visual interface update their textual equivalent and vice-versa. However, allowing syntax errors in the textual editor and

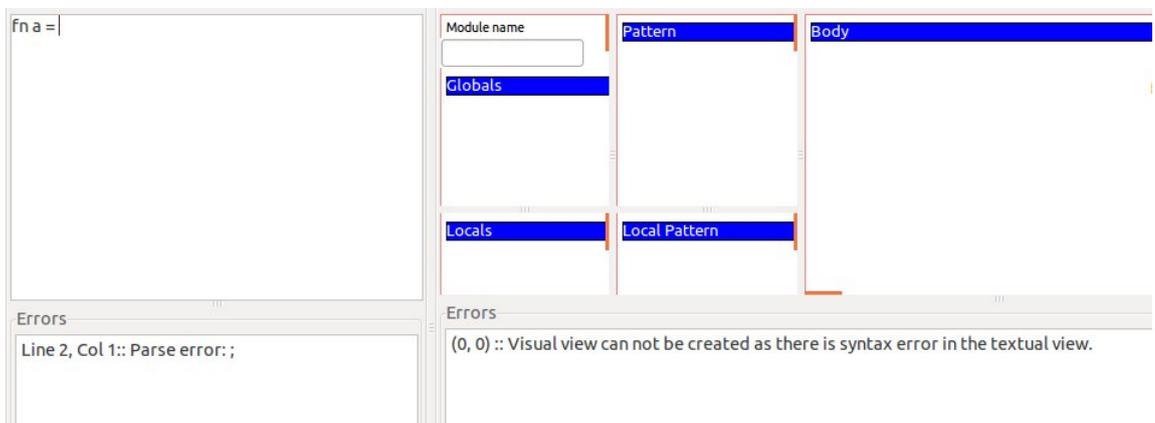
not allowing them in the visual editor forced a decision to be made about how the synchronization should work. It might be possible to have an error free syntax-directed textual editor, but the disadvantage of using a syntax-directed editor is that it prevents the user from making a radical change quickly (Bai, 2003). This was seen as a barrier to program development by end-users. Consequently a major effort was put into implementing a new visual system rather than devoting time and effort to implementing a syntax directed textual editor (research has already been conducted in this area). The following shows the adjustments in synchronization:

- If an editing in the textual editor contains a syntax error in the program, nothing will be shown in the visual editor. Though it is possible to keep the visual program at the same state as the last syntactically correct program, the visual view will not reflect the textual program and will be misleading to the user.
- The following solution was adopted: the syntax error will be shown in the textual error list, and a message will be shown in the visual error list that “The visual view can not be created as there is a syntax error in the textual view”.

Figure 4.22a shows the system view before a syntax error, and Figure 4.22b shows the system view after a syntax error was made by editing in the textual view.



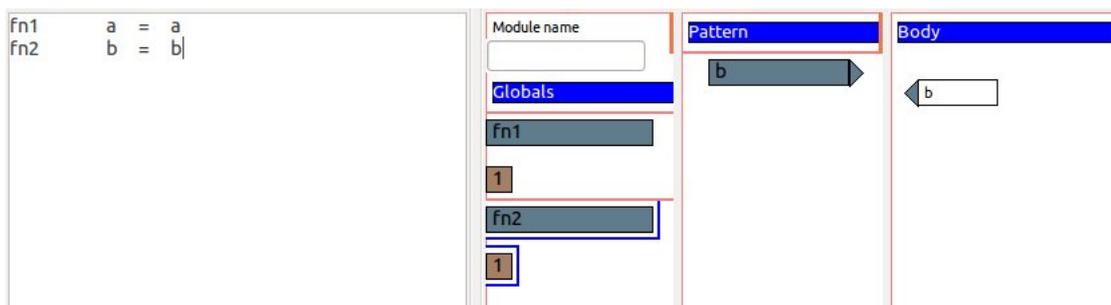
(a) Before



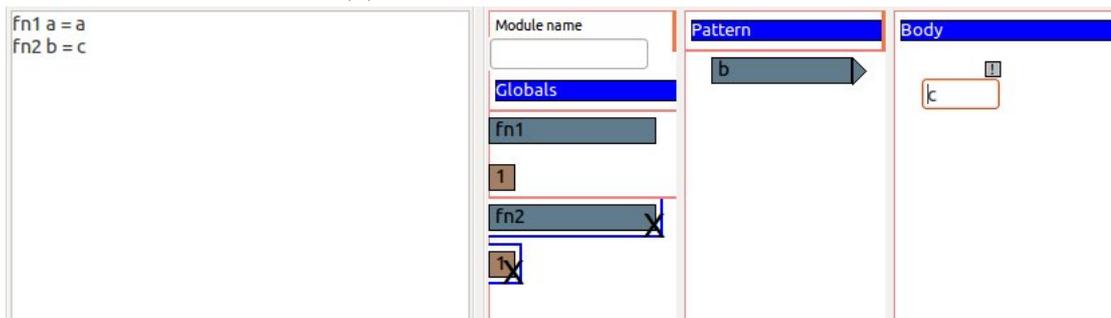
(b) After

Figure 4.22: System views before and after a syntax error.

As a syntax-directed textual editor is not being used, users can use their own layout while editing in the textual system, but while editing in the visual editor, a pretty-print is generated with a default layout for the textual view. It is always possible to have a textual version with the layout set by the user the last time he/she edited in the textual editor, but it is beyond the scope of this thesis to look at textual editing, the focus of this research is visual editing. Figure 4.23a shows the textual view when a user set his/her own layout by editing in the textual view, and Figure 4.23b shows the textual view generated by the system when the user edited it in the visual view.



(a) User defined textual layout



(b) System defined textual layout

Figure 4.23: User and system defined textual layout of a program.

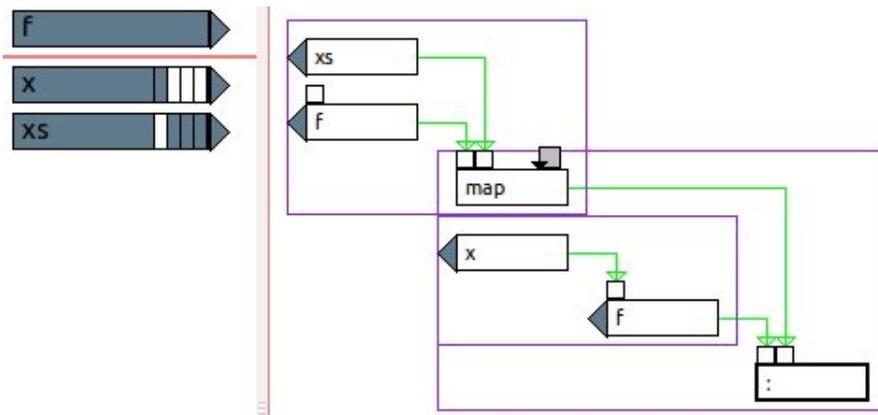
4.4.2 The choice of a tree structure

The dataflow in the visual representation of HASKEU uses a tree structure, i.e. a rooted graph, instead of a more general acyclic graph. A tree is a Directed Acyclic Graph (or a DAG) that does not contain cycles where a child can only have one parent (Christofides, 1975).

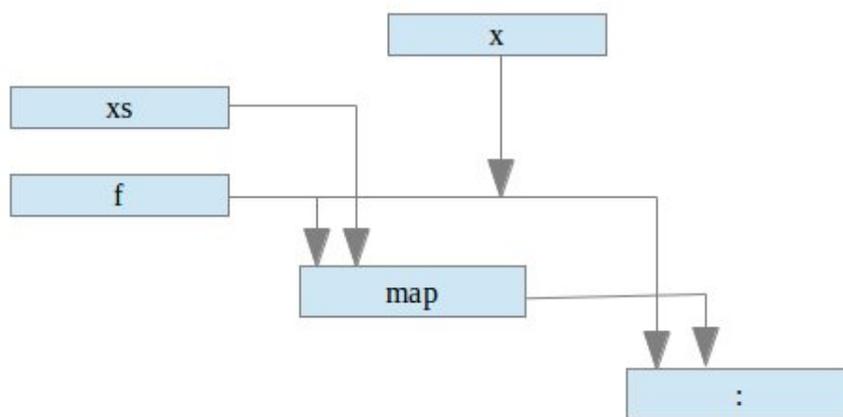
In HASKEU, it was chosen to use a tree structure with automatic layout and special symbols (see Section 4.1.3) to avoid edges crossing (so as to avoid spaghetti code). See Figure 4.25 below to see a comparison between a tree structure in HASKEU and a possible DAG of the second clause of the `map` function. In Figure 4.24b where a DAG has been used, the parent-child relationships are not obvious because of edges crossing.

4.4.3 Higher-order functions in visual view

HASKEU can express higher-order language features very effectively. Taking the simple function `fn f = f`, then from the function body in the visual view it can be seen that `f` has no argument slot (see Figure 4.25a). As soon as an argument is added to `f` as in `fn f a = f a`, it can be seen that an argument slot has been dynamically created (see Figure 4.25b). If the function is expressed as `fn f a = f (f a)`, then all the occurrences of `f` shown in the function body have one argument slot (see Figure 4.25c). Again, if another argument is added to the first `f` in the function body as in `fn f a b = f (f a b)`,



(a) map function as a tree structure in HASKEU



(b) map function as a DAG

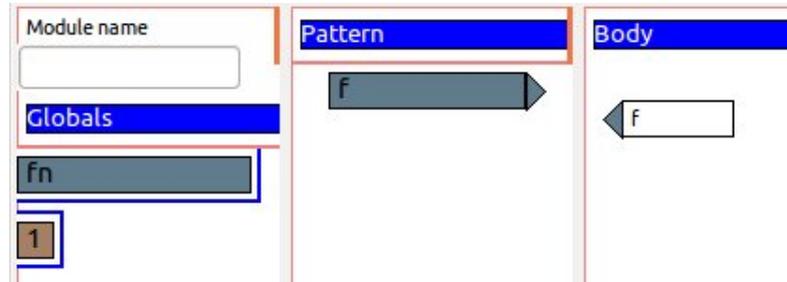
Figure 4.24: Tree vs DAG.

all the `f` occurrences in the function body have two argument slots now (see Figure 4.25d). The second `f` in the function body has an unused argument slot which clearly indicates that it has a function as an output (which also indicates that this second `f` is a partial application).

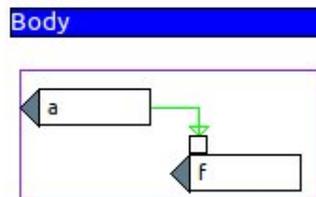
4.5 Conclusion

As a summary, the end-user programming system (HASKEU) designed for this research will support both visual and textual programming, allowing for a smooth transition from one to the other as programming expertise improves. The primary interface will allow end-users to write programs in a visual dataflow language consisting of boxes and arrows — a box representing a process and an arrow representing the dataflow between processes. Experiments have shown that novice users find this kind of dataflow language easier to understand (Kimura et al., 1986). The secondary interface will allow end-users to write programs in a conventional textual language, which is not entirely intuitive for novice end-users, but will become more meaningful as their expertise increases. Changes will propagate between the visual and textual interfaces, so that they are always consistent. The HASKEU system will not be task-specific; but instead it is intended to aid end-users to do general purpose programming.

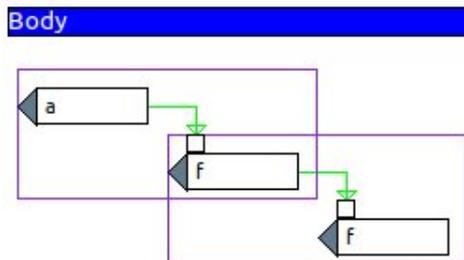
The visual programming system in HASKEU is purposely incomplete in that it does not aim to include all of Haskell's vast syntax, and only very simple Haskell functions can be expressed visually in this system. The visual



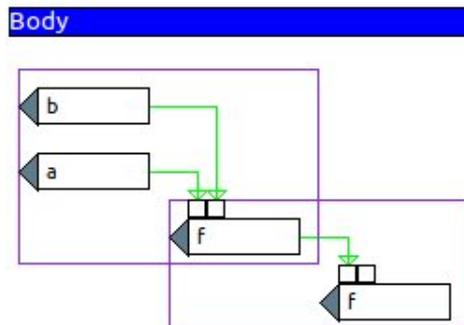
(a) f has no argument slot (in function $\text{fn } f = f$).



(b) f is a function and has one argument slot (in function $\text{fn } f \ a = f \ a$).



(c) All the occurrences of f have one argument slot (in function $\text{fn } f \ a = f \ (f \ a)$).



(d) All the occurrences of f have two argument slots and the second f is a partial application (in function $\text{fn } f \ a \ b = f \ (f \ a \ b)$).

Figure 4.25: Higher-order functions in HASKEU visual view.

system of HASKEU aims to give the user the feeling that functions are as simple as lambda calculus. If the users type a textual program into HASKEU that the visual system cannot handle, then an error message is shown in the visual view as “Some of the code in the textual system is incompatible with this version of the visual system”. In a future version of HASKEU, those currently unsupported syntax will be included and will be shown in as simple a form as possible by considering HCI issues. However, end-users should find the visual programming system in HASKEU a useful starting point upon the learning ladder of functional programming. Users are not asked to give type signatures of functions explicitly, which is also optional in the Haskell programming language. In the next chapter the implementation of this design will be shown.

Chapter 5

Implementation

The implementation of HASKEU follows the “MVC as a FRP” framework as described in Chapter 3. The HASKEU system was implemented in a pure functional paradigm which was a big challenge. It is natural to think of a programming development environment in terms of the state of the program under development and the events that the programmer creates to change the state. This challenge was overcome with the use of the “MVC as a FRP” framework. The challenges which were anticipated when implementing this system are outlined below:

1. Synchronizing the textual state (which can contain syntax errors) and the visual state (which is free from any syntax errors).
2. Drawing a visual layout of the syntax tree which involves recursive tree traversal.
3. Manipulating different data types of different nodes in the syntax tree using *higher-order types*

4. Representing type information and errors, as this involve type checking in different levels in the syntax tree.
5. Building an infinite redo/undo data structure which uses lazy evaluation.
6. Selecting an item, adding a new item or deleting an existing item in the syntax tree which also involves recursive tree traversal and checking the state of a node for its position in the visual layout, and also understanding the data structure of the syntax tree.
7. Validating syntax errors from different key presses, which involve value matching of different characters and value checking after a key is pressed.
8. Adjusting the existing GUI library for the “MVC as a FRP” framework.
9. Drawing shapes with low-level graphics.
10. Adjusting unexpected behaviours of some widgets taken from the existing GUI library.

The following external libraries have been chosen for this implementation, as it was discovered that they fulfill the purpose by experimenting with them and by reviewing literature on the web:

1. *reactive-banana* : a practical library for functional reactive programming (Apfelmus, 2015a).
2. *wxHaskell* : wxHaskell is a portable and native GUI library for Haskell (Leijen, 2014).

3. *reactive-banana-wx* : provides some GUI examples for the reactive-banana library, using *wxHaskell* (Apfelmus, 2015b).
4. *haskell-src-exts* : a suite of annotable datatypes describing the abstract syntax of Haskell (Broberg, 2014).
5. *haskell-type-exts* : a type checker for Haskell as embodied syntactically by the *haskell-src-exts* (Broberg, 2012a).

Figure 5.1 shows a general outline of the MVC layout (the connections between the model, the controllers and the views) in this system. The arrows in the figure indicate that controllers are changing the model and the views are generated from the model. The lines indicate which part of the model the controllers are changing and from which part of the model the views are displayed. The main two views and controllers manage the textual and visual state of the system. Other views and controllers have the following operations - to open and save files, to manage type information, and to manage the model change history to redo and undo any operation. The details of this system will be discussed in the following order: the model, the controllers and the views.

5.1 The Model

The model represents the core information that the system is being used to access and manipulate. The model is the centre of the system, the views/controllers (visual, textual, file information, type information, and history of model changes)

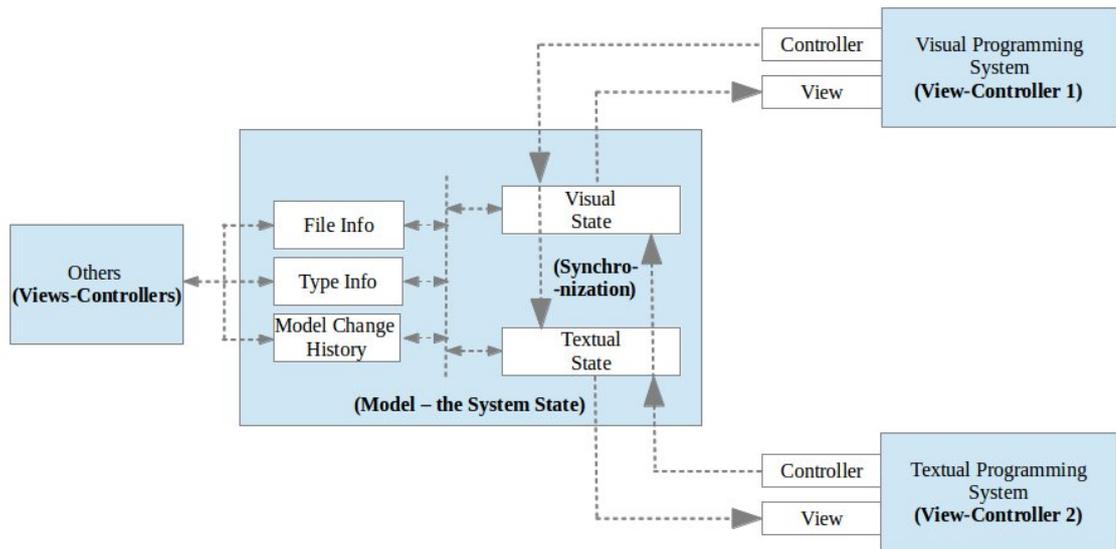


Figure 5.1: General outline of the MVC layout in the system.

manipulate and visualize the model. The controller interprets event inputs and updates the model state. The views reflect the state of the model as appropriate.

The model contains information about the the state of the system. The high-level definition of the system state (model) data type is as follows:

```

data SystemState =
  SystemState
  {
    textualState      ::TextualState,
    visualState       ::VisualState,
    fileInfo          ::FileInfo,
    typeInfo          ::TypeInfo,
    modelChangeHistory ::ModelChangeHistories
  }
  deriving (Show)

```

In this implementation, a textual program, that does not parse, has no syntax tree. So, the textual state is the program as a list of characters. On the other hand, syntactic errors are not possible in the visual system because of validation. So, the visual state of the program is in the form of a shape tree (a syntax tree in which items are annotated with visual information). Any visual item is a node in the syntax tree, so nodes will also be referred as items in this chapter. The `TextualState` and `VisualState` also hold the state of the editors (widgets), because the widget's views are always changed with a model change, hence the original value of the widget is lost. For example, while editing in the textual editor, the model is changed with an updated value, and as soon as the model is changed the textual editor view is also changed. At this time the textual editor's state of the insertion point goes to the end of the list of characters. So, it is necessary to keep track of the insertion point too. `FileInfo` contains information about the source file as provided by the user of the system. `TypeInfo` contains information about the type of all the defined and undefined functions in the current module and also information about the type of the functions in the other embedded modules for test purposes. `ModelChangeHistories` keeps the history of the changes to the model such as to implement undo and redo operations.

`TextualState` includes the textual program `textStr`, the detailed textual error messages and error location `textualErrors`, the index of selected error message by the user `selectedTxtErr`, and the cursor position in the text editor

insertionPt.

```
data TextualState =  
    TextualState  
    {  
        textStr      :: String,  
        textualErrors :: [ErrorMsg],  
        selectedTxtErr :: Int,  
        insertionPt  :: Int  
    }  
  
    deriving (Show)
```

The VisualState data type is given below:

```
data VisualState =  
    VisualState  
    {  
        shapeTree      :: HSE.Module (SrcSpanInfo, ItemState),  
        visualErrors   :: [ErrorMsg],  
        selectedVisualErr :: Int,  
        mouseCoordinate :: (Int,Int),  
        modeOfOperation :: ModeOfOperation,  
        areaOfOperation :: AreaOfOperation,  
        fnClButtons    :: [VirtualButton],  
        patButtons     :: [VirtualButton],  
        expButtons     :: [VirtualButton],
```

```

    scr1PosAnnEdtGbl      :: Point,
    scr1PosAnnEdtLcl     :: Point,
    scr1PosAnnEdtGblPat  :: Point,
    scr1PosAnnEdtLclPat  :: Point,
    scr1PosAnnEdtBody    :: Point
}

deriving (Show)

```

Here, `shapeTree` is the syntax tree where visual attributes of items are individually stored, and the Haskell syntax tree data type is used as defined in the `Haskell-Src-Exts` library (Sheard and Jones, 2002); the `ItemState` datatype contains visual attributes of an item and `SrcSpanInfo` is the textual correspondence to it; `visualErrorMsg` contains textual detail of visual error information; `selectedVisualErr` is the index of selected error message by the user; `mouseCoordinate` contains the coordinates of the mouse cursor in a pane; `modeOfOperation` is either selection mode or edit mode or one of the add new item modes; `areaOfOperation` is where the mouse cursor is positioned among the five panes; some virtual buttons are used for adding new items in the visual program (`fnClButtons`, `patButtons`, `expButtons`), so that these occupy minimal space in the user interface; the scrolled positions on the different panes are needed to adjust the positions of the editable text controls in a `wxHaskell` panel, and these text controls are used to edit annotation of the items in the five panes.

The `ItemState` contains the following visual attributes of an item:

```
data ItemState =  
    ItemState  
    {  
        itemPosition      :: Point,  
        itemSize          :: Size,  
        mseInside         :: Bool,  
        selected          :: Bool,  
        annEditMode       :: Bool,  
        annInsertionPoint :: Int,  
        argSelected       :: Int,  
        outputSelected    :: Bool,  
        nodeType           :: (TyDefined, Maybe Sigma),  
        nodeTypeArgs      :: [ItemState],  
        typeErr           :: String,  
        synonym           :: String,  
        tooltipText       :: String,  
        joinToChildNodes  :: JoinToChildNodes,  
        groupBox          :: (Point, Size)  
    }  
  
    deriving (Show)
```

Here, `itemPosition` is the position of an item on the screen, and `itemSize` is the size of an item; `mseInside` is set to `True` or `False` when the mouse cursor

enters or leaves the edge of an item; the `selected` and `annEditMode` denote an item's selected and editable modes respectively; `argSelected` is the argument number of an item when the mouse cursor focuses on an argument slot, and it is used to show the type information for a specific argument and also to add an argument to an item; `outputSelected` is used to add an existing item as an argument to a new item; `nodeType` is the type information of an individual item and `nodeTypeArgs` contains type information and visual attributes of an individual argument slot; `typeErr` contains the type error message if an item is applied incorrectly; `synonym` is used to show the clause number of a function; `toolTipText` is the tooltip text of an item to show some error description or type information; `joinToChildNodes` is the list of dataflow arcs to connect the parent with the child items; `groupBox` is a rectangular box to focus items in a scope of expression as a group.

5.2 The Controllers

The definition of the system state (see Figure 5.1) shows that the textual and visual state of a program are two different parts in the model — a list of characters and a syntax tree. This is because a textual program cannot create a syntax tree if it cannot be parsed. So, to keep both states consistent, an active synchronization is needed between all the controllers of the visual and textual systems, which means that a textual controller must update both the textual and visual state, and similarly for a visual controller update. Any change in the programming system also updates the model history stack to implement

a redo/undo operation, so all the controllers that change a program need a default history update operation.

According to the MVC as a FRP framework (as described in Chapter 3, each controller has to have a business rule. To keep both states (visual and textual) synchronized, a class called `BRSync` has been created, where managing the history of the model changes and synchronization from textual to visual and vice-versa are embedded in the business rule functions. The coding for the `BRSync` is given below:

```
class BRSync e where

  br :: BRName -> BusinessRule e SystemState

  brSyncV :: BRName -> BusinessRule e SystemState

  brSyncT :: BRName -> BusinessRule e SystemState

  brSyncV b e m = manageHistory VisualChange $
                    convertFromVisual (br b e m)

  brSyncT b e m = manageHistory TextualChange $
                    convertFromTextual (br b e m)
```

Here, every controller implements a `br` function to have a business rule where `BRName` is the identifier of the business rule. The two default functions `brSyncV` and `brSyncT` are responsible for doing the actual synchronization and managing the history of the model changes. All the visual controllers are attached to the `brSyncV` function which updates the textual state from the visual state and all the textual controllers are attached to the `brSyncT` function which updates

the visual state from the textual state in the MVC system. This way, all the controllers stay compatible in synchronizing the two views and keep records of all the changes. If any third view-controller were needed to be added (e.g., braille for blind programmers), then new information would need to be added in the model data structure. It may be necessary to add another function to the class, which could have a name of `brSyncB`. This function will change the textual and visual states from a braille state. Also, the `brSyncV` and `brSyncT` functions need to be changed in order to change the braille state.

5.2.1 Visual Layout

The visual system uses an automatic layout to display a program. Any add, edit or delete operation in the program structure recalculates the item's visual attributes automatically. All the program items exist in a syntax tree. An ordinary module tree structure is defined in the `Haskell.Src.Exts` library as below:

```
data Module l
    = Module l          (Maybe (ModuleHead l))
                        [ModulePragma l] [ImportDecl l] [Decl l]
```

where `l` is the annotation type. By default it is set to textual annotation by the `Haskell.Src.Exts` library. The automatic layout algorithm calculates the position of items in the syntax tree, and assign them dimensions. `ModuleHead l` includes module name information. `[Decl l]` includes all the

function definitions. A single function is represented by the data constructor `FunBind l [Match l]` where `[Match l]` is the list of clauses. A single clause is represented by the data constructor as below:

```
Match l (Name l) [Pat l] (Rhs l) (Maybe (Binds l))
```

Where `[Pat l]` is the list of parameters, `Rhs l` contains the tree structure of the function body and `(Maybe (Binds l))` contains the local functions inside a `where` block. Among the guarded and unguarded `Rhs l`, only the unguarded one `UnGuardedRhs l (Exp l)` is implemented.

```
data Rhs l
    = UnGuardedRhs l (Exp l) -- unguarded right hand side (/exp/)
```

As different items in the syntax tree have different data types, *higher-order types* (Peyton Jones, 2002) are used in class `TreeManager` to generalize the functions which set the layout of items as below:

```
class (Functor t, Annotated t) => TreeManager t a b where
    setAutoLayout          :: b -> t a -> (b, t (a,b))
    setAutoLayoutPrec     :: b -> Bool -> (Int, Int) -> t a
                           -> ((Int, Int), t (a,b))
    mapAutoLayout         :: b -> [t a] -> (b, [t (a,b)])
    makeChildNodesJoins   :: [b] -> t (a,b) -> t (a,b)

    mapAutoLayout lp lstNode      =
        mapAccumL setAutoLayout lp lstNode
```

In this class declaration, `t` is the type variable of the data constructor of an item in the syntax tree, `a` is the textual annotation type and `b` is the visual

annotation type. The function `setAutoLayout` takes the visual attributes of the last item and the current item as arguments. Then, `setAutoLayout` calculates the position and sets other visual attributes of the current item and returns the current item's visual attributes and the item with both textual and visual annotation as a pair. The reason for returning the visual attributes of the current item separately is that we can use the Haskell function `mapAccumL` to calculate the visual attributes for a list of items (for example, list of function, list of local function, list of parameter) by using a common method. The function `mapAutoLayout` applies the auto layout operation to a list of items. Its default definition is given above.

The function `setAutoLayoutPrec` is used to set the layout of items of expressions in a precedence order (items in function body). The Haskell expression `data Exp l` is the *recursive tree-shaped* data type. Among all the `Exp` data constructors the followings have been used:

```
data Exp l
  = Var l (QName l)           -- variable
  | Con l (QName l)          -- data constructor
  | Lit l (Literal l)        -- literal constant
  | App l (Exp l) (Exp l)    -- ordinary application
  | List l [Exp l]           -- list expression
  | Paren l (Exp l)          -- parenthesised expression
```

In this implementation, an infix application expression has not been used, so all the operators are function applications. An infix application does not

show differently in the dataflow graph because this graph shows which operations must be performed before others in a clear way. Using the above definition of `Exp`, only the `App` data constructor can create branches and the other data constructors can be either a leaf or another node, the only child of which can be another sub-tree. The `App 1 (Exp 1) (Exp 1)` will be called `App Left Right` to describe the algorithm. From the above definition of `Exp 1`, it can be seen that the syntax tree is a binary one.

To give an example, the syntax tree structure and the dataflow graph in the system of the expression `map ((+) a) b` are shown in Figure 5.2. It uses a left-to-right and a top-to-bottom approach to show the dataflow of items in a grid layout. There is a two step process to map the syntax tree to a dataflow graph. These two steps involve two different recursive traversals of the tree — the first one is to find the row and column positions of an item in the grid layout and the second one is to find the children of an item in order to join all the children to their parent using dataflow arcs. The first one involves traversing all the children and grandchildren of an item, while the second one involves traversing only the direct children.

To Calculate the Position of a Node

The *reverse in-order traversal* is used to calculate the row and column number of items in the grid by using the `setAutoLayoutPrec` function recursively. The right-most leaf in the tree is set as the top-left-most (first) item in the

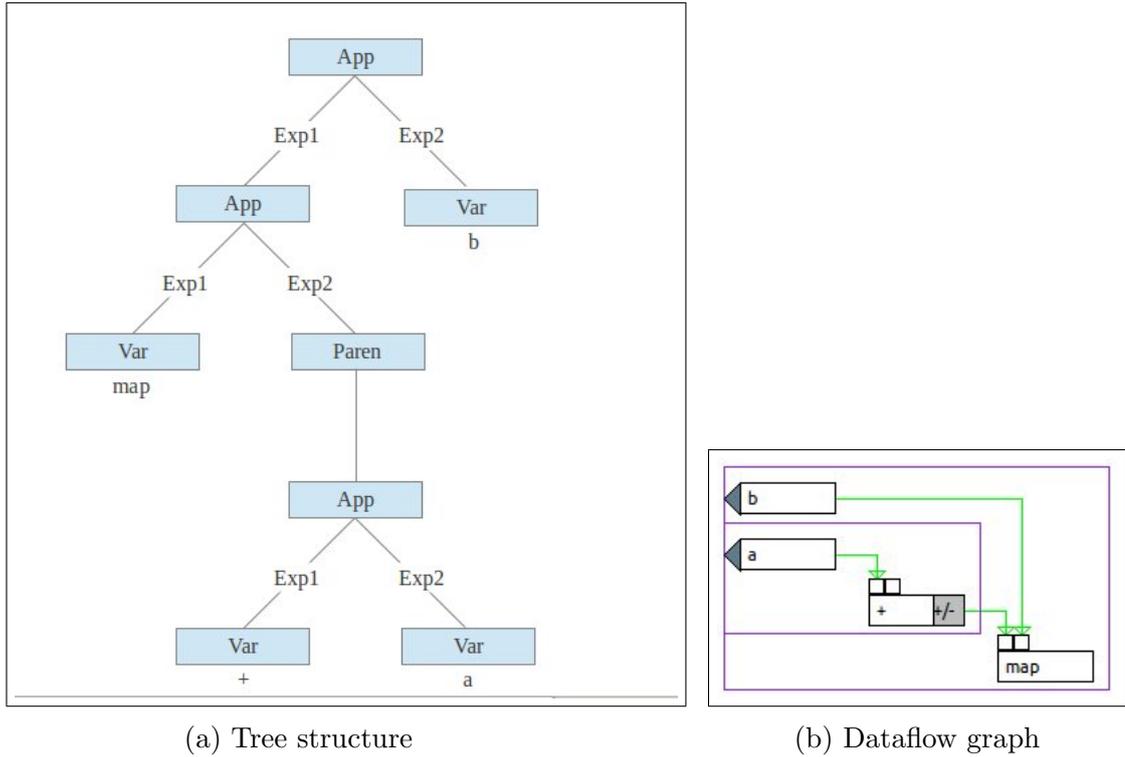


Figure 5.2: Tree structure and dataflow graph of expression map ((+) a) b.

dataflow graph, and the items in the tree are shown in reverse order in the dataflow graph. If a leaf item is found in the right sub-tree, the row position is increased by 1. If a leaf item is found in the left sub-tree, both the row and column positions are increased by 1. To sequence of operations between left and right sub-tree, the values of row and column positions returned by the operation on the right sub-tree are passed to the operation of the left sub-tree.

The algorithm is defined as a series of recursive operations at each node and starting with the row and column position of -1 and 0 respectively as follows:

1. Traverse the right sub-tree and increase the row by 1 for each leaf node.
2. Visit the root.

3. Traverse the left sub-tree and increase both the row and column by 1 for each leaf node.

When items are annotated with their positions the root of the tree or subtree is also annotated with the position of the left leaf or only leaf item. This makes the next step of finding the children quicker by reducing the time complexity of the recursion.

To Find the Children of a Node

The *pre-order traversal* is used to find the direct children of each leaf item in the tree by using the `makeChildNodesJoins` function recursively. While traversing the left sub-tree, the function adds the annotation of the right node (which is actually the item state of the left leaf or only leaf item in the subtree) to a list with each iteration until a leaf item found. While traversing the right sub-tree the list of children is reset to an empty list `[]` and then its left sub-tree is traversed. It does not matter whether the left or right subtree operation is performed first.

The algorithm is defined as a series of recursive operations at each node and starting with a empty list of children `[]` as follows:

1. Visit the root.
2. Traverse the left sub-tree and add the item state of the right nodes in a list.
3. Traverse the right sub-tree by setting the list of children as `[]`.

5.2.2 Type Management

The *haskell-type-exts* library has been used primarily to implement the type checking in the syntax tree (Broberg, 2012b). This library lacks the following functionalities to comply with this implementation. The purpose is not to build a type checker system, but to show the type of items and type errors in the system. So, the *haskell-type-exts* library is updated as given below to meet the requirements.

1. The *haskell-type-exts* does not support an annotated tree, so a function has been implemented to unannotate an item before giving it to the library for a type check.
2. If any undefined function is found in a module, this library stops type checking and throws out an error message. A function is implemented so that when any undefined function is found, it gives the function a type of `a`, and continues type checking the rest.
3. The type checker does not support a full list of parameters, so the parameters we needed have been added.
4. The type checker returns a list of functions with their type, but a syntax tree with the type result of each expression is needed. A `TypeManager` class (details given below) is implemented which returns a syntax tree with type check results of each expression.

The `TypeManager` class has the same type variables as tree manager, and it has an additional type variable to pass the parent node in the `setType` function. Take as an example of passing the parent node, to find the type of an expression in a function body, the parameter types and function name are needed, as the syntax tree does not tell us if an expression is a parameter or a recursive function.

```
class (TreeManager t a b) => TypeManager t1 t a b where
    setType          :: TypeInfo -> t1 (a, b) -> t (a,b) -> t (a,b)
    setType _ _ t    = t
    mapSetType       :: TypeInfo -> t1 (a, b) -> [t (a,b)] -> [t (a,b)]
    mapSetType ty t1 lstNode =
        map (setType ty t1) lstNode
```

The `setType` function traverses the syntax tree and finds the type of individual parameters and expressions, and also finds any type error. The `mapSetType` function applies the `setType` operation to a list of similar items, for example to a list of clauses.

The following function `unAnn` as given in the `UnAnnotation` class is frequently used to unannotate an item before giving it to the *haskell-type-exts* library for a type check, where `t1` is the annotated tree and `t2` is the unannotated tree.

```
class UnAnnotation t1 t2 where
    unAnn :: t1 a -> t2
```

The following function `getAllVarEnv` is used to find all the undefined functions in a program. When an undefined function is found, it is given a temporary type, and this operation is repeated recursively until all the undefined functions have been given types. The first parameter of this function which is a tuple, are the types and temporary types of functions in the embedded modules (which have been used for test purposes), the second parameter (also a tuple), are the types and temporary types in current module, and the third parameter is the unannotated syntax tree of the module.

```

getAllVarEnv ::      ([(QName, Sigma)], [(QName, Sigma)])
                  ->  ([(QName, Sigma)], [(QName, Sigma)])
                  ->  Module
                  ->  IO ([(QName, Sigma)], [(QName, Sigma)])

getAllVarEnv (embeddedVar, tempEmbedVar) (moduleVar, tempVar) mod =
    do
    let Tc tc = typecheckModule mod
        env <- mkTcEnv (embeddedVar ++ tempEmbedVar ++
                      moduleVar ++ tempVar) [] []
        tcRes <- tc env

    case tcRes of
        Left e ->
            getAllVarEnv (embeddedVar, tempEmbedVar)
                (moduleVar, (tempVar ++ (mkTempVar $ show e)))
                mod
        Right e -> return (moduleVar ++ e, tempVar)

```

The type of any item `TcType` is a recursive tree-shaped data type as defined in the *haskell-type-exts* library.

```

data TcType = TcForAll [TcTyVarBind] [TcAsst] Rho -- Forall type
    | TcTyFun TcType TcType -- Function type
    | TcTyCon QName -- Type constants
    | TcTyVar TyVar -- Always bound by a ForAll
    | TcTyApp TcType TcType -- Type application
    | MetaTv MetaTv -- A meta type variable

```

To get the individual argument type of an item, the following function is created which converts the recursive tree-shaped type into a list consisting of argument types and a return type. This way, the arguments of a function and unused arguments can be found.

```

mkTypeToList :: TcType -> [TcType] -> [TcType]
mkTypeToList (TcForAll a b r) lst = lst ++ (mkTypeToList r lst)
mkTypeToList a@(TcTyFun r1 r2) lst = lst ++ [(r1)]
                                         ++ (mkTypeToList r2 lst)
mkTypeToList a@(TcTyCon _) lst = lst ++ [(a)]
mkTypeToList a@(TcTyVar _) lst = lst ++ [(a)]
mkTypeToList a@(TcTyApp _ _) lst = lst ++ [(a)]
mkTypeToList a@(MetaTv _) lst = lst ++ [(a)]

```

To find the type error of an expression, the `setType` function checks an application for all of its arguments recursively. For example, in an expression `map a b`, it checks both `map a` (to check `a` is a correct argument here) and `map a b` (to check `b` is a correct argument here), and to find out which of the arguments are incorrect. If the type checker returns an error then the type error is set in the incorrect argument node.

5.2.3 Infinite Redo/Undo

The system allows infinite redo/undo. An infinite list is used to simulate the redo/undo operations, and this infinite list is commonly known as stream. Haskell's *lazy evaluation* makes it possible to implement this infinite redo/undo facility nicely in the system.

The infinite list should work like a first-in last-out stack to sequence the redo/undo operations. To make the infinite list work like a first-in last-out stack, a current position of the redo/undo stack is kept of which value is set to the position of the last item if a new system state is added, is decreased by one if a undo button is pressed and increased by one if an redo button is pressed. A system state is needed in a specific position when a redo/undo button is pressed.

In Section 5.2, it was shown that all redo/undoable events are mapped with the `manageHistory` function in the system. The `manageHistory` adds the current system state (which is actually the output of an event mapping function `BusinessRule`) to the redo/undo stack. With the use of lazy evaluation, the infinite list of system states are expressed in a recursive form, and when a specific system state from the list is needed, no further evaluation is necessary because that system state has already been evaluated by an redo/undoable event. Figure 5.3 shows the process between the redo/undo stack with the redo/undoable events and the redo/undo events. A redo/undo event is not a

redo/undoable event, which means any redo/undo button pressed doesn't add any system state to the stack because it would break the redo/undo sequence.

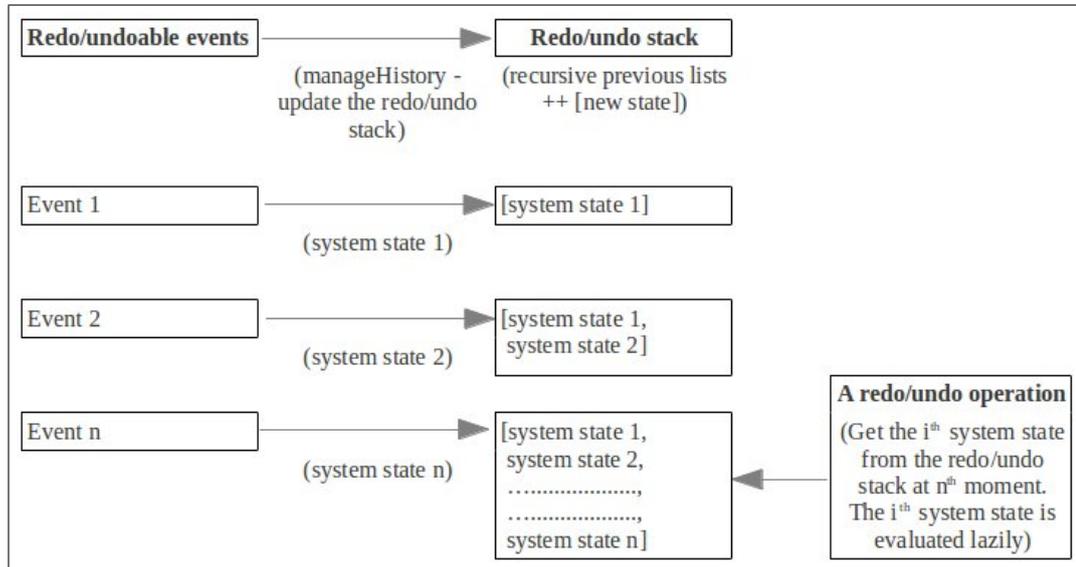


Figure 5.3: The redo/undo stack.

Evaluation of the Space and Time Behaviour of the Infinite Redo/Undo

An experiment has been conducted to evaluate the infinite redo/undo feature of HASKEU using the profiling facilities of the HASKELL system. The purpose of profiling is to improve the understanding of a program's execution behaviour. This profiling system assigns costs to cost centres. The time or space required to evaluate an expression is a cost. The program annotations around expressions are called cost centres. All the costs incurred by an annotated expression are then assigned to an enclosing cost centre. At run-time, the profiling system remembers the stack of the enclosing cost centres for any expression and at the end, it generates a call-graph of cost attributions. To test the infinite redo/undo feature, the system was used for a distinct number of events (100, 200, 500, and 1000). First all the undos were carried out and

then afterwards all the redos. Figure 5.4 shows the heap profile graph after 1000 events have been undone and then redone.

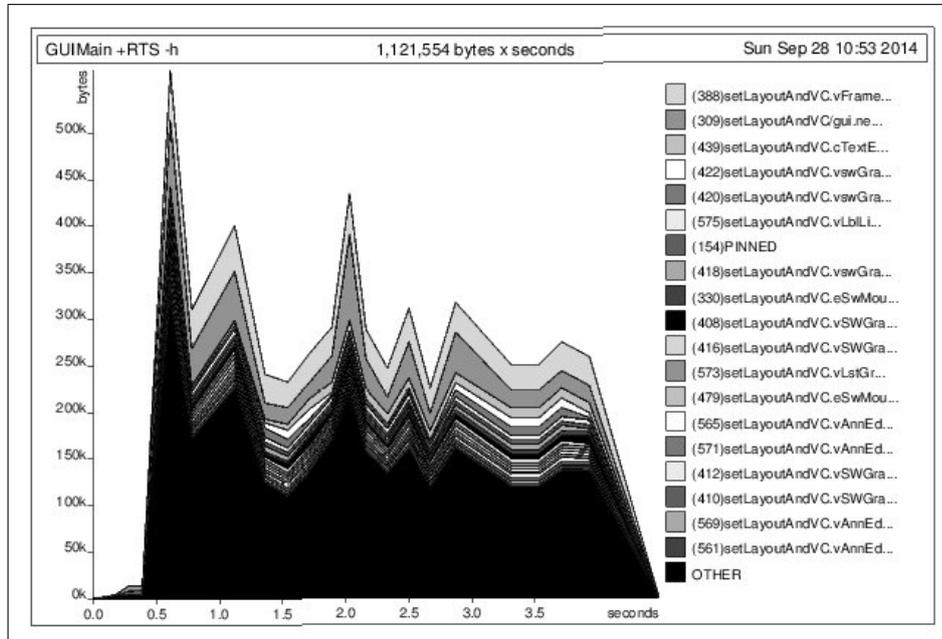


Figure 5.4: Heap profile to evaluate infinite redo/undo.

In an interactive program, time usage is surely governed by how fast the user can move the mouse and type text. So, the evaluation of the space and time behaviour of the infinite redo/undo in HASKEU should care about space usage only, not the time. If there was a growth in space consumption then, as time goes by more space is consumed (and not freed). On a graph, that would appear as a steady upward slope from left to right. The heap profile graph in Figure 5.4 does not have such a slope. This proves that there is no growth in space consumption by using infinite redo/undo in HASKEU.

5.2.4 To Select an Item

Three types of selection of an item are needed:

1. Select to show sub-items or to delete. It works on mouse click when the system's mode of operation is selected as `ModeSelection`.
2. Select to edit annotation. It also works on mouse click but when the system's mode of operation is selected as `ModeLabelEdit`.
3. Select to show tool-tips or to show the position of a new item. It works on mouse move on the screen for any mode of operation.

Any visual item in the system has a start position and a size information. Any mouse operation on the screen checks if the mouse position is inside an item boundary. A syntax tree traversal is needed to deselect and select an item. Any selection operation must deselect other previous selected items first. Selecting or deselecting items in the tree is as simple as calling the `Functor` typeclass method `fmap` as well-defined in the *haskell.src.exts* library for syntax tree.

```
selectTree p selTy tree      =
    fmap (selectAnn p selTy) tree
deselectTree tree            =
    fmap deselectAnn tree

selectAnn p selTy svi@(s, vi) =
    selectAnnOnType p selTy (insideGIArea p vi) svi
deselectAnn (s, vi) =
    (s, vi {selected=False, annEditMode=False, mseInside=False})
```

Selection of an item also checks in which part (upper or lower) of the item the mouse is positioned (`p` in the above code), because while adding a new item it is necessary to know if the new item will be added as an argument or as an application to the existing item.

As there are five panes for the five types of syntax tree nodes, a mouse click on a specific pane involves traversing a specific part in the syntax tree which can save time by not traversing the whole tree. Once an item is found to be selected, it stops traversing the rest of the tree. For example, the following code is used to select items in the pattern only of a selected function:

```
selFnPat p selTy lstMtch =
    map checkSelMtch lstMtch
  where
    checkSelMtch mtch@(Match a b lstP d e)
      | (isSel mtch) = Match a b (map checkInPat lstP) d e
      | otherwise    = mtch
    checkSelMtch mtch      = mtch
    -- as user can also write program textually,
    -- can create other |Match| pattren
    checkInPat pat        = selectTree p selTy pat
```

5.2.5 To Add a New Item

Adding a new item means adding a new node in the syntax tree. The following function node with default values is added to the syntax tree when a user adds a new function in the visual system. The function body must have a

default value, because of the function body structure defined in the syntax tree. The default function body is set as a literal string with the value of `--FunctionIsNotDefinedYet--`. A function body containing this default value is not shown to the user.

```
newFunc =
    let    var    = Lit sviInit (String sviInit
                                "--FunctionIsNotDefinedYet--"
                                "--FunctionIsNotDefinedYet--")
        rhs    = UnGuardedRhs sviInit var
        mtch   = Match sviInit (Ident sviInit "f") [] rhs Nothing
    in    FunBind sviInit [mtch]
```

The following code shows new pattern item nodes, in which some items have default values.

```
newPat patTy = case patTy of
    ModePatVar      -> PVar sviInit (Ident sviInit "a")
    ModePatWild     -> PWildcard sviInit
    ModePatEmptyLst -> PList sviInit []
    ModePatListCons -> PParen sviInit (PInfixApp sviInit p1 sCons p2)
                                where
                                    p1 = PVar sviInit (Ident sviInit "x")
                                    sCons = Special sviInit (Cons sviInit)
                                    p2 = PVar sviInit (Ident sviInit "xs")
    ModePatStr      -> PLit sviInit (String sviInit "abc" "abc")
    ModePatInt      -> PLit sviInit (Int sviInit 0 "0")
    ModePatChar     -> PLit sviInit (Char sviInit 'c' "c")
    ModePatBool     -> PApp sviInit
                                (UnQual sviInit (Ident sviInit "True")) []
```

To add a new node in the function body the following code is used, which creates different `exp` nodes with default values. The `if-then-else` syntax is replaced by the predefined function `cond`, whose type `cond :: Bool -> a -> a -> a` is given in the embedded test module.

```
expVar expTy = case expTy of
  ModeExpApp      -> Var sviInit (UnQual sviInit (Ident sviInit "a"))
  ModeExpOp       -> Var sviInit (UnQual sviInit (Symbol sviInit "+"))
  ModeListCons    -> Con sviInit (Special sviInit (Cons sviInit))
  ModeEmptyList   -> List sviInit []
  ModeCnstStr     -> Lit sviInit (String sviInit "abc" "abc")
  ModeCnstInt     -> Lit sviInit (Int sviInit 0 "0")
  ModeCnstChar    -> Lit sviInit (Char sviInit 'c' "c")
  ModeCnstBool    -> Con sviInit (UnQual sviInit (Ident sviInit "True"))
  ModeIfStmt     -> Var sviInit (UnQual sviInit (Ident sviInit "cond"))
```

The illegal case of applying a constant to an argument while adding a new item in the function body is checked. Any item can be added as an argument to an existing item, but an existing item cannot be added as an argument to the new item (e.g., constants). The following code validates this criteria of adding new items in the function body:

```
data ExpTy = ExpInput | ExpOutput
newExp expTy prevExp ExpInput =
  App sviInit prevExp (expVar expTy)
newExp expTy prevExp ExpOutput = case expTy of
  ModeExpApp      -> App sviInit (expVar expTy) prevExp
```

```

ModeExpOp      -> App sviInit (expVar expTy) prevExp
ModeListCons   -> App sviInit (expVar expTy) prevExp
ModeEmptyList  -> prevExp
ModeCnstStr    -> prevExp
ModeCnstInt    -> prevExp
ModeCnstBool   -> prevExp
ModeIfStmt     -> App sviInit (expVar expTy) prevExp

```

5.2.6 To Delete/Edit an Item

Items can be deleted from any of the five panes. When an user selects an item in a specific pane, the value of `areaOfOperation` is set to that specific pane, and then when a user clicks on the delete button the item in the specific pane is deleted. This is because two items can be selected in two different panes at the same time, but only the last selected item has to be deleted. The following code checks the area of operation before deleting an item:

```

deleteNode :: SystemState -> SystemState

deleteNode (SystemState f t g ty h) =

    let      Module (s, vi) mh mp imp lstDecl = shapeTree g

            mod = case (areaOfOperation g) of

                    GblFn   -> Module (s, vi) mh mp imp (delSelFn lstDecl)

                    LclFn   -> Module (s, vi) mh mp imp (delSelLclFn lstDecl)

                    FnArg   -> Module (s, vi) mh mp imp (delSelFnPat lstDecl)

                    LclFnPat-> Module (s, vi) mh mp imp (delSelLclFnPat lstDecl)

                    FnBody  -> Module (s, vi) mh mp imp (delSelFnExp lstDecl)

    in      (SystemState f t g{shapeTree = mod} ty h)

```

A cascade delete is implemented in this system, so that after deleting an

item there will not be any syntax error. Cascade delete in this system means all the sub-items of a node will be deleted if a node is deleted. Deleting an item from global functions, local functions, global function pattern, or local function pattern is as simple as deleting an item from a Haskell list. Deleting an item in a function body involves a recursive call of a function `delExp` (given below) to check which item is selected in the tree. If an item is selected from two items in an `App` node, then the function returns the other unselected node. If none of the items in the `App` node is selected, then the function calls itself recursively to check any selected item in the subtrees. If a selected item is the last item in the tree, then the function body is set as the literal string with the default value of `--FunctionIsNotDefinedYet--`.

```

delExp exp@(App svi eOne eTwo)
    | (isSel eTwo) = eOne
    | (isSel eOne) = eTwo
    | otherwise   = App svi (delExp eOne) (delExp eTwo)

delExp exp@(Paren svi eOne) =
    Paren svi (delExp eOne)

delExp exp@(Var _ _) =
    if      (isSel exp)
    then   Lit sviInit (String sviInit
                        "--FunctionIsNotDefinedYet--"
                        "--FunctionIsNotDefinedYet--")
    else   exp

```

Editing the annotation of an item also involve traversing the syntax tree to find a selected item first, and then the annotation is replaced by the value

contained in an annotation text control.

5.2.7 To Validate Syntax Error

The annotation value is always checked before it is changed to avoid syntax errors in the visual system. Some of the coding which performs the checks are shown below.

The function `chkModuleNameBeginsWith` checks if a module name begins with a upper case letter or not:

```
chkModuleNameBeginsWith prevStr newStr
  | (((ord $ head newStr) >= 97) && ((ord $ head newStr) <= 122))
    = prevStr
  | otherwise
    = newStr
```

The following function checks if a function name is valid:

```
chkValidFuncName prevStr newStr
  | (newStr == "")
    = prevStr
  | (((ord $ head (newStr)) >= 65) && ((ord $ head (newStr)) <= 90))
    = prevStr
  | isNotAnString newStr
    = prevStr
  | otherwise
    = newStr
```

The following function checks if a string is an invalid operator:

```

isNotAnOperator newStr =
    or $ map chkEachChar newStr
    where
        chkEachChar ch = ((ord ch < 32) || (ord ch > 63))

```

The following function checks if a string is an invalid integer:

```

isNotAnInt newStr =
    or $ map chkEachChar newStr
    where
        chkEachChar ch = ((ord ch < 48) || (ord ch > 57))

```

5.2.8 To Adjust wxHaskell Events

The *wxhaskell* events need to be adjusted in order to be used in this functional reactive programming based framework. There were a small number of events missing in the wxHaskell library, but they were necessary for the implementation of this system. wxHaskell is built on top of wxWidgets which is a comprehensive C++ library portable across all major GUI platforms. wxHaskell consists of two libraries, WXCore and WX. The WXCore library provides the core interface to wxWidgets its functionality. The WX library is implemented on top of WXCore and provides many useful functional abstractions to make the raw wxWidgets interface easier to use. The function `newEvent` has been used to create new events. Some examples are given below:

The `windowScroll` event works when a user scrolls a window pane using the mouse or keyboard:

```

windowScroll :: WX.Event (Window a) (EventScroll -> IO ())

windowScroll = WX.newEvent    "windowScroll"

                    windowGetOnScroll windowOnScroll

```

The keyboardUp event works when a user releases a keyboard button:

```

keyboardUp :: WX.Event (Window a) (EventKey -> IO ())

keyboardUp = WX.newEvent    "keyboardUp"

                    windowGetOnKeyUp (windowOnKeyUp)

```

The keyboardDown event works when a user presses a keyboard button:

```

keyboardDown :: WX.Event (Window a) (EventKey -> IO ())

keyboardDown = WX.newEvent    "keyboardDown"

                    windowGetOnKeyDown

                    (windowOnKeyDown)

```

All the wxHaskell events need to be represented as FRP events to be used in this design pattern. We used some functions provided in the `reactive-banana.wx` library for this representation process.

The function shown below represents the `windowScroll` event in the wx-Haskell library to an FRP event using the `event1` function.

```

evSwScroll :: Frameworks t =>

    ScrolledWindow () -> Moment t (Event t EventScroll)

evSwScroll sw = do

    eScroll <- event1 sw windowScroll

    return (eScroll)

```

5.3 The Views

5.3.1 To Draw the Shape Tree

The visual view of a program is drawn from the shape tree. Any occurrence of an event (keyboard press, mouse click, mouse move, button click etc) in any program editor (textual or visual) changes the shape tree and the visual layout of the program is created. Then, the visual display of the program is drawn from the shape tree by rendering graphics on wxHaskell panes. So, to draw the visual items on the screen required a tree traversal. The following class is used to traverse the tree to draw items of the shape tree:

```
class DrawShapeTree a where

    drawItem :: DC () -> ModeOfOperation -> a -> IO ()

    drawItemMB :: DC () -> ModeOfOperation -> Maybe a -> IO ()

    drawListItems :: DC () -> ModeOfOperation -> [a] -> IO ()

    drawItem dc dMode shp          = return ()

    drawItemMB dc dMode Nothing    = return ()

    drawItemMB dc dMode (Just shp) = drawItem dc dMode shp

    drawListItems dc dMode lstShp  =
        sequence_ (map (drawItem dc dMode) lstShp)
```

where `drawItem` function draws a single item, `drawItemMB` draws a item with a `Maybe` data type, and `drawListItems` draws a list of items. We always pass the `ModeOfOperation` value of the `SystemState` is always passed to these functions, because the item view depends on the `ModeOfOperation` (eg, if the mode of operation is selection then “add new item” icons will not be shown).

The rendering of graphics is not a huge operation in this system, because the whole shape tree is not drawn at one time, rather only the list of function names of the module and the details of a selected function are shown. Because of lazy programming, the auto layout does not have to calculate position and size information of all the items. So, a user can get immediate feedback of the visual layout while editing a program on the screen. An item can be drawn using low-level graphic operations on the screen. By low-level it is meant that an item consists of some or all of rectangles, triangles, lines and text. The coding shown below display an function item on the screen:

```
let Point x y = itemPosition vi
drawRect dc (rect      (itemPosition vi) (itemSize vi))
              [        color := black,
                brush := brushSolid (colorRGB 96 123 139)]
drawText dc (synonym vi) (pt (x+5) y) []
```

The drawing of an item includes some or all of the following operations:

1. Drawing the boundary of an item.
2. Drawing specific icons for items (eg, list icon, integer icon etc).
3. Drawing argument boxes for function applications.
4. Drawing a group box and joining it to child nodes for function applications.
5. Drawing the annotation of an item.
6. Drawing another boundary if the item is selected.

7. Drawing cross marks if an item contains errors.
8. Displaying tooltip text when the mouse hovers on an item.
9. Drawing special icons for certain items (undefined application, recursive application, an item in the function body which is also parameter).

5.3.2 To Show the Annotation Text Editor

When an item's editing mode is true, a text editor becomes visible with the same size and position of the item, and shows the item annotation in it. The following code shows how to set the attributes of an annotation text editor (the view of the text editor) by the interface logic function (starting with "il") to edit the annotation of a local function name.

```
let vAnnEdtVisiLclFn = \m ->
    view (txtAnnEdtLclFnVw, "visible") m ilAnnEdtVisiLclFn
let vAnnEdtTxtLclFn = \m ->
    view (txtAnnEdtLclFnVw, "text") m ilAnnEdtTxtLclFn
let vAnnEdtPosLclFn = \m ->
    view (txtAnnEdtLclFnVw, "position") m ilAnnEdtPosLclFn
let vAnnEdtInsPtLclFn = \m ->
    view (txtAnnEdtLclFnVw, "insertionPoint") m ilAnnEdtInsPtLclFn
```

The wxHaskell text controls do not change their position with the scrolling of their container pane. So, a function called `origToVirtuScroll` is used to force the text control by calculating an item's virtual position (the original position of an item less how much the pane is scrolled) in a scrolled screen.

```

origToVirtuScroll    ::      SystemState
                    ->      (String, ItemState)
                    ->      (String, ItemState)

origToVirtuScroll m (s, origVi) =
    let sp          = getSpecificScrollPos m
        origP       = itemPosition origVi
        virtuVi     = origVi {itemPosition = pt
                               (pointX origP - pointX sp)
                               (pointY origP - pointY sp)}
    in (s, virtuVi)

```

Then, any editing in the annotation text editor is controlled by its controller to change the system state.

5.3.3 To Calculate Row and Column of a Text Control from Insertion Point

A wxHaskell text control does not provide row and column numbers of the current cursor position. Rather it gives an insertion point which combines the row and column numbers into one number. In contrast, the syntax tree created by the `haskell.src.exts` module works with the row and column positions of a program. So, the row and column positions need to be calculated to focus the visual item of a textual item in the current cursor position and vice-versa. Also, the current row and column positions need to be shown on the screen in order to aid textual programming. The following function is used to get row and column positions from the insertion point:

```

getRowColFromInsPt :: String -> Int -> (Int, Int)

getRowColFromInsPt s ins =

    let    lenLines = map length (lines s)

        in    getLnCol (-1) 0 0 ins lenLines

```

The function below is used to get insertion point from the row and column positions:

```

getInsPtFromRowCol :: String -> Int -> Int -> Int

getInsPtFromRowCol s l c =

    let lenLines = map length (lines s)

        linesErr = take (l-1) lenLines

    in sum linesErr + (l-1) + (c-1)

```

5.3.4 To Enable/Disable Redo/Undo Buttons

To implement Redo/Undo, the states of the system are stored. Undo brings the previous state back and redo moves forward to the next state. A stack is used to support redo/undo, and the current position contains the index of the current state in the stack. The disable state of the redo or the undo button indicates that there is no more redo or undo operation to be done. The following functions are used to find the disable/enable states of the redo/undo buttons.

```

isEndOfUndo :: SystemState -> Bool

isEndOfUndo (SystemState _ _ _ _ (ModelHistories h cPos)) =

```

```

cPos > 0

isEndOfRedo :: SystemState -> Bool

isEndOfRedo (SystemState _ _ _ _ (ModelHistories h cPos)) =

    (length h) > (cPos + 1)

```

5.3.5 To Add More wxHaskell Attributes

There are some wxHaskell attributes missing in the the current wxHaskell library, but they are necessary for the implementation of this system. The following functions are used to create new attributes: some of the code created is given below:

```

insertionPoint :: WX.Attr (TextCtrl a) Int

insertionPoint =      newAttr "insertionPoint"

                    textCtrlGetInsertionPoint (textCtrlSetInsertionPoint)

mouseCursor :: WX.Attr (Window a) (Cursor ())

mouseCursor = newAttr "mouseCursor" windowGetCursor

                (\w c -> do      windowSetCursor w c

                                return ())

```

5.3.6 Difficulties and Achievements

It was difficult to find the relevant libraries and to learn what they do from little documentation in some cases, and then to apply and customize them. Using wxHaskell library widgets was also difficult. Sometimes some alternatives were chosen as some widgets were found not to be in working order. Some

new widget events and attributes needed to be added. More work needed to be done in order to draw shapes with low-level graphics. There were also difficulties while adjusting the *wxHaskell* library with the functional reactive library *reactive.banana*, because the *reactive.banana.wx* library to do this job is not complete for the purposes needed. Understanding the syntax tree in the *haskell.src.exts* library was difficult as there is very little documentation and examples provided.

A library was developed in HASKEU which called `MVC_WX.lhs` (given in Appendix E), where many other *wxHaskell* events and attributes were created and adjusted for the *reactive.banana* library. This may be useful for developers who want to work with *wxHaskell* and *reactive.banana*. Some of the functionality from this library was sent to the author of *reactive.banana* and was much appreciated. In the next chapter the experiment process and result of a usability test will be shown.

Chapter 6

Usability Experiment and Result

6.1 Experiment

A usability test was conducted to evaluate the system by testing it on end-users. This way, direct feedback from the real users were obtained about the usability of the system.

The usability test was designed as follows: a programming exercise was devised; some instructions were written (as suggested by Robins and Roundtree (Robins et al., 2003)) to complete this exercise and a pilot test conducted with one user; the problems with the instructions were fixed, and then tried with four users; the following details were measured — the time taken to complete the test, accuracy (correctness or incorrectness), and emotional response

(what the users think) using a questionnaire (given in Appendix D.1). The next step was to check whether the usability testing met the usability goals (described in Section 6.1.5)) as suggested by Mayhew and Nielsen (Mayhew, 1999; Nielsen, 1993).

6.1.1 User Manual

A user manual was prepared to give assistance to the end-users while using the programming system as suggested by Blake and Bly (Blake and Bly, 1993). This manual was provided in a traditional printed format as well as in an electronic (pdf) format as suggested by Price and Brockmann (Price and Apple Computer, 1984; Brockmann, 1990). The user manual is given in Appendix C.

6.1.2 Selection of End-Users

This experiment focused on the programming knowledge aspect of the end-users, and two groups of participants were selected based on their programming experience. The first group consisted of skilled programmers, and the second group were of non-programmers and the both groups had no prior knowledge of functional programming. The skill level in programming was measured based on how long a person had been working with programming languages, and how many programming languages they knew. The programming skill of each end-user was observed before the selection process, and then a decision was

taken if the person was to be selected.

It was decided to recruit end-users by invitation from different professions, but with no previous involvement with this research. A plan was set to conduct studies of a small set of five end-users where two of them were skilled programmers and the others were non-programmers. Each person in the skilled programmer group knew of at least three programming languages (other than purely functional programming languages), and they had used these languages regularly in the previous four years. The three non-programmers were two business administrators and a physicist. The first person to test the system was one business administrator who was only able to fix problems with the instructions for the programming exercise and was not included in the final test. (the instructions are shown later in Section 6.1.4). It was assumed that if the instructions were easily understandable by a non-programmer, then the skilled programmers could also understand it. The age group of all participants was between 30 and 40, and each person had at least an undergraduate degree. Three of the participants were former graduates of the University of Gloucestershire.

6.1.3 To Devise a Programming Exercise

After the selection of the end-users was finalised, these end-users performed the same programming tasks with both the textual and visual programming

systems in order to collect data to perform a qualitative and quantitative comparison between the two sets of users and between the textual and visual approaches. The quantitative data measured in this programming exercise were both the speed with which end-users could create functional programs (how long does it take?) and the accuracy (correctness or incorrectness) of the completed program. The qualitative data were appropriateness of those programs (how well do they meet their specification?). The end-users were also encouraged to ask questions as they worked. The qualitative data was also taken from the questionnaire (given in Appendix D.1), but recorded anonymously.

The purpose of this programming exercise was to write a **reverse** function. The reason this function was chosen was because most of the visual actions of the system (globals, locals, clauses, patterns, function body and recursion) are involved in building this function. Two of the selected end-users (one programmer and one non-programmer) were asked to complete the program textually first and the other two were asked to complete the program visually first.

6.1.4 Instructions for the Exercise

After the first user had completed this exercise as a pilot, the instructions were adjusted as shown below:

Exercise: Write a Haskell function to return the elements of a finite list in reverse order.

Sample Input/Output:

Input : [6,2,4,9,5]

Output : [5,9,4,2,6]

Important Concepts (Pattern matching, Recursion):

The two important concepts needed to complete the `reverse` function are pattern matching and recursion. In Haskell, very concise and elegant solutions to problems can be worked out using these two concepts. Recursion is the way of defining a function in which the function is called inside its own definition. Pattern matching is a dispatch mechanism of choosing which pattern can be matched by a given function invocation.

Function Structure:

Table 6.1: Function Structure

No	Item	Description	Steps
1	Global function name	Global function name	Create a function with the name <code>reverse</code> .
2	Global clause 1 pattern	The only input parameter in pattern is a list of items	Add a parameter variable <code>l</code> .

3	Global function body	It calls a local function <code>rev</code> (which will be defined later). This function call should be applied to two arguments — 1) the original list — <code>l</code> and 2) an empty list — <code>[]</code> where the items of the original list will be added in a reverse order.	Apply <code>rev</code> to <code>l</code> and <code>[]</code> .
4	Local function	A local function with two clauses. Each clause has two parameters in its pattern. The first parameter is the original list, items of which are added to the second parameter in a reverse order.	Create a local function with name <code>rev</code> which should have two clauses.
5	Local clause 1 pattern	Clause 1 has the edge condition. If this condition is not specified then the function will produce an infinite loop. The edge condition is the empty list. When all the items from the original list are pulled out, it will become an empty list and the function should return the reverse list. So, the first parameter in the pattern of the clause 1 is an empty list <code>[]</code> , and the second parameter in the pattern of the clause 1 is the reversed list.	Add two parameters in the pattern — empty list <code>[]</code> and parameter variable <code>a</code> .
6	Local clause 1 body	The function body of clause 1 returns just the reversed list <code>a</code> .	Return <code>a</code> .
7	Local clause 2 pattern	In clause 2, the original list is split into a head and a tail (<code>x:xs</code>), so that the head can be added at the end of the reversed list and the tail is split into a head and a tail again by calling the function recursively until the tail is empty. So, the first parameter in the pattern of the clause 2 is a split list (<code>x:xs</code>), and the second parameter in the pattern of the clause 2 is the reversed list.	Add two parameters in the pattern — list (<code>x:xs</code>) and parameter variable <code>a</code> .

8	Local Clause 2 Body	Call of this local function is called recursively where the first argument is the tail of the original list xs and the second argument is the reversed list with the head added to it using the (:). So, the second argument is apply of (:) to two arguments — head x and reversed list a .	Apply rev (so that recursively) to two arguments — xs and (:), where the second argument (:) also have two arguments, x and a .
---	------------------------	---	---

6.1.5 Usability Goals

The following usability goals were set based on the above task. These usability goals were checked against the results obtained from the usability testing.

- Both the experienced programmer and non-programmer groups should complete the task faster visually than textually;
- The system should allow a smooth transition from visual to textual and vice-versa as the users' programming expertise increases. For example, if a person learns the textual system first, he can finish the task visually faster than a person who has not learnt either system yet.
- The experienced programmer group should complete the task faster than the non-programmer group both textually and visually, but the percentage of performance difference between the two groups should be higher using the textual interface than the visual one. A percentage measure to calculate the performance difference between the two groups in either system (see Section 6.2.1) will be applied to the completion times using

both interfaces by the two groups, and it is expected that this measure will be smaller for the visual task.

The researcher also performed the above task to confirm the possible expected outcomes.

6.1.6 Experiment Process

All participants were asked to read and sign a consent form (see Appendix D.2) before the the session started. The purpose of the consent form was to provide participants with a clear statement that described the aims of the experiment and the nature of involvement of participants. All participants did the experiment individually. The experiment consisted of two different sessions — a textual experiment session and a visual one. Each session was followed by a short training session to introduce each textual and visual system. In this training session, the end-users were trained to program a function `append`. This training was sufficient for them to use the system without making many mistakes. This `append` was chosen so that the exercise would be similar to the training, so that there would no confusion.

The training sessions were followed by a short 30 minutes tutorial. The textual tutorial was provided by the website <http://learnyouahaskell.com/>, written by Miran Lipovaa (Lipovaa, 2011). This was a very good external validity of the textual training session as it was designed for beginners. The

user manual, created by the researcher, was used as the visual tutorial. These training sessions were also important to bring all participants to the same level before they did the experiment. During the experiment, the participants used their own machines and were accompanied by the researcher. If they had any relevant questions, then these were answered by the researcher. The participants were always encouraged to see the effects of the textual system in the visual system and vice-versa while programming, so that one or the other experiment sessions would be easier to do. At the end of each experiment, participants described their experience so far by answering a questionnaire (see Appendix D.1).

6.2 Result

The completion rate of the above task in both the textual and visual systems by the four participants was 100%. The correctness of the finished tasks in visual programming was 100% and in textual programming was 100% and so the overall correctness was 100%.

6.2.1 Performance Comparison Using Quantitative Data

Performance Comparison for the First Usability Goal

The first usability goal was that both experienced programmer and non-programmer groups would complete the task faster visually than textually. The quantitative data obtained from the completion times indicated that

three participants (two non-programmers and a programmer) had completed the task faster visually than textually. One programmer had completed (in the sequence of visually first textually later) the task faster textually than visually. This could be because the programmer already knew about some other textual language and/or in doing the visual exercise first, he/she had learnt the textual syntax. The completion times of the four participants using the textual system were recorded as 20 mins, 11 mins, 36 mins, and 22 mins, while the completion times using the visual system were recorded as 9 mins, 13 mins, 11 mins, and 19 mins (see Table 6.2), thus achieving the first usability goal in this small sample. Figure 6.1 shows a comparison of completion time between textual and visual system.

Table 6.2: The completion times (in minutes) by the participants using the both textual and visual systems

Participant	Textually First	Visually Later	Visually First	Textually Later
Programmer 1	20 mins	9 mins		
Programmer 2			13 mins	11 mins
Non-Programmer 1	36 mins	11 mins		
Non-Programmer 2			19 mins	22 mins

This figure shows that the average completion time textually was 22.25 mins, and average completion time visually was 13 mins. This suggests the task completion time when performed textually is almost double that when it is performed visually.

Performance Comparison for the Second Usability Goal

The second usability goal was that the system should allow a smooth transition from visual to textual system and vice-versa as programming expertise

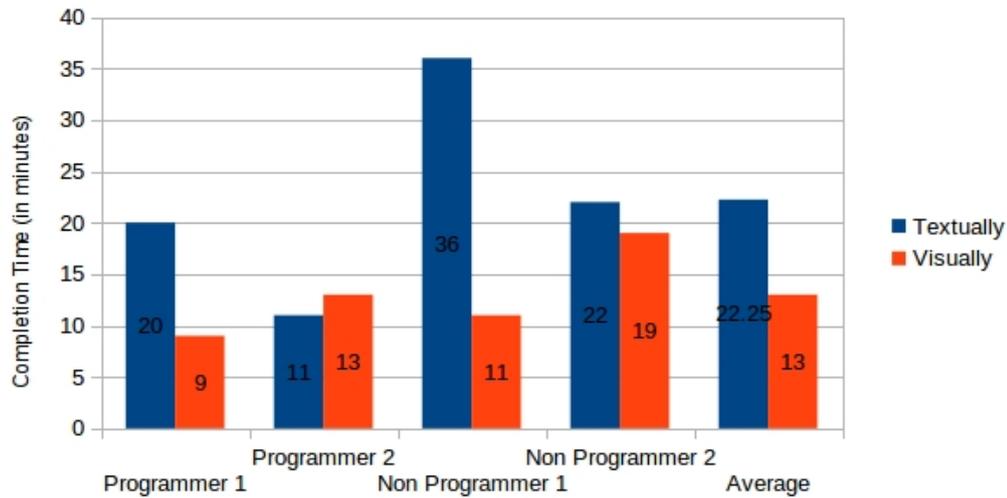
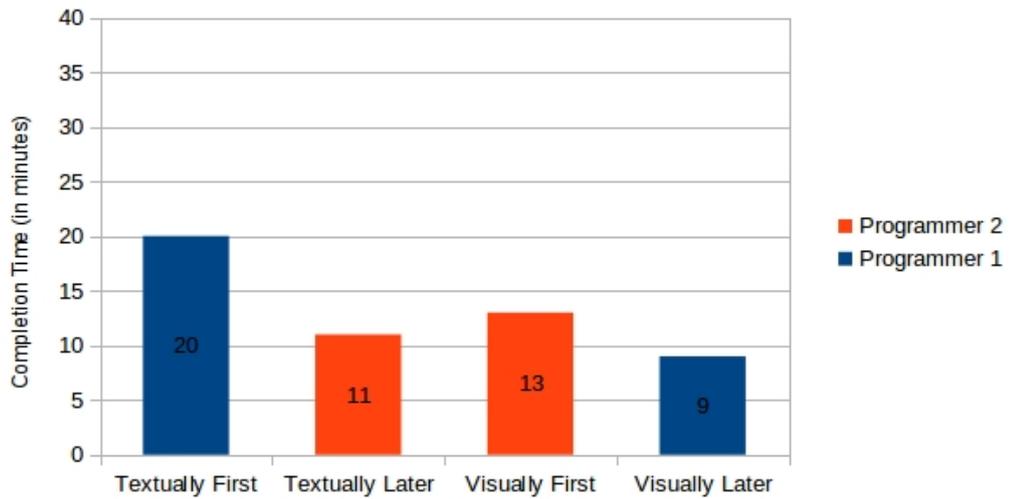


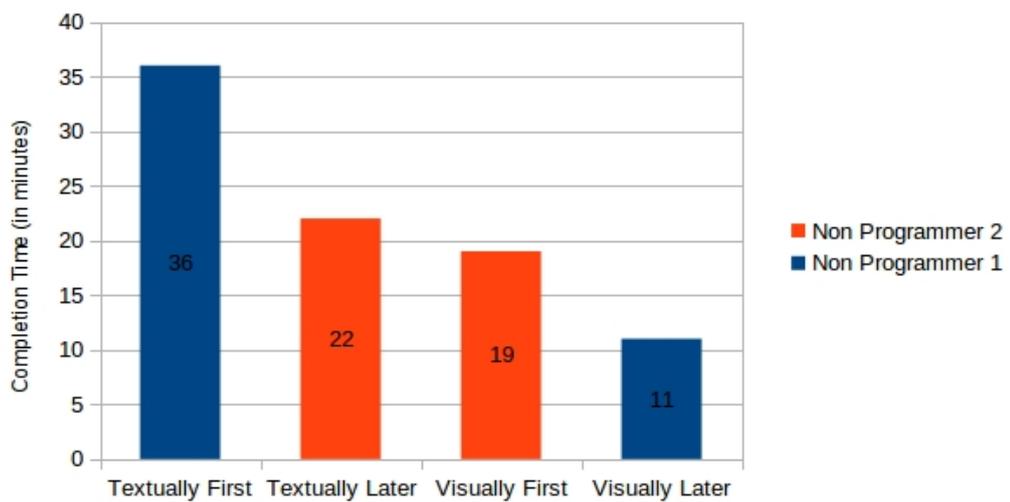
Figure 6.1: Performance of all participants - visually vs textually.

increases. From the quantitative data obtained from the completion times, it can be seen that each participant could complete a program faster either visually or textually if he/she had already completed the program in the other system (see Figure 6.2).

In this programmer group, the participant who completed the task visually first completed the task textually 9 mins faster than the other participant who completed the task textually first (see part of Figure 6.2 in Figure 6.3). In the programmer group, the participant who completed the task textually first completed the task visually 4 mins faster than the other participant who completed the task visually first. The same was true in the non-programmer group. The participant who completed the task visually first completed the task textually 14 mins faster than the other participant who completed the task textually first. The non-programmer participant who completed the task



(a) Transition rate from one system to the other of programmer group.



(b) Transition rate from one system to the other of non-programmer group.

Figure 6.2: Transition rate from one system to the other of two groups.

textually first completed the task visually 8 mins faster than the other non-programmer participant who completed the task visually first. This suggests a transition ability of the system from visual to textual and vice-versa as programming expertise increased.

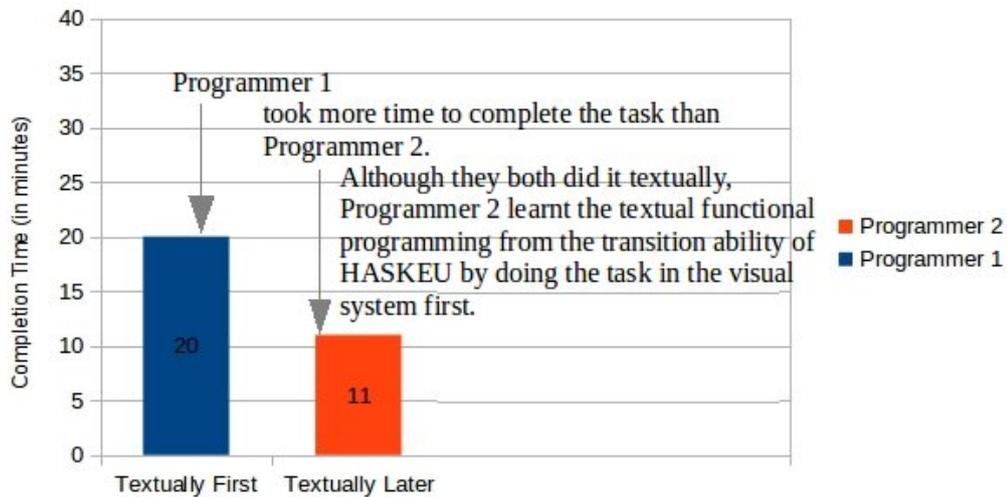


Figure 6.3: Transition ability of the system.

Performance Comparison for the Third Usability Goal

The third usability goal was that the experienced programmer group should complete the task faster than the non-programmer group both textually and visually, but the percentage of performance difference between the two groups should be higher using textual interface than the visual one. The percentage performance difference is calculated by using the following formula:

In any system (visually or textually),

the percentage performance difference =

$$(\text{Average CT by PG} - \text{Average CT by NPG}) /$$

$$(\text{Maximum CT by any user}) * 100$$

where

CT = Completion Time

NPG = Non-Programmer Group

PG = Programmer Group

Using this formula, the percentage performance difference rate between the two groups in the textual system was found to be 37.5% and the performance difference rate in the visual system was found to be 21.05%. This indicates that non-programmer group are closer to the programmers when performing the task visually rather than textually.

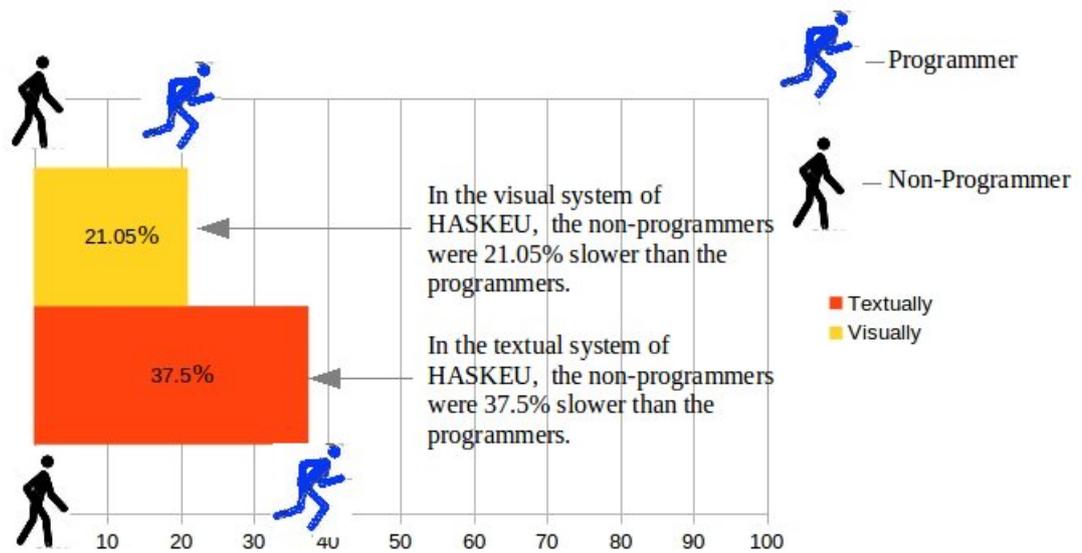


Figure 6.4: Performance difference rate between two groups.

6.2.2 Suggestion from the Qualitative Data

It was intended to perform qualitative analysis on the reasons why users did not complete the assigned tasks using HASKEU. However, this was not possible as all users completed the tasks and this aspect will be deferred to future work. This exercise was not tested with a programmer with functional programming knowledge. From the answers to the questionnaire, a programmer noticed an inconsistency, which was that in visual system a number is shown for each clause and in the textual system the function name in a clause is shown. A programmer suggested a dataflow line would be better than a recursive symbol. The non-programmers did not find the type display very useful. Maybe later if they work on functional programming they will find it useful. None of the participants found the textual error reporting very useful, but they thought that showing errors visually was a good idea. A programmer commented that preventing syntax errors while editing in the visual system is a good idea, but he suggested that having a beep or using a colour change if a user presses a wrong key would be helpful. Both the programmers mentioned in the free text box on the questionnaire that having the option to use both the textual and visual systems together really makes program development easier. An unexpected benefit 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.

6.3 Conclusion

To summarise, for this usability test, the visual system can be seen as a good starting point for end-users to learn functional programming, and the textual system was found helpful too as the end-users' expertise increased. The testing has been done with a very limited number of end-users and only one exercise has been given. There could be other suggestions for improvements from the end-users if the testing was done with larger numbers and over a longer time period. The next chapter contains the conclusions.

Chapter 7

Conclusions

In this thesis, an approach is given for implementing a combined textual and visual programming system of Haskell. This chapter discusses the achievements, suggests the limitations and possible future works of this research, and then concludes.

7.1 Achievements

The following achievements have been realised in this research which are the direct outcome of fulfilling the research objectives given in Section 1.7:

1. The design and implementation of a novel programming system in Haskell to support both visual and textual programming allowing for a smooth transition from one to the other as the end-user's programming expertise increases (achievement of research objectives 1, 2, 3).

2. The fact that the changes occur in both textual and visual systems simultaneously has been found to be helpful by end-users in their program development (achievement of research objectives 1, 2, 3).
3. An implementation of a framework of the Model-View-Controller (MVC) design pattern in a functional programming language has been achieved. Programmers brought up on object-oriented programming languages may benefit from this framework by being able to use this very useful and widely used design pattern in functional programming languages (achievement of research objective 3).
4. The thesis has proposed and implemented a visual system for functional programming, which produces error-free syntax. Sometimes it is really hard, specially in textual programming, to understand and fix an error from a syntax error message. This visual system works with syntax tree nodes, and it is not possible to create a node in the syntax tree which has been the result of a syntax error. The system also checks if any editing of annotation causes a syntax error and does not allow incorrect changes to be made to an annotation for that specific node (achievement of research objective 2).
5. As an implementation of an on-time system, this system shows type information within a visual program while creating/ editing a program, rather than showing it after compilation (achievement of research objective 2).

6. The system has produced simple but useful visual error reporting of type errors for end-users. The visual type error reporting has helped end-users to understand and locate a type error more precisely. As a summary of this error reporting, cross marks in the dataflow arc in the function body indicate type errors and both end-points contain the type information shown as tooltip text (achievement of research objective 2).
7. The system has shown that an infinite redo/undo facility is possible in a system (theoretically and practically), as a consequence of the lazy evaluation nature of Haskell. During the testing process by the researcher and during the usability test by the end users, no reports of crashes were detected because of any infinite redo/undo action. Also, an experiment has been conducted to evaluate the space behaviour of the infinite redo/undo feature of HASKEU using the profiling facilities of the HASKELL system (achievement of research objective 3).

7.2 Limitations and Suggested Future Developments

The visual programming system in HASKEU is incomplete. To design and implement a complete visual representation of Haskell syntax and a complete visual programming tool needs a huge amount of work. In HASKEU, only very simple Haskell syntax can be represented visually. Some other Haskell constructs (eg, guarding, case, list comprehension) can be expressed from this

simple syntax but possibly clumsily, and it is not possible to create some syntax (e.g., type classes and instances, data structures) from this simple system. To establish a formal visual programming system, even for end-users, the necessity of a complete visual notation of a Haskell syntax tree needs more detailed research on cognitive and user-interface design issues, and possibly a prototype editor implementation. A more detailed usability study will be necessary.

The following limitations of the thesis have been identified and the way to overcome them in future developments are specified below:

1. Haskell has a vast syntax and only a small portion is covered in this thesis in a visual representation. A complete visual notation and a programming tool to support all the notations can be seen as a future goal. A complete visual notation can be given by carefully considering HCI issues to design many other icons and the dataflow between them and then studying their usability.
2. The system has implemented low-level graphic operations of wxHaskell on the screen to show the effect of direct manipulation. By low-level is meant that an effect consists of some or all of rectangles, triangles, lines and text. The quality of graphics to show the effects of direct manipulation could be improved. Additional graphics quality (e.g., use of picture, animation, special widgets) to show the effect of direct manipulation would be beneficial.
3. One important advantage of functional programming languages is the

ease with which one can embed a domain-specific language (DSL). This programming system is currently a general purpose programming language. Designing visual programming for task-specific languages has some advantages to offer end-users, as it affords users ready understanding of what the primitives of the language do. The design of visual programming editors for domain-specific libraries (eg, animation, music synthesizers, robots) could be thought about as future plans.

4. A short usability study was organized on a very small set of end-users. The number of users should be increased to get more useful and accurate results from the usability study. If an advanced level of visual notation is created at a later date, then an advance level usability study by actual Haskell programmers would be beneficial.
5. More complete visual error reporting is one area for research. Right now the system produced for this thesis has very simplified error reporting. Some of error representation sometimes may not be understood properly in order to be fixed by user. A more detailed understanding of textual error messages, may be helpful to design the visual error reporting for an advanced level of programming.
6. The textual editor is a very simple, plain text editor. The textual programming can be improved by implementing some advance techniques such as syntax directed editing.
7. An automatic dataflow layout has been implemented to show a visual

program. Sometimes, manual layout is also useful for some advanced users or a combination of both can make program development easier. A future development could be a mixed layout and then to have a usability study performed using it.

8. An integrated development environment (IDE) for this Haskell programming system could also be considered, where a compiler and a debugger could be integrated with the system.

From the analysis of the usability test results on end-users, it can be predicted that visual programming may not be a replacement for textual programming, and textual programming may not be a replacement for visual programming. They can support each other in the end-user's learning, development and maintenance activities of functional programming.

Finally, HASKEU is still at an very early stage. Based on the usability testing on end-users, it can be hoped that HASKEU will inspire end-users to learn and improve their functional programming skills. Adding more visual notations would enable advanced users to program with more advanced features. In chapter 1, the research statement was specified as "It is feasible to develop an end-user functional programming system that consists of a visual programming system and a textual programming system and for the end-user to have a smooth transition between the two, particularly as the end-users' programming expertise improves and increases. This end-user functional programming system can be implemented in a functional paradigm.", and this

has been validated. The contributions of the HASKEU system, which combines various technologies and the benefit of the combined visual and textual programming for functional programming described in this thesis promise to have a broad impact on the usability of functional programming in the future and the increase in popularity of functional languages.

Appendices

Appendix A

Appropriateness of Using Software Analysis and Design, and UML Diagrams for Functional Programs

This Appendix discusses the appropriateness of using software analysis and design (see Section A.1), and the appropriateness of using UML diagrams (see Section A.2) for functional programs.

A.1 Appropriateness of using software analysis and design

Although the Software Engineering Life Cycle (SDLC) gives a general overview of ordering different phases of software engineering, the implementation and documentation of the phases (from analysis to maintenance) depend on the underlying programming paradigm of the programming language on which the system will be developed.

A programming language can support good software design. The structured programming paradigm was introduced in 1960 to improve the clarity, quality, and development time of a program with the use of new features such as subroutines, block structures and for and while loops (Dahl et al., 1972). Structured programming is supported by a structured system analysis and design method (SSADM) (Downs et al., 1988). The structured paradigm is consistent for all the programming languages it supports (some of the initial languages were: ALGOL (Grune, 1977), Pascal (Jensen and Wirth, 1974), PL/I (Hughes, 1986), and Ada (Barnes, 1984)). This SSADM is not used now.

Now, the object-oriented paradigm has succeeded the structured paradigm with many new concepts such a class, object, inheritance, encapsulation, polymorphism etc (Booch, 1994; McLaughlin et al., 2006). Object-oriented developments are built on small encapsulated units that provide an interface to

be used by others. Hence, in a sensibly developed object-oriented program, reusable elements are identified and changes only have a local effect. Such a sensible development needs practice and OO developers are fortunate that they can get support from OO analysis and design methodologies (OOADM). Any popular methodology like OOADM has a graphical modelling language and a supporting *CASE (computer-aided software engineering)* tool which provide a graphical representation of the system. Examples of popular object-oriented programming languages are Java (Booch, 1994), C++ (Stroustrup, 2000), Smalltalk (Kay, 1996), Objective-c (Kochan, 2009). Many earlier OOADMs can be found such as the Booch method (Booch, 1994), Fusion, (Coleman et al., 1994), and OMT (Rumbaugh et al., 1991). But the most recent UML (Unified Modeling Language) (Booch et al., 2005) has eclipsed many of the earlier development methodologies and is ubiquitous nowadays. It is a fact that early OOADMs and supporting notations were unified into UML (hence the name) and this is the most common model now. Consequently, the support provided by UML has an important role when choosing a software development environment. One popular CASE tool is Rational Rose (Software, 1994) which supports most of the methods just mentioned including UML.

The functional programming paradigm does not need the support of such a modelling language and case tool. Although the functional programming community is well-established and with a broad user-base, it still does not feel the necessity to implement a CASE tool or software development methodology to

support analysis and design in a functional language. Some notable progress can be seen in the areas of testing and debugging such as Hunit (Claessen et al., 2010) and QuickCheck (Claessen and Hughes, 2011), for testing and GHCi (Himmelstrup, 2006), and HAT (Chitil and Luo, 2007) for debugging. This shows that some common software development support tools do exist for functional programming, so it is not that the functional community does not like to use tools where necessary. Some people proposed that it would be good to have a software development analysis and design methodology for the functional paradigm (Wadler, 1998; Russell, 2001; Ryder and Thompson, 2005). One attempt can be found in the literature to develop a software design and analysis methodology for functional programs. Russell (Russell, 2001) in his PhD thesis showed a functional analysis and design (FAD) model which was intended to support the analysis and design of the functional programming paradigm. In reality, it is hardly ever used in the functional programming community (no discussion about it can be seen in the Haskell-Cafe (Haskell-Cafe, 2015) which is the comprehensive Haskell archive network) and no CASE implementation has been seen yet which supports this modelling language. This shows that the need to use an analysis and design tool for a functional paradigm is still uncommon.

Functional languages are declarative and they are so high level (because of the higher-order function, currying, immutable states etc.) that a programmer uses concepts that do not require a design and analysis methodology. Diagrams

are less useful for functional languages than OOP languages. Instead, many of these design diagrams can be expressed in types, or in signatures or using type classes in Haskell.

The development of HASKEU was not done using a formal analysis and design method because such a method does not exist, and the importance of using such an approach was not the scope of the thesis.

There is always debate/confusion in the programming community about the differences between programming paradigms, and the use of UML for functional programming can be seen by some people as a debatable issue. The UML diagrams are based on mutable states where there is no notion of state in a functional program. Hence, a functional program does not have any mutable objects. Relationships between immutable objects are of no interest, because such relationships are invalid. In functional programming, one function may call another. To design the overall system is not the most challenging aim, which is to implement the functions doing the calling. Because there are no side-effects, it is natural to divide a system into functions that may be developed independently. UML is not an appropriate notation to support this. The next section gives more explanation about appropriateness of using UML to support the analysis and design of functional programs.

A.2 Appropriateness of using UML

Design often involves diagrams, especially in the OOP paradigm, whereas functional programming hardly uses diagrams to show the design of a program. Because functional programming does not have a recognized design methodology, some attempts have been taken to make use of UML-for-OOP-like diagrams for functional programming. In Wakeling (Wakeling, 2001), it was shown how three types of UML diagrams could be used for functional programs and then how functional code in Haskell could be produced from those diagrams. These three types of diagram are use case diagrams, class diagrams and sequence diagrams. In a use case diagram, each of the different courses of action has an accompanying textual description. The intention of the diagram and the textual description is to make it obvious to all stakeholders what will be offered to the user and this should make it easier to agree. Using the use case diagrams as a basis, a number of classes having the required functionality can be produced. Once the classes have been decided, sequence diagrams can be drawn to show how they achieve the use cases. Among the three diagrams, only the use case diagrams can be created for functional programs without many restrictions. The other two i.e. class diagrams and sequence diagrams require the designer to adopt a functional style. Once the functional style has been adopted in drawing UML diagrams, Wakeling thinks the generated code would look imperative. This is because, apart from use case diagrams, the basic premise of UML diagrams rests on the notion of state which is the main restriction on drawing functional style diagrams in UML. There are

some other minor restrictions in class diagrams because functional programs do not have inheritance features and they only have the multiplicities 1..1, 0..1, and 0..* (as Haskell types can be either `a` or `Maybe a` or `[a]`). Although this restriction about multiplicities could be lifted by introducing new types (e.g., `data OneDotDotStar a = OneDotDotStar a [a]`). The restrictions in sequence diagrams are that a Haskell function should not have a free variable and that the local side-effects on the state of an object are disallowed.

Wakeling also mentioned that some other state-based diagrams (state chart diagram, activity diagram) are not very useful to design functional programs. Other diagrams (component diagram, deployment diagram) do not relate to a functional or other programming system development. A component diagram describes the relationships between the program components. A deployment diagram describes the relationships between components in a component diagram and the processors or the devices. A component diagram could be mapped to a script for a configuration management system, it is hard to see what more could be done with either diagram.

The other diagrams include a collaboration diagram which is another form of a sequence diagram, and an object diagram showing the current state of an object does not convey much meaning about the design of the system. Wakeling managed to produce Haskell code from UML diagrams, however he believed that the code produced looks imperative. In order to make it useful it is nec-

essary to be converted into a declarative form but this is a time-consuming process and it is not natural for a functional style of programming.

Another minor work by Marcin Szlenk (Szlenk, 2011) also tried to map UML to Haskell and only showed how to model class diagrams for functional programs. In his mapping, the data types and functions of a module need to be associated to each-other by creating different classes and this may produce strange module. Szlenk wanted to investigate broadening the scope of the Haskell included in the UML profile, but the researcher has found no further work published by Szlenk in this area.

Drawing upon the experience of Wakeling and Szlenk, the decision was made to investigate in more detail the appropriateness of using UML in functional languages. Wakeling attempted to draw functional style code from UML diagrams and as reported above, it can be seen that the generated functional code is actually not in a useful style. Again, as the basic premise of UML diagrams rests on the notion of state, the question may arise whether it is possible to remove that notion and sensibly use the diagrams. To answer this question the researcher implemented a simple example in a stateful way in Java and in a stateless way in Haskell. Then it was checked if a UML class or sequence diagram could be created in a meaningful way for the Haskell program from a reverse engineering perspective. The example is about a user login system. In Java, there were two classes - a `User` class that encapsulated the user name

and password attributes and a `UserLogIn` class that contained the list of users of the system and had a state for the currently logged in user. The `UserLogin` class also had three functions which were to retrieve the existing user list, to perform the login and to complete the registration.

```
public class User
{
    private String user_ID;
    private String password;
}

public class UserLogIn
{
    private ArrayList<User> userlist = new ArrayList<User>();
    private User currentUser = null;
    public void retrieveExistingUserList() {...}
    public boolean logIn (User u) {...}
    public boolean registration (User u) {...}
}
```

In Haskell, there was a new data type `User` which has one constructor with two fields for the user name and password. The `User` was defined in a module called `UserLogIn`, and the module has the same three functions as in the Java `UserLogIn` class.

```
module UserLogIn where
```

```
data User = User String String

retrieveExistingUserList :: IO [User]

logIn :: User -> [User] -> User

registration :: User -> IO [User]
```

The differences between stateless Haskell and stateful Java implementations of user login system are analysed below:

- (a) In Java, the `userList` and `currentUser` attributes in the `UserLogIn` class are the state of the all users and the current user in the system respectively. The `userList` can be changed in the `retrieveExistingUserList` and `registration` functions and the `currentUser` can be changed in the `logIn` function.

In Haskell, the current user and the user list are not member variables of the module. In fact, they are parameters of some functions where necessary and so these two fields cannot be shown in a class diagram. This actually means that for functional programs class diagrams will be without any member variables.

- (b) In Java, the `User` and `UserLogIn` class have to be in two different classes (there cannot be even one inner and one outer class) and hence their relationships are important in a class diagram. See Figure A.1.

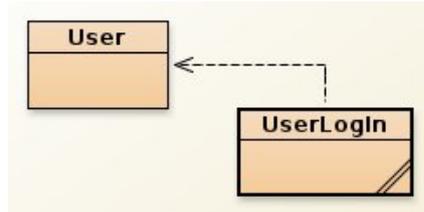


Figure A.1: Class diagram of user login system.

In Haskell, it is possible to have two modules (one for just declaring the data type `User` and one for the three functions) which looks odd from the modularization point of view, but in this way, a relationship diagram may be possible between the `User` and the `UserLogIn` module which is the same as in the Java class diagram shown above. If it was wanted just to have one class diagram for the one implemented module then declaring the `User` as an inner class would be a good idea, but it needs the designer to have functional programming knowledge.

- (c) In Java, methods can be distributed into classes by considering how they are changing and using the class level attributes. For example, the `retrieveExistingUserList` and `register` methods change the private `userList` attribute. Another example, the `login` method accesses the `userList` attribute to check the validity of a user and change the value of the `currentUser` attribute.

In Haskell, there is no such use of state and the `retrieveExistingUserList` and `register` functions return a completely new user list. A list of users is always passed as an argument to the `logIn` function and then the `logIn` function returns a completely new user. So, these three functions

can literally belong to anywhere in the program, as they are not sharing any state. However, having them in one class diagram can produce a one module structure containing the three functions. Nevertheless, it should be stated that in an encapsulated class diagram, there is no point in using the “hide state data” feature as this is not relevant for functional programming.

- (d) In Java, the sequence diagram for the user login action function must have the two functions in sequence a) `retrieveExistingUserList` (to update the `userList` attribute first) and b) `logIn` to check the user id and password in the retrieved `userList`. See Figure A.2.

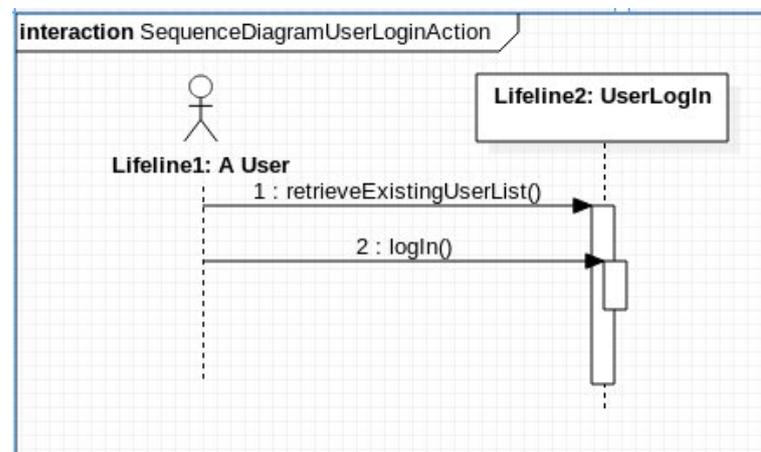


Figure A.2: Sequence diagram of user login action.

In Haskell, the `logIn` function needs to have a user list parameter (to be declarative) and any need to call this function will remind the developer to retrieve the user list first. The aim of the sequence diagram is to remind the developer to perform the actions in sequence is redundant for functional languages.

In summary, HASKEU was not developed using a formal analysis and design method because such methods are not deemed useful in the functional programming community, as discussed in Section A.1. Functional programmers have investigated the utility of using UML style diagrams to develop functional programs over the years. The conclusion has always been that UML is not suitable for functional programming development. HASKEU was not developed using UML notation but followed a standard functional programming development process which consists of dividing the problem into sub-functions and developing these independently.

Appendix B

Spaghetti Code

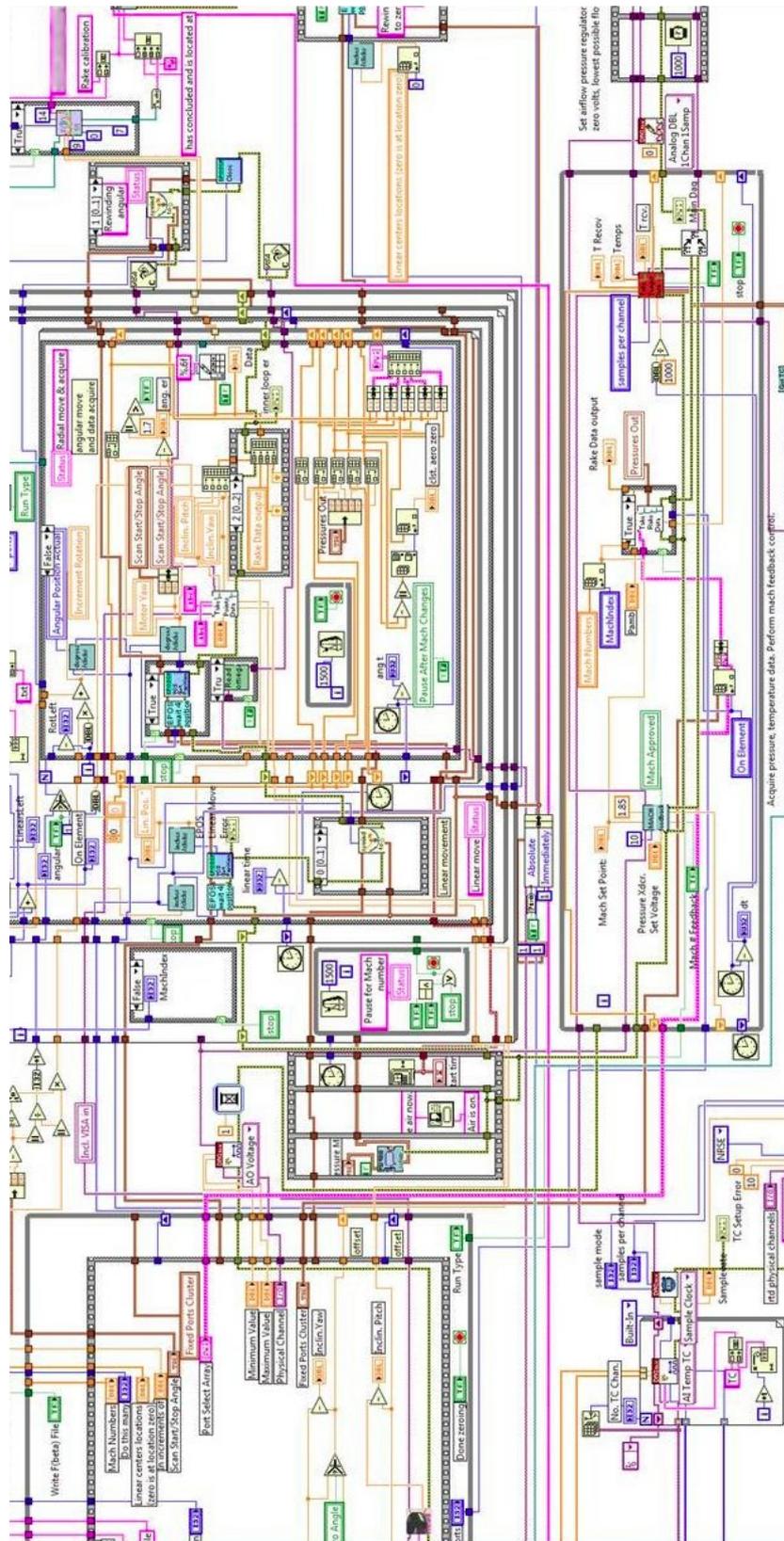


Figure B.1: Spaghetti Code in LabVIEW (Carr, 2011) .

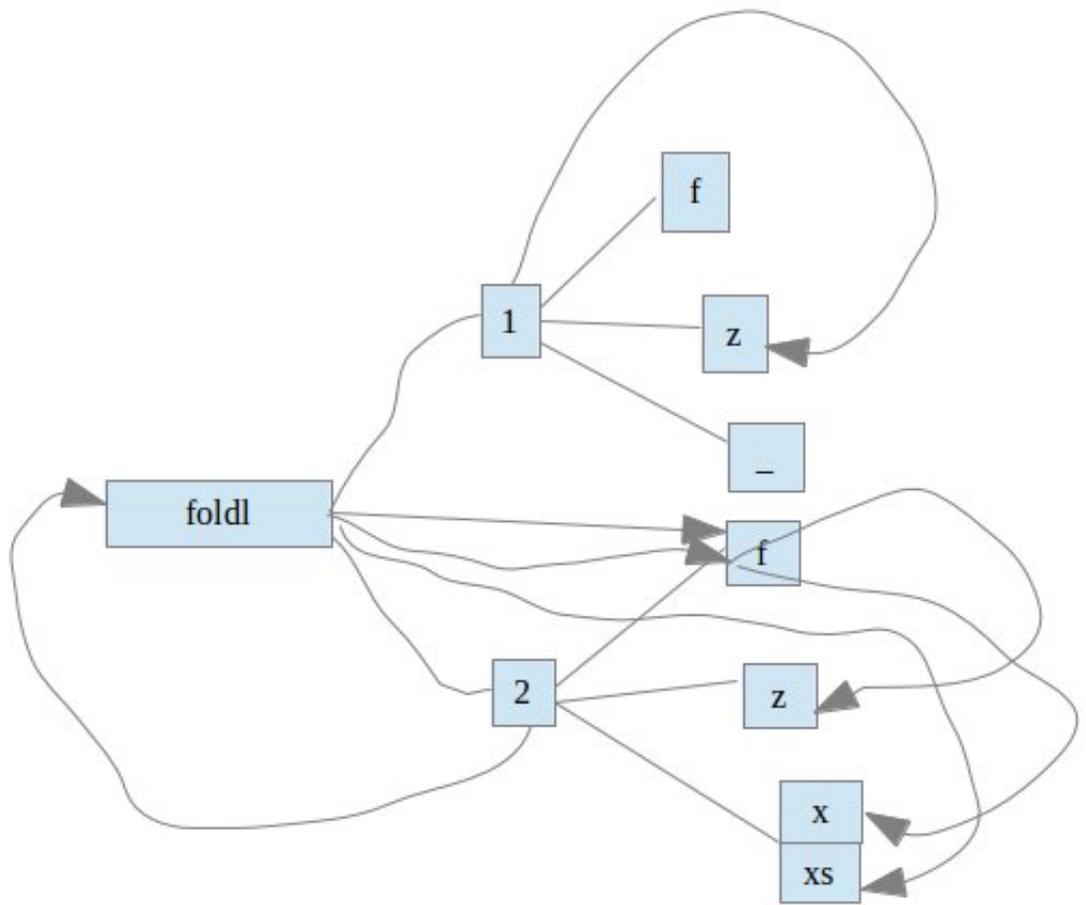


Figure B.2: A illustration of Spaghetti Code in early HASKEU.

Appendix C

User Manual

USER'S MANUAL

HASKEU

(HASKELL FOR END-USERS)

A PROGRAMMING SYSTEM FOR END-USER FUNCTIONAL PROGRAMMING

Contents

1	General Information	iii
1.1	System Overview	iii
1.2	Installation	iii
1.2.1	Starting HASKEU	iii
1.3	Exploring the Interface	iv
1.3.1	Global Toolbar	vi
1.3.2	Textual Programming Toolbar	vii
1.3.3	Display of insertion point position in the Textual Program Editor	viii
1.3.4	Textual Program Editing Area	ix
1.3.5	Textual Error List	x
1.3.6	Visual Programming “Select/Edit/Delete” Toolbar	xi
1.3.7	Visual Program Editing Area	xii
1.3.8	Display of the mouse cursor position in the Visual Program Editor	xiv
1.3.9	Visual Programming “Select/Edit/Delete” Toolbar	xv
1.3.10	Visual Error List	xix

2	Programming	xx
2.1	Textual Programming	xx
2.2	Visual Programming	xx
2.2.1	Understanding Different Item Icons	xxi
2.2.2	To Select an Item	xxv
2.2.3	To Add Annotation	xxv
2.2.4	To Delete an Item	xxv
2.2.5	To Add a New Function/ Local Function	xxvi
2.2.6	To Add a New Function Clause/ Local Function Clause	xxviii
2.2.7	To Add a New Parameter/ Local Parameter	xxx
2.2.8	To Add an Expression	xxxi
2.3	Understanding Visual Errors	xxxiii
2.4	Testing a Program	xxxvi
2.4.1	To Save a Program	xxxvi
2.4.2	To Compile and Test a Program	xxxvii

1 General Information

1.1 System Overview

This end-user functional programming system (HASKEU) supports both visual and textual programming, allowing for a smooth transition from one to the other as a user's programming expertise increases. The primary interface is a visual dataflow language consisting of boxes and arrows — a box representing a process and an arrow representing the dataflow between processes. The secondary interface is conventional textual language. Changes flow between the visual and textual interfaces, so that they are always consistent.

1.2 Installation

Currently full, in-person support is provided for the software installation including operating system (Linux) installation.

1.2.1 Starting HASKEU

1. Double-click the desktop icon 
2. The main screen appears, and the end-user functional programming

system (HASKEU) starts.

1.3 Exploring the Interface

The following Figure 1 shows the HASKEU user interface.

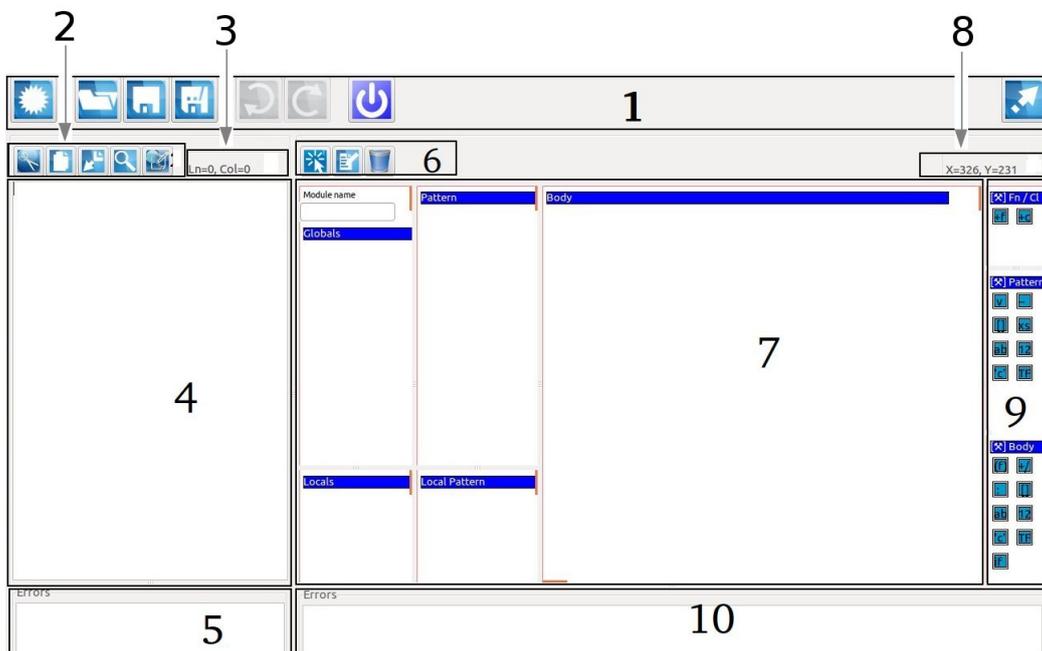


Figure 1: The HASKEU user interface.

1. Global Toolbar
2. Textual Programming Toolbar
3. Display of insertion point position in the Textual Program Editor
4. Textual Program Editing Area

-
5. Textual Error List
 6. Visual Programming “Select/Edit/Delete” Toolbar
 7. Visual Program Editing Area
 8. Display of mouse cursor position in the Visual Program Editor
 9. Visual Programming “Add new item” Toolbar
 10. Textual Error List

All the panes in the textual and visual program editing area are resizable and scrollable.

1.3.1 Global Toolbar

The global toolbar (see Figure 2) contains the menus which are common for both textual and visual programming.

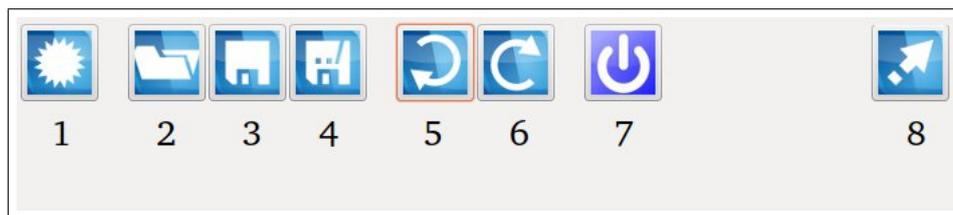


Figure 2: The global toolbar.

Table 1: The global toolbar menus

Menu No	Icon	Menu Name	Description
1		New Module	Create a new module
2		Open File	Open a file
3		Save	Save a file
4		Save as	Save as a file
5		Undo	Undo the last action
6		Redo	Redo the last undo action
7		Quit	Exit the program
8		New Window	Open a new window

1.3.2 Textual Programming Toolbar

The textual programming toolbar (see Figure 3) contains the menus to facilitate the textual programming system.



Figure 3: The textual programming toolbar.

Table 2: The textual programming toolbar menus

Menu No	Icon	Menu Name	Description
1		Cut	Deletes a selection of text to move it to another area
2		Copy	Copies a selection of text to duplicate it in another area, while keeping the original text
3		Paste	Places the cut/copied text in a new area
4		Find	Finds the next instance of any string of characters
5		Replace	Finds the next instance of any string of characters and replaces them with another specified string

1.3.3 Display of insertion point position in the Textual Program Editor

The label (shown as no “3” in Figure 1) shows the line number and column position of the insertion point in the textual program editor.

1.3.4 Textual Program Editing Area

The textual program editing area is where one can write functional programs textually. Figure 4 shows the textual program editing area with an example program included in it. This program editing area is resizable and scrollable. Any editing operations in the textual program updates its visual equivalent in the visual program editing area.

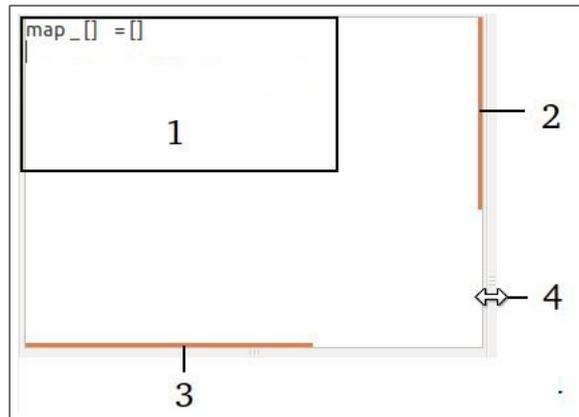


Figure 4: The textual program editing area.

1. The example textual program
2. Vertical scrollbar
3. Horizontal scrollbar
4. Mouse cursor changes appearance on the edges when resizing the editing area

1.3.5 Textual Error List

The listbox (shown as no “5” in Figure 1) shows all the syntactic and semantic error messages in the textual program. This error display area is resizable and scrollable. The following format is used in the textual error messages. The display of the subformats enclosed within the () brackets is optional.

[line no, column no] (Global function name and clause no) (Local function name and clause no) [Description of error]

1.3.6 Visual Programming “Select/Edit/Delete” Toolbar

The visual programming “Select/Edit/Delete” toolbar (see Figure 5) contains the menus to select, edit or delete an item in the visual programming system.



Figure 5: The visual programming “Select/Edit/Delete” toolbar.

Table 3: The visual programming “Select/Edit/Delete” toolbar menus

Menu No	Icon	Menu Name	Description
1		Select	Click on this button to enter selection mode and then click on any item in the visual program to select it
2		Edit	Click on this button to enter edit mode and then click on any item in the visual program to modify the annotation
3		Delete	Click on this button to delete a selected item in the visual program

1.3.7 Visual Program Editing Area

Figure 6 shows the organization of the visual program editing area, which is split into five panes, and each pane is resizable and scrollable. The visual program editing area is where one can write functional programs visually. Any editing operations in the visual program area updates its textual equivalent in the textual program editing area.

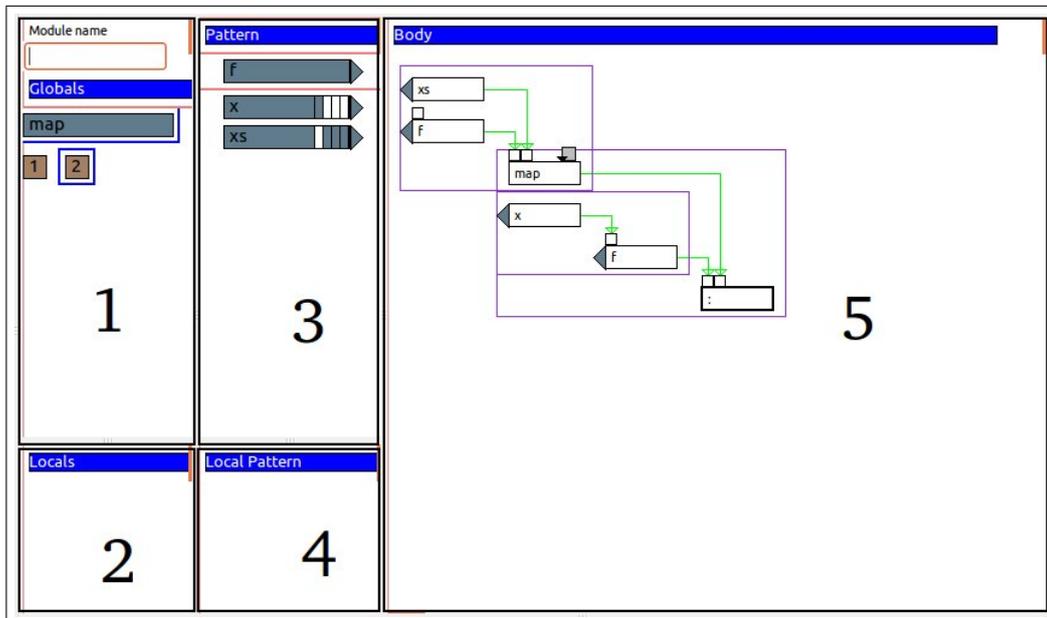


Figure 6: The visual program editing area.

1. The list of global functions and their clauses in a module
2. The list of local functions and their clauses of a selected global function clause

-
3. The pattern of a selected global function clause
 4. The pattern of a selected local function clause
 5. The body of a selected global or local function clause

1.3.8 Display of the mouse cursor position in the Visual Program Editor

The label (shown as no “8” in Figure 1) shows the coordinates of the mouse cursor position in a pane.

1.3.9 Visual Programming “Select/Edit/Delete” Toolbar

The visual programming “Add new item” Toolbar (see Figure 7) contains the menus to add new items in the panes of the visual programming system.

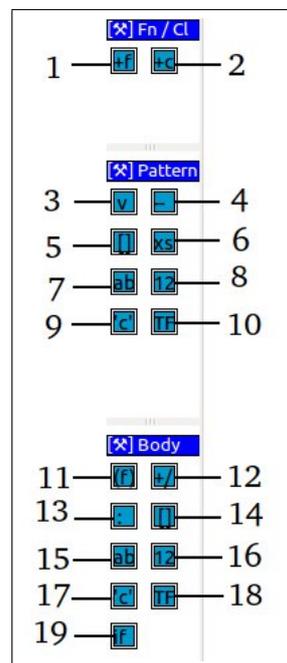


Figure 7: The visual programming “Add new item” toolbar.

Table 4: The visual programming “Add new item” toolbar menus

Menu No	Icon	Menu Name	Description
1		New function	Adds a new function on the global or local pane
2		New clause	Adds a new clause on the global or local pane
3		Variable	Adds a new variable on the pattern or local pattern pane
4		Wild-Card	Adds a new Wild-Card on the pattern or local pattern pane
5		Empty List	Adds a new Empty List on the pattern or local pattern pane
6		List (x:xs)	Adds a new List (x:xs) on the global or local pattern pane
7		String	Adds a new string on the global or local pattern pane
8		Int	Adds a new Int on the global or local pattern pane

Table 4: The visual programming “Add new item” toolbar menus

Menu No	Icon	Menu Name	Description
9		Char	Adds a new Char on the global or local pattern pane
10		Bool	Adds a new Bool on the global or local pattern pane
11		Function Application	Adds a new function application or parameter on the global or local function clause body
12		Operator	Adds a new operator on the global or local function clause body
13		List Constructor (:)	Adds a new list constructor (:) on the global or local function clause body
14		Empty List []	Adds a new empty list constructor on the global or local function clause body
15		String	Adds a new String on the global or local function clause body

Table 4: The visual programming “Add new item” toolbar menus

Menu No	Icon	Menu Name	Description
16		Int	Adds a new Int on the global or local function clause body
17		Char	Adds a Add a new Char on the global or local function clause body
18		Bool	Adds a new Bool on the global or local function clause body
19		If-Then-Else	Adds a new function application “cond” which is a replacement of “If-Then-Else” on the global or local function clause body

1.3.10 Visual Error List

The listbox (shown as no “10” in Figure 1) displays all the semantic error messages in the visual program. This error display area is resizable and scrollable. The following format is used in the visual error messages. The sub-formats enclosed within () first brackets are optional by displayed.

[Position in the screen] [Section] (Global function name and clause no)
(Local function name and clause no) [Description of error]

2 Programming

2.1 Textual Programming

The syntax of textual programming uses sequences of text and it describes a combination of regular expressions that form a syntactically correct program. The meaning given to a combination of what is handled by the semantics. Please refer to read textbooks on functional programming to learn textual syntax and semantics.

2.2 Visual Programming

The main benefits that can be gained by using this visual programming system are:

1. Less syntax reduces the error rates.
2. Help with language semantics is provided.
3. Syntactic and semantic errors are avoided before attempting compilation.
4. Faster learning and higher retention rates can be achieved.
5. Exploration is encouraged.

-
6. The programmer is always kept aware of the result by feedback being continually provided.
 7. The object of interest is immediately visible.

The following sections describe different actions to construct a program visually.

2.2.1 Understanding Different Item Icons

All items in this system are displayed as annotated icons. An item's view changes to indicate its selection or edit mode. Figure (see Figure 8) shows a function application *rev* in three different modes — unselected, selected and edit. A blue outlined rectangle indicates the selection mode, and annotation appears as an editable text field. A purple rectangle focuses on a group of items in the scope of an expression.

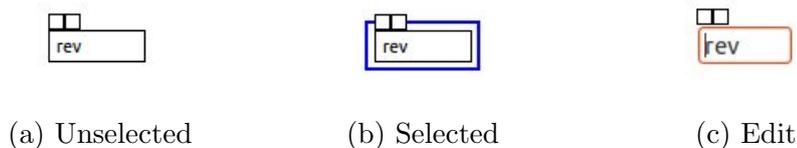


Figure 8: An item view in unselected, selected and editable modes.

The following three tables shows different item icons used to visualize items in different parts of a function:

Table 5: Icons for global or local function name and clauses

Icon	Description
	Function clauses are numbered in order underneath the function name

Table 6: Icons for items in patterns

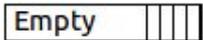
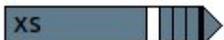
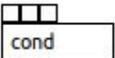
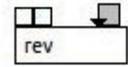
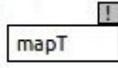
Icon	Description
	An Integer constant
	A Character constant
	A String constant
	A Boolean constant
	A variable. Parameter variables are shown with triangle at the right.
	An empty list
 	A non-empty list ($x : xs$)
	A wild-card

Table 7: Icons for items in the function body

Icon	Description
	An Integer constant
	A Character constant
	A String constant
	A Boolean constant
	A Function application <i>reverse</i> . Argument slots are drawn on the top-left corner of an item and aligned horizontally. The type information of individual argument with their description is shown when the mouse pointer is over that argument slot and the whole type information of an application, shown when the mouse pointer is over the application box, is displayed as tooltip text.
	if an item used in the function body is a parameter, then it is displayed with a triangle at the left.
	An Operator +

Table 7: Icons for items in the function body

Icon	Description
	The list constructor <code>[]</code> . A rectangle with a bold outline denotes one of the two list constructors.
	The list constructor <code>::</code> . A rectangle with a bold outline denotes one of the two list constructors.
	The <i>cond</i> function. This is a replacement of the if – then – else syntax. The first argument is the condition, the second argument is the result if the condition is true, and the third argument is the result if the condition is false.
	A recursive symbol. An arrowed arc is used on the top right-hand corner of the box for a recursive application.
	Undefined symbol. An unobtrusive “!” symbol is shown at the top-right corner of an undefined application.

2.2.2 To Select an Item

To select an item in the five panes of the visual program area, do the following:

1. Click the button 
2. Click on the item to make a selection.
3. A blue outlined rectangle indicates the item has been selected.

2.2.3 To Add Annotation

Any item in our visual system can be displayed with annotated icons.

1. Click the button 
2. Click on the item to edit its annotation.
3. Change the annotation by typing characters using the keyboard.

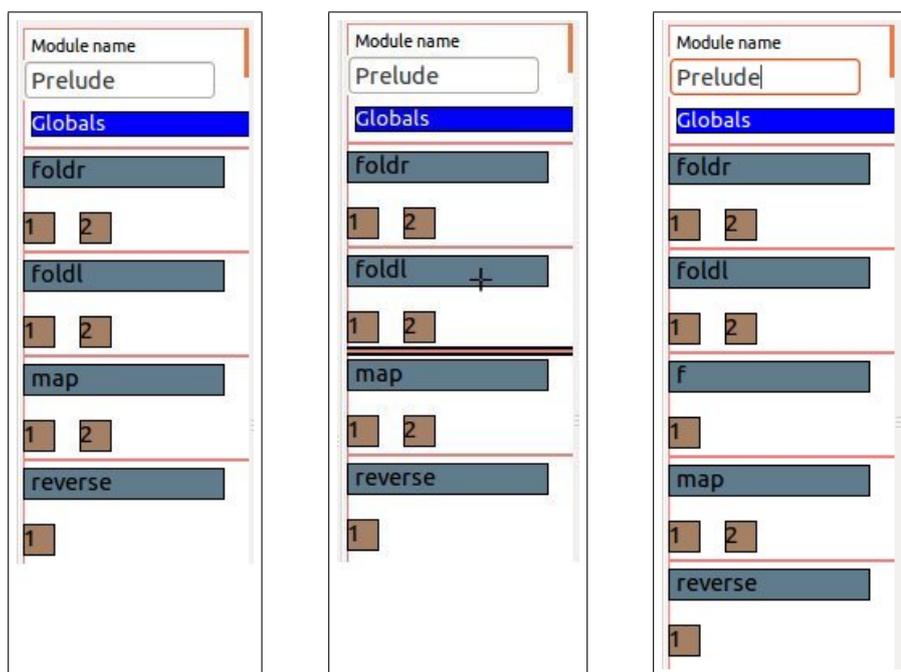
2.2.4 To Delete an Item

To delete an item, do the following:

1. Select an item to delete.
2. Click the button 
3. The selected item disappears.

2.2.5 To Add a New Function/ Local Function

1. Click the button .
2. Bring the mouse cursor to the global or local pane. The mouse cursor changes to a  icon.
3. When the user positions the mouse cursor over an existing function, a horizontal double line appears to indicate the position of the new function (see Figure 9b). If there is no existing function, click anywhere in the pane to insert a placeholder.
4. When the user clicks on the mouse, a placeholder is inserted with a default annotation “f” into the list as shown in Figure 9c.
5. Edit the default annotation to give the function a name.

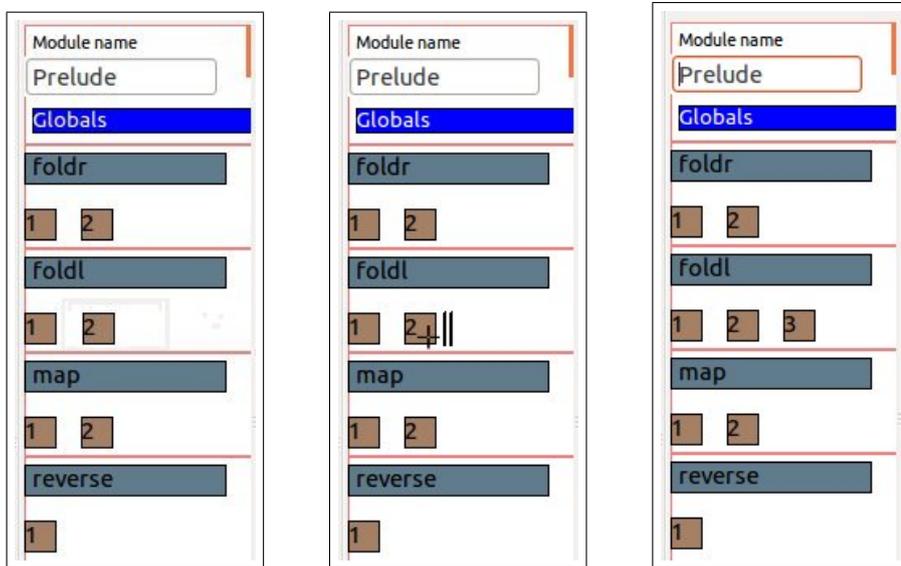


(a) List of functions (b) Selecting new position (c) New function inserted

Figure 9: Adding a new function.

2.2.6 To Add a New Function Clause/ Local Function Clause

1. Click the button 
2. Bring the mouse cursor to the global or local pane. The mouse cursor changes to a  icon.
3. When the user positions the mouse cursor over an existing clause, a vertical double line appears to indicate the position of the new clause (see Figure 10b).
4. When the user clicks on the mouse, a placeholder is inserted into the list as shown in Figure 10c. The clause number will be automatically given by the system.

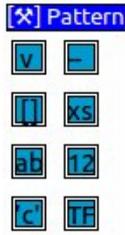


(a) List of clauses (b) Selecting new position (c) New clause inserted

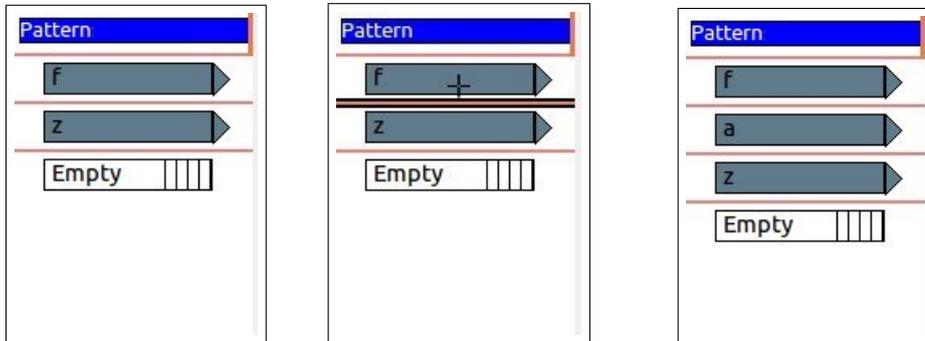
Figure 10: Adding a new clause.

2.2.7 To Add a New Parameter/ Local Parameter

1. Click any of the buttons below to add a specific parameter



2. Bring the mouse cursor to the pattern or local pattern pane. The mouse cursor changes to a \dagger icon.
3. When the user positions the mouse cursor over an existing parameter, a horizontal double line appears to indicate the position of the new parameter (see Figure 11b). If there is no existing parameter, click anywhere in the pane to insert a placeholder.
4. When the user clicks on the mouse, a placeholder is inserted into the list as shown in Figure 11c.
5. Edit the annotation to give the parameter a name.

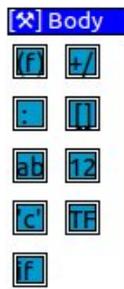


(a) List of parameters (b) Selecting new position (c) New parameter inserted

Figure 11: Adding a new parameter.

2.2.8 To Add an Expression

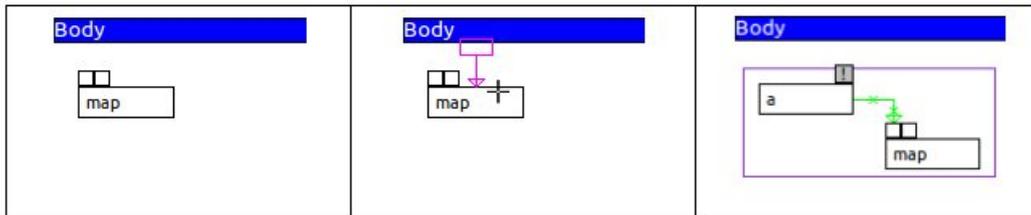
1. Click any of the buttons below to add a specific expression



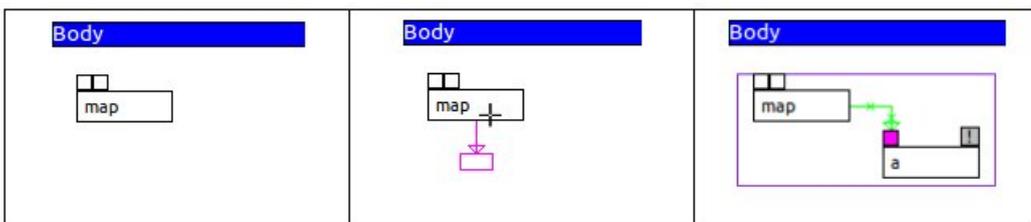
2. Bring the mouse cursor to the body pane. The mouse cursor changes to a \oplus icon.
3. A new argument can be added to an existing item and also an existing item can also be added as an argument to a new item. If the mouse cursor is positioned in the upper part of an item (in this example, *map*),

then a symbol indicating “add argument” appears (see Figure 12a), and if it is positioned in the lower part of an item, then a symbol indicating “add as argument” appears (see Figure 12b). No symbol appears in the illegal case of applying a constant to an argument. If there is no existing expression, click anywhere in the pane to insert a placeholder.

4. When the user clicks on the mouse, a placeholder is inserted into the function body. Figure 12 illustrates the procedure for adding an argument.



(a)



(b)

Figure 12: Adding an argument.

2.3 Understanding Visual Errors

An overview (a list) of all errors can be seen visually in the globals pane. Any function name and/or clause number with a cross mark against it indicates the existence of errors. Figure 13 denotes that the clause number “1” in function *foldl* and the clause number “2” in function *map* contain errors.

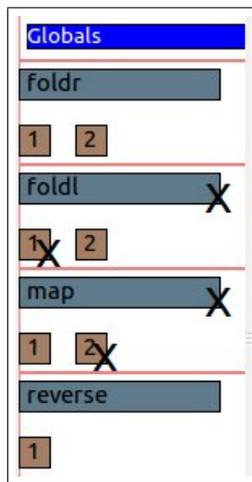


Figure 13: Overview of errors.

The visual error report can express error details in a single view (see Figure 14). Cross marks in the dataflow arc in the function body indicate type errors and both end-points contain the type information as tooltip text.

The function body shows all the type errors in the dataflow graph, not

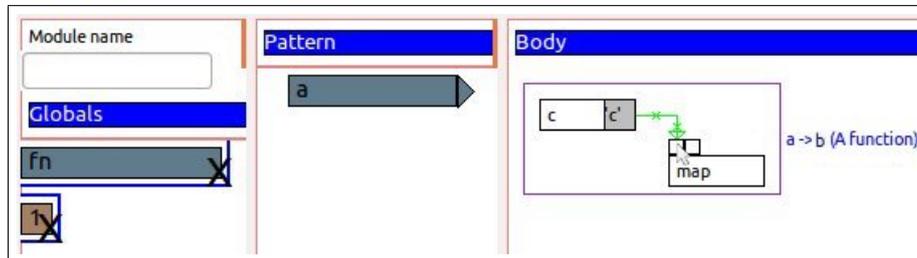


Figure 14: Error - type mismatch.

just the first one. Another small Haskell program (see Figure 15) is given below as an example: Here b is undefined, hence applying map to b is incorrect, and hence $(map\ b)$ cannot produce anything. Applying another map to this $(map\ b)$ is also incorrect. In functional programming, a previous error may be the cause of some later errors.

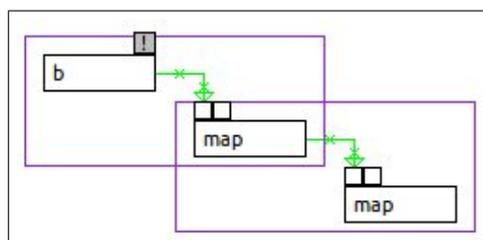


Figure 15: Showing all errors.

A type mismatch can be caused during unification. Unification of two types means that they are assumed to be of the same type. In the case of

a type mismatch during unification, it is hard for the type checker to tell which wrong parameter makes the other parameters wrong, only the user would know. The following program (see Figure 16) highlights part of the problem. Here, *map* uses the same parameter *a* in its two arguments where one is correct and the other is not. From the visual view of this function, the user can see all the uses of *a* and how many of them have been used incorrectly.

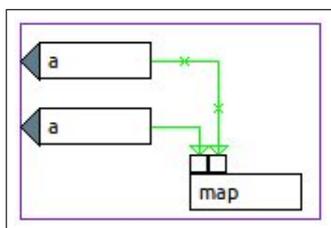


Figure 16: Error - unification.

An unobtrusive “!” symbol is shown on the top-right corner of an undefined application and also a tooltip text “Function not defined” is shown (see Figure 17).

Any unused argument slot is shown in the colour magenta (see Figure 18), so that the user will know an unnecessary argument has been used.

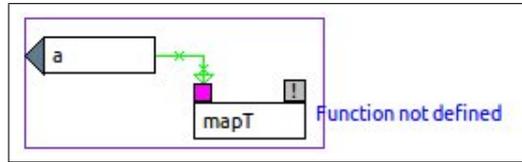


Figure 17: Error - undefined function.

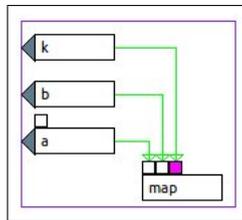


Figure 18: Error - unused argument.

2.4 Testing a Program

2.4.1 To Save a Program

You must save your program if you want to quit the program without losing your work, and if you want to test a program. When you save the program, it is stored as a file on your computer. Later, you can open the file, and change it.

1. Click the Save button in the Global toolbar.

-
2. Specify the location where you want to save the document in the Save in box. Type a file name in the File name box.
 3. Click Save.
 4. The document is saved as a file with an extension “.hs”. The file name in the Title Bar changes to reflect the saved file name.

2.4.2 To Compile and Test a Program

1. Press (Ctrl+Alt+T) on your keyboard to open a terminal.
2. Use the command

```
cd folderLocation
```

Here “folderLocation” is the location of the folder where the “.hs” file has been saved.
3. Type in the command

```
ghci fileName.hs
```

Here “fileName” is the name of the saved file. If it shows some compile time errors, then go back to the program and try to fix the errors.
4. If there are no compile time errors, then type in the command

```
functionName arguments
```

Here “functionName” is the name of the function to be tested, and

“arguments” are the argument values to test the function.

5. Check the results.

Appendix D

Usability Test - Questionnaire and Consent Form

D.1 Usability Test - Questionnaire

Questions about the usability of HASKEU

- 1) Which system did you find easier to write a program?
 - a) Textual
 - b) Visual

- 2) Did the system allow a easy transition from visual to textual and vice-versa?
 - a) Yes
 - b) NoIf not, please specify why

- 3) Were the icons used in the visual programming easier to understand, and could they be easily remembered for use next time?
 - a) Yes
 - b) NoIf not, please specify why

- 4) Did you find the dataflow of the program more understandable when it was shown visually or textually?
 - a) Textually
 - b) Visually

- 5) Which way of showing error messages was easier to understand and then fix?
 - a) Textual
 - b) Visual

- 6) Which way of showing error messages was easier to understand and then fix?
 - a) Yes
 - b) NoIf not, please specify why

Questions about the usability of HASKEU

7) Did you find it a good thing that the system did not allow you to make syntax error?

a) Yes

b) No

If not, please specify why

8) Did you find any inconsistencies between the two systems?

a) Yes

b) No

If not, please specify where

9) Please write your comments about the system and specify where the system could be improved?

D.2 Usability Test - Consent Form

HASKEU Usability Test

Participant Consent Form

This usability study will evaluate the end-user functional programming system. We would like to see how participants can complete some tasks using this system. The aim is not to evaluate your ability, but this testing will evaluate the system to provide information on how it can be improved.

During this test, participants will be asked to do some tasks using the system and then they will be asked to fill out a questionnaire. The testing session will last no longer than three hours.

If you feel uncomfortable during this testing session or if you do not want to finish a task, then simply move on the next task. Also, you can leave at any time if you want to.

It is hoped that about 5 people will be involved in this usability test. The results will be included in a report. No names of participants will be included in this report and no identification detail will be associated with any data.

I,,
have read and fully understand the extent of the study and I agree to take part in this user testing session. I have been given a blank copy of this consent form for my records.

Signed Date

Appendix E

The Library API - MVC_WX

The Library API of the MVC_WX Module in HASKEU Implementation

```
-----  
--  
-- Module      : MVC_WX.lhs  
-- Author      : Abu Alam  
--  
-- Maintainer  : Abu Alam, s0408730@connect.glos.ac.uk  
--  
-- Purpose     : Utility functions for creating wxHaskell events and attributes  
--               and adjusting them for the reactive.banana library.  
-----  
  
module MVC_WX where  
  
import qualified Graphics.UI.WX as WX  
import Graphics.UI.WX hiding (Event, Attr)  
import Graphics.UI.WXCore hiding (View, Event)  
import Reactive.Banana.WX  
import Reactive.Banana  
import FindReplaceUtil  
  
-- Wx Widget Events to Wx Banana Events  
  
-- Event occurs when user clicks a button  
evButtonCommand      :: (Frameworks t, Reactive w, Commanding w)  
                    => w -> Moment t (Event t ())  
  
evButtonCommand w = do  
    eCommand <- event0 w command  
    return (eCommand)
```

```

-- Event occurs when user opens a file open dialog box

evButtonCommandFileOpen      :: Frameworks t
                               => BitmapButton ()
                               -> Frame ()
                               -> Moment t (Event t (Maybe FilePath, String))

evButtonCommandFileOpen b w = do
    addHandler <- liftIONow $ event1ToAddHandler b (event0ToEvent1 command)
    fromAddHandler
        $ mapIO (const $ openPage w) addHandler

-- open a file open dialog box

openPage :: Frame () -> IO (Maybe FilePath, String)

openPage win =
    do
        maybePath <- fileOpenDialog    win True True
                                "Open file..." [("Haskells (*.hs)",["*.hs"]),
                                ("Texts (*.txt)", ["*.txt"]),
                                ("Any file (*.*)",["*.*."]) " " " "

        case maybePath of
            Nothing -> return (maybePath, "")
            Just path -> do
                fileContents <- readFile path
                return (maybePath, fileContents)

-- Event occurs when user opens a file save dialog box

evButtonCommandFileSave      :: Frameworks t
                               => BitmapButton ()
                               -> Frame ()
                               -> TextCtrl()
                               -> Moment t (Event t (Maybe FilePath))

```

```

evButtonCommandFileSave b w t = do

    addHandler <- liftIONow $ event1ToAddHandler b (event0ToEvent1 command)

    fromAddHandler

        $ mapIO (const $ savePage w t) addHandler

-- Open a file save dialog box

savePage :: Frame () -> TextCtrl() -> IO (Maybe FilePath)

savePage w t = do

    winTitle <- get w text

    --infoDialog w winTitle winTitle

    case (winTitle==windowTitle) of

        True -> savePageAs w t

        False->

            do

                let path = drop (length windowTitle + 3) winTitle

                    textCtrlSaveFile t path

                return (Just path)

-- Event occurs when user opens a file save as dialog box

evButtonCommandFileSaveAs    :: Frameworks t

                                => BitmapButton ()

                                -> Frame ()

                                -> TextCtrl()

                                -> Moment t (Event t (Maybe FilePath))

evButtonCommandFileSaveAs b w t = do

    addHandler <- liftIONow $ event1ToAddHandler b (event0ToEvent1 command)

    fromAddHandler

        $ mapIO (const $ savePageAs w t) addHandler

```

```

-- Open a file save as dialog box

savePageAs :: Frame () -> TextCtrl () -> IO (Maybe FilePath)

savePageAs win txtEditor =

    do

        maybePath <- fileSaveDialog    win True True

                                     "Save file..." [("Haskells (*.hs)",["*.hs"]),

                                     ("Texts (*.txt)", ["*.txt"]),

                                     ("Any file (*.*)",["*..*"])] "" ""

        case maybePath of

            Nothing -> return Nothing

            Just path ->

                do

                    textCtrlSaveFile txtEditor path

                    return maybePath

-- Event occurs when user uses cut in a text control

evButtonCommandCut :: Frameworks t =>

    BitmapButton () -> TextCtrl() -> Moment t (Event t ())

evButtonCommandCut b t = do

    addHandler <- liftIONow $ event1ToAddHandler b (event0ToEvent1 command)

    fromAddHandler

        $ mapIO (const $ textCtrlCut t) addHandler

-- Event occurs when user uses copy in a text control

evButtonCommandCopy :: Frameworks t =>

    BitmapButton () -> TextCtrl() -> Moment t (Event t ())

evButtonCommandCopy b t = do

    addHandler <- liftIONow $ event1ToAddHandler b (event0ToEvent1 command)

    fromAddHandler

        $ mapIO (const $ textCtrlCopy t) addHandler

```

```

-- Event occurs when user uses paste in a text control

evButtonCommandPaste :: Frameworks t =>

    BitmapButton () -> TextCtrl() -> Moment t (Event t ())

evButtonCommandPaste b t = do

    addHandler <- liftIONow $ event1ToAddHandler b (event0ToEvent1 command)

    fromAddHandler

        $ mapIO (const $ textCtrlPaste t) addHandler

-- Event occurs when user opens a find dialog box

evButtonCommandFind    :: Frameworks t

                        => BitmapButton ()

                        -> Frame ()

                        -> TextCtrl()

                        -> FindReplaceData ()

                        -> Moment t (Event t ())

evButtonCommandFind b w t fr = do

    addHandler <- liftIONow $ event1ToAddHandler b (event0ToEvent1 command)

    let guiCtx = GUICtx w t fr

    fromAddHandler

        $ mapIO (const $ justFind guiCtx) addHandler

-- Event occurs when user opens a replace dialog box

evButtonCommandReplace    :: Frameworks t

                            => BitmapButton ()

                            -> Frame ()

                            -> TextCtrl()

                            -> FindReplaceData ()

                            -> Moment t (Event t ())

evButtonCommandReplace b w t fr = do

    addHandler <- liftIONow $ event1ToAddHandler b (event0ToEvent1 command)

```

```

let guiCtx = GUICtx w t fr

fromAddHandler

    $ mapIO (const $ findReplace guiCtx) addHandler

-- Event occurs when user changes text in a text control
evTextChanged :: Frameworks t =>
    TextCtrl w -> Moment t (Event t (String, Int))
    -- Text Editor String, Insertion Point, Textua Errs, Gra Errs
evTextChanged w = do
    addHandler <- liftIONow $ event1ToAddHandler w (event0ToEvent1 onText)
    fromAddHandler
        $ filterAddHandler (const $ textCtrlIsModified w)
        $ mapIO (const $ (liftA2 (,)) (get w text) (get w insertionPoint) ) addHandler

-- Event occurs when user releases a key in a text control
evTextKBUUp :: Frameworks t =>
    TextCtrl w -> Moment t (Event t (String, Int))
    -- Text Editor String, Insertion Point, Textua Errs, Gra Errs
evTextKBUUp txt = do
    eKeyboardUp <- event1 txt keyboardUp
    bText <- (behavior txt text)
    bInsertionPoint <- (behavior txt insertionPoint)
    let eTextInsPt = (uncurry (liftA2 (,))
                    (bText, bInsertionPoint)) <@ eKeyboardUp
    return eTextInsPt

-- Event occurs when user selects an item in the choice box
evSelChoice :: Frameworks t =>
    Choice () -> Moment t (Event t Int)
evSelChoice w = do

```

```

    addHandler <- liftIONow $ event1ToAddHandler w (event0ToEvent1 select)

    fromAddHandler $ mapIO (const $ get w selection) addHandler

-- Event occurs when user selects an item in the list box

evSelListBox :: Frameworks t => -- From wx Banana

    SingleListBox b -> Moment t (Event t Int)

evSelListBox w = do

    liftIONow $ fixSelectionEvent w

    addHandler <- liftIONow $ event1ToAddHandler w (event0ToEvent1 select)

    fromAddHandler $ mapIO (const $ get w selection) addHandler

-- Fix @select@ event not being fired

-- when items are *un*selected. -- From wx Banana

fixSelectionEvent listbox =

    set listbox [ on unclick := handler ]

    where

    handler _ = do

        propagateEvent

        s <- get listbox selection

        when (s == -1) $ (get listbox (on select)) >>= id

-- Event occurs when user moves the mouse

evMouseMove :: (Frameworks t, Reactive w)

    => w

    -> Moment t (Event t EventMouse)

evMouseMove w = do

    eMouse <- event1 w mouse

    let eMouseMove = filterE (\event -> (isMouseMove event)) $ eMouse

    return (eMouseMove)

evMouseLeftUp :: (Frameworks t, Reactive w)

```

```

=> w

-> Moment t (Event t EventMouse)

evMouseLeftUp w = do

  eMouse <- event1 w mouse

  let eMouseLeftUp = filterE (\event -> (isMouseLeftUp event)) $ eMouse

  return (eMouseLeftUp)

-- Event occurs when user releases mouse left button

evMouseLeftDown      :: (Frameworks t, Reactive w)

=> w

-> Moment t (Event t EventMouse)

evMouseLeftDown w = do

  eMouse <- event1 w mouse

  let eMouseLeftDown = filterE (\event -> (isMouseLeftDown event)) $ eMouse

  return (eMouseLeftDown)

-- Event occurs when user depresses the enter key

evReturnKeyPressed    :: (Frameworks t, Reactive w)

=> w

-> Moment t (Event t EventKey)

evReturnKeyPressed w = do

  eKB <- event1 w keyboard

  let eKBReturn = filterE (\event -> isKBEnterPressed event) $ eKB

  return (eKBReturn)

-- Event occurs when user clicks on a slider

evSliderCommand :: Frameworks t => Slider () -> Moment t (Event t Int)

evSliderCommand w = do

  addHandler <- liftIONow $ event1ToAddHandler w (event0ToEvent1 command)

  fromAddHandler

  $ mapIO (const $ get w selection) addHandler

```

```

-- Fix scrolled window @repaint@ event not being fired when Model change
evTxtChgRepaintSw :: Frameworks t =>
    TextCtrl w -> [ScrolledWindow ()] -> Moment t (Event t ())
evTxtChgRepaintSw t lstSw = do
    addHandler <- liftIONow $ event1ToAddHandler t (event0ToEvent1 onText)
    fromAddHandler
        $ filterAddHandler (const $ textCtrlIsModified t)
        $ mapIO (const $ (swRepaints lstSw)) addHandler

evBtnCommRepaintSw :: Frameworks t =>
    BitmapButton () -> [ScrolledWindow ()] -> Moment t (Event t ())
evBtnCommRepaintSw b lstSw = do
    addHandler <- liftIONow $ event1ToAddHandler b (event0ToEvent1 command)
    fromAddHandler
        $ mapIO (const $ (swRepaints lstSw)) addHandler

evSwMouseRepaintSwAll :: Frameworks t =>
    ScrolledWindow () -> [ScrolledWindow ()] -> Moment t (Event t ())
evSwMouseRepaintSwAll w lstSw = do
    addHandler <- liftIONow $ event1ToAddHandler w mouse
    fromAddHandler
        -- $ mapIO (const $ (swRepaint w)) addHandler
        $ mapIO (const $ (swRepaints lstSw)) addHandler -- we may need this

swRepaints lstW =
    sequence_ (map swRepaint lstW)

swRepaint w =
    do
    propagateEvent
    repaint w

```

```

-- Event occurs when user scrolls a scrolled window

evSwScroll :: Frameworks t =>

    ScrolledWindow () -> Moment t (Event t EventScroll)

evSwScroll sw = do

    eScroll <- event1 sw windowScroll

    return (eScroll)

-- new wx attributes/ Events

onText :: WX.Event (Control a) (IO ())

onText = WX.newEvent "onText" controlGetOnText controlOnText

windowScroll :: WX.Event (Window a) (EventScroll -> IO ())

windowScroll = WX.newEvent    "windowScroll"

                                windowGetOnScroll windowOnScroll

keyboardUp :: WX.Event (Window a) (EventKey -> IO ())

keyboardUp = WX.newEvent    "keyboardUp"

                                windowGetOnKeyUp (windowOnKeyUp)

keyboardDown :: WX.Event (Window a) (EventKey -> IO ())

keyboardDown = WX.newEvent    "keyboardDown"

                                windowGetOnKeyDown (windowOnKeyDown)

insertionPoint :: WX.Attr (TextCtrl a) Int

insertionPoint =    newAttr "insertionPoint"

                    textCtrlGetInsertionPoint (textCtrlSetInsertionPoint)

mouseCursor :: WX.Attr (Window a) (Cursor ())

mouseCursor =    newAttr "mouseCursor" windowGetCursor

                (\w c -> do    windowSetCursor w c

                                return ())

```

```

-- Event Filters

isKbEnterPressed :: EventKey -> Bool

isKbEnterPressed (EventKey KeyReturn _ _) = True

isKbEnterPressed _ = False

isMouseLeftUp :: EventMouse -> Bool

isMouseLeftUp (MouseLeftUp _ _) = True

isMouseLeftUp _ = False

isMouseLeftDown :: EventMouse -> Bool

isMouseLeftDown (MouseLeftDown _ _) = True

isMouseLeftDown _ = False

isMouseLeftDrag :: EventMouse -> Bool

isMouseLeftDrag (MouseLeftDrag _ _) = True

isMouseLeftDrag _ = False

isMouseMove :: EventMouse -> Bool

isMouseMove (MouseMove _ _) = True

isMouseMove _ = False

```

Appendix F

Source Code of Functor, Applicative and Monad

This Appendix shows the full list of all functions of Haskell's `Functor`, `Applicative` and `Monad` classes and their default instances. The source code is taken from the Haskell website (Haskell Website - Base, n.d.).

```

-----
-- |
-- Module      :  GHC.Base
-- Copyright   :  (c) The University of Glasgow, 1992-2002
-- License     :  see libraries/base/LICENSE
--
-- Maintainer  :  cvs-ghc@haskell.org
-- Stability   :  internal
-- Portability :  non-portable (GHC extensions)
--
-- Basic data types and classes.
--
-----

#include "MachDeps.h"

module GHC.Base
  (
    module GHC.Base,
    module GHC.Classes,
    module GHC.CString,
    module GHC.Magic,
    module GHC.Types,
    module GHC.Prim,
    module GHC.Err
  )
  where

import GHC.Types
import GHC.Classes

```

```

import GHC.CString

import GHC.Magic

import GHC.Prim

import GHC.Err

import GHC.IO (failIO)

import GHC.Tuple ()

import GHC.Integer ()

infixr 9  .

infixr 5  ++

infixl 4  <$

infixl 1  >>, >>=

infixr 1  =<<

infixr 0  $, $!

infixl 4 <*>, <*, *>, <***>

default ()

#if 0

data Bool = False | True

data Ordering = LT | EQ | GT

data Char = C# Char#

type String = [Char]

data Int = I# Int#

data () = ()

data [] a = MkNil

not True = False

(&&) True True = True

otherwise = True

```

```

build = error "urk"

foldr = error "urk"

#endif

data Maybe a = Nothing | Just a
    deriving (Eq, Ord)

class Monoid a where

    mempty  :: a

    mappend :: a -> a -> a

    mconcat :: [a] -> a

    mconcat = foldr mappend mempty

instance Monoid [a] where

    mempty  = []

    mappend = (++)

    mconcat xss = [x | xs <- xss, x <- xs]

instance Monoid b => Monoid (a -> b) where

    mempty _ = mempty

    mappend f g x = f x 'mappend' g x

instance Monoid () where

    mempty      = ()

    _ 'mappend' _ = ()

    mconcat _   = ()

instance (Monoid a, Monoid b) => Monoid (a,b) where

    mempty = (mempty, mempty)

    (a1,b1) 'mappend' (a2,b2) =

    (a1 'mappend' a2, b1 'mappend' b2)

```

```

instance (Monoid a, Monoid b, Monoid c) => Monoid (a,b,c) where
    mempty = (mempty, mempty, mempty)
    (a1,b1,c1) 'mappend' (a2,b2,c2) =
    (a1 'mappend' a2, b1 'mappend' b2, c1 'mappend' c2)

instance (Monoid a, Monoid b, Monoid c, Monoid d) => Monoid (a,b,c,d) where
    mempty = (mempty, mempty, mempty, mempty)
    (a1,b1,c1,d1) 'mappend' (a2,b2,c2,d2) =
    (a1 'mappend' a2, b1 'mappend' b2,
    c1 'mappend' c2, d1 'mappend' d2)

instance (Monoid a, Monoid b, Monoid c, Monoid d, Monoid e) =>
    Monoid (a,b,c,d,e) where
    mempty = (mempty, mempty, mempty, mempty, mempty)
    (a1,b1,c1,d1,e1) 'mappend' (a2,b2,c2,d2,e2) =
    (a1 'mappend' a2, b1 'mappend' b2, c1 'mappend' c2,
    d1 'mappend' d2, e1 'mappend' e2)

instance Monoid Ordering where
    mempty      = EQ
    LT 'mappend' _ = LT
    EQ 'mappend' y = y
    GT 'mappend' _ = GT

instance Monoid a => Monoid (Maybe a) where
    mempty = Nothing
    Nothing 'mappend' m = m
    m 'mappend' Nothing = m
    Just m1 'mappend' Just m2 = Just (m1 'mappend' m2)

instance Monoid a => Applicative ((,) a) where

```

```

    pure x = (mempty, x)

    (u, f) <*> (v, x) = (u 'mappend' v, f x)

class Functor f where

    fmap      :: (a -> b) -> f a -> f b

    (<$)      :: a -> f b -> f a

    (<$)      = fmap . const

class Functor f => Applicative f where

    pure :: a -> f a

    (<*>) :: f (a -> b) -> f a -> f b

    (*>) :: f a -> f b -> f b

    a1 *> a2 = (id <$ a1) <*> a2

    (<*) :: f a -> f b -> f a

    (<*) = liftA2 const

    (<*>*) :: Applicative f => f a -> f (a -> b) -> f b

    (<*>*) = liftA2 (flip ($))

    liftA :: Applicative f => (a -> b) -> f a -> f b

    liftA f a = pure f <*> a

    liftA2 :: Applicative f => (a -> b -> c) -> f a -> f b -> f c

    liftA2 f a b = fmap f a <*> b

    liftA3 :: Applicative f => (a -> b -> c -> d) -> f a -> f b -> f c -> f d

    liftA3 f a b c = fmap f a <*> b <*> c

    join      :: (Monad m) => m (m a) -> m a

    join x    = x >>= id

class Applicative m => Monad m where

    (>>=)    :: forall a b. m a -> (a -> m b) -> m b

```

```

(>>)      :: forall a b. m a -> m b -> m b

m >> k = m >>= \_ -> k

return     :: a -> m a

return     = pure

fail       :: String -> m a

fail s     = error s

(=<<)      :: Monad m => (a -> m b) -> m a -> m b

f =<< x     = x >>= f

when       :: (Applicative f) => Bool -> f () -> f ()

when p s   = if p then s else pure ()

sequence  :: Monad m => [m a] -> m [a]

sequence = mapM id

mapM      :: Monad m => (a -> m b) -> [a] -> m [b]

mapM f as = foldr k (return []) as

           where

           k a r = do { x <- f a; xs <- r; return (x:xs) }

liftM     :: (Monad m) => (a1 -> r) -> m a1 -> m r

liftM f m1 = do { x1 <- m1; return (f x1) }

liftM2    :: (Monad m) => (a1 -> a2 -> r) -> m a1 -> m a2 -> m r

liftM2 f m1 m2 = do { x1 <- m1; x2 <- m2; return (f x1 x2) }

liftM3    :: (Monad m) => (a1 -> a2 -> a3 -> r) -> m a1 -> m a2 -> m a3 -> m r

liftM3 f m1 m2 m3 = do { x1 <- m1; x2 <- m2; x3 <- m3; return (f x1 x2 x3) }

liftM4    :: (Monad m) => (a1 -> a2 -> a3 -> a4 -> r)

           -> m a1 -> m a2 -> m a3 -> m a4 -> m r

liftM4 f m1 m2 m3 m4 = do { x1 <- m1;

```

```

        x2 <- m2;

        x3 <- m3;

        x4 <- m4;

        return (f x1 x2 x3 x4)
    }

liftM5 :: (Monad m) => (a1 -> a2 -> a3 -> a4 -> a5 -> r)
      -> m a1 -> m a2 -> m a3 -> m a4 -> m a5 -> m r

liftM5 f m1 m2 m3 m4 m5 = do {
    x1 <- m1;

    x2 <- m2;

    x3 <- m3;

    x4 <- m4;

    x5 <- m5;

    return (f x1 x2 x3 x4 x5)
}

ap      :: (Monad m) => m (a -> b) -> m a -> m b
ap m1 m2      = do { x1 <- m1; x2 <- m2; return (x1 x2) }

instance Functor ((->) r) where
    fmap = (.)

instance Applicative ((->) a) where
    pure = const
    (<*>) f g x = f x (g x)

instance Monad ((->) r) where
    return = const
    f >>= k = \ r -> k (f r) r

instance Functor ((,) a) where
    fmap f (x,y) = (x, f y)

```

```

instance Functor Maybe where

    fmap _ Nothing      = Nothing

    fmap f (Just a)    = Just (f a)

instance Applicative Maybe where

    pure = Just

    Just f <*> m      = fmap f m

    Nothing <*> _m     = Nothing

    Just _m1 *> m2    = m2

    Nothing *> _m2    = Nothing

instance Monad Maybe where

    (Just x) >>= k     = k x

    Nothing >>= _      = Nothing

    (>>) = (>*)

    return              = Just

    fail _              = Nothing

infixl 3 <|>

class Applicative f => Alternative f where

    empty :: f a

    (<|>) :: f a -> f a -> f a

    some :: f a -> f [a]

    some v = some_v

    where

        many_v = some_v <|> pure []

        some_v = (fmap (:) v) <*> many_v

    many :: f a -> f [a]

    many v = many_v

    where

```

```

many_v = some_v <|> pure []

some_v = (fmap (:) v) <*> many_v

instance Alternative Maybe where

    empty = Nothing

    Nothing <|> r = r

    1 <|> _ = 1

class (Alternative m, Monad m) => MonadPlus m where

    mzero :: m a

    mzero = empty

    mplus :: m a -> m a -> m a

    mplus = (<|>)

instance MonadPlus Maybe

instance Functor [] where

    fmap = map

instance Applicative [] where

    pure x = [x]

    fs <*> xs = [f x | f <- fs, x <- xs]

    xs *> ys = [y | _ <- xs, y <- ys]

instance Monad [] where

    xs >>= f = [y | x <- xs, y <- f x]

    (>>) = (>>)

    return x = [x]

    fail _ = []

instance Alternative [] where

    empty = []

    (<|>) = (++)

```

```

instance MonadPlus []

foldr      :: (a -> b -> b) -> b -> [a] -> b

foldr k z = go

      where

          go []      = z

          go (y:ys) = y 'k' go ys

build     :: forall a. (forall b. (a -> b -> b) -> b -> b) -> [a]

build g = g (:) []

augment  :: forall a. (forall b. (a->b->b) -> b -> b) -> [a] -> [a]

augment g xs = g (:) xs

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

map _ []      = []

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

mapFB :: (elt -> lst -> lst) -> (a -> elt) -> a -> lst -> lst

mapFB c f = \x ys -> c (f x) ys

(++) :: [a] -> [a] -> [a]

(++) []      ys = ys

(++) (x:xs) ys = x : xs ++ ys

otherwise      :: Bool

otherwise      = True

type String = [Char]

eqString :: String -> String -> Bool

eqString []      []      = True

eqString (c1:cs1) (c2:cs2) = c1 == c2 && cs1 'eqString' cs2

eqString _      _      = False

```

```

maxInt, minInt :: Int

id                :: a -> a
id x              = x

assert :: Bool -> a -> a
assert _pred r = r

breakpoint :: a -> a
breakpoint r = r

breakpointCond :: Bool -> a -> a
breakpointCond _ r = r

data Opaque = forall a. 0 a

const                :: a -> b -> a
const x _           = x

(.) :: (b -> c) -> (a -> b) -> a -> c
(.) f g = \x -> f (g x)

flip                :: (a -> b -> c) -> b -> a -> c
flip f x y         = f y x

($)                :: (a -> b) -> a -> b
f $ x              = f x

($!)               :: (a -> b) -> a -> b
f $! x             = let !vx = x in f vx

until              :: (a -> Bool) -> (a -> a) -> a -> a
until p f = go

  where
    go x | p x      = x
          | otherwise = go (f x)

```

```

asTypeOf          :: a -> a -> a
asTypeOf          = const

-----

-- Functor/Applicative/Monad instances for IO
-----

instance Functor IO where

    fmap f x = x >>= (return . f)

instance Applicative IO where

    pure = return

    (<*>) = ap

instance Monad IO where

    m >> k      = m >>= \ _ -> k

    return     = returnIO

    (>>=)      = bindIO

    fail s     = failIO s

returnIO :: a -> IO a

returnIO x = IO $ \ s -> (# s, x #)

bindIO :: IO a -> (a -> IO b) -> IO b

bindIO (IO m) k = IO $ \ s -> case m s of (# new_s, a #) -> unIO (k a) new_s

thenIO :: IO a -> IO b -> IO b

thenIO (IO m) k = IO $ \ s -> case m s of (# new_s, _ #) -> unIO k new_s

unIO :: IO a -> (State# RealWorld -> (# State# RealWorld, a #))

unIO (IO a) = a

getTag :: a -> Int#

getTag !x = dataToTag# x

```

```

-----
-- Numeric primops
-----

quotInt, remInt, divInt, modInt :: Int -> Int -> Int

(I# x) 'quotInt' (I# y) = I# (x 'quotInt#' y)
(I# x) 'remInt' (I# y) = I# (x 'remInt#' y)
(I# x) 'divInt' (I# y) = I# (x 'divInt#' y)
(I# x) 'modInt' (I# y) = I# (x 'modInt#' y)

quotRemInt :: Int -> Int -> (Int, Int)

(I# x) 'quotRemInt' (I# y) = case x 'quotRemInt#' y of
    (# q, r #) ->
        (I# q, I# r)

divModInt :: Int -> Int -> (Int, Int)

(I# x) 'divModInt' (I# y) = case x 'divModInt#' y of
    (# q, r #) -> (I# q, I# r)

divModInt# :: Int# -> Int# -> (# Int#, Int# #)

x# 'divModInt#' y#
| isTrue# (x# ># 0#) && isTrue# (y# <# 0#) =
    case (x# -# 1#) 'quotRemInt#' y# of
        (# q, r #) -> (# q -# 1#, r +# y# +# 1# #)
| isTrue# (x# <# 0#) && isTrue# (y# ># 0#) =
    case (x# +# 1#) 'quotRemInt#' y# of
        (# q, r #) -> (# q -# 1#, r +# y# -# 1# #)
| otherwise
    =
    x# 'quotRemInt#' y#

shiftL# :: Word# -> Int# -> Word#

```

```

a 'shiftL#' b | isTrue# (b >=# WORD_SIZE_IN_BITS#) = 0##
              | otherwise                          = a 'uncheckedShiftL#' b

shiftRL# :: Word# -> Int# -> Word#

a 'shiftRL#' b | isTrue# (b >=# WORD_SIZE_IN_BITS#) = 0##
              | otherwise                          = a 'uncheckedShiftRL#' b

iShiftL# :: Int# -> Int# -> Int#

a 'iShiftL#' b | isTrue# (b >=# WORD_SIZE_IN_BITS#) = 0#
              | otherwise                          = a 'uncheckedIShiftL#' b

iShiftRA# :: Int# -> Int# -> Int#

a 'iShiftRA#' b | isTrue# (b >=# WORD_SIZE_IN_BITS#) = if isTrue# (a <# 0#)
                                                         then (-1#)
                                                         else 0#
              | otherwise                          = a 'uncheckedIShiftRA#' b

iShiftRL# :: Int# -> Int# -> Int#

a 'iShiftRL#' b | isTrue# (b >=# WORD_SIZE_IN_BITS#) = 0#
              | otherwise                          = a 'uncheckedIShiftRL#' b

#ifdef __HADDOCK__

data RealWorld

#endif

```

Bibliography

Aaland, M. (1998). *Photoshop for the Web*, O'Reilly & Associates, Inc., Sebastopol, CA, USA.

Alam, A. (2014). *A Programming System for End-User Functional Programming*, PhD thesis, University of Gloucestershire.

Allen, Rob and Lo, Nick and Brown, Steven (2008). *Zend Framework in Action*, Manning Publications Co., Greenwich, CT, USA.

Angelov, K. and Marlow, S. (2005). Visual Haskell: A Full-featured Haskell Development Environment, *Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell*, Haskell '05, ACM, New York, NY, USA, pp. 5–16.

Apfelmus, H. (2012). Reactive-banana.

URL: <http://www.haskell.org/haskellwiki/Reactive-banana>

Apfelmus, H. (2015a). A practical library for functional reactive programming.

URL: <http://hackage.haskell.org/package/reactive-banana>

Apfelmus, H. (2015b). Provides some GUI examples for the reactive-banana library, using wxHaskell.

URL: <http://hackage.haskell.org/package/reactive-banana-wx>

- Argudo, J. (2009). *CodeIgniter 1.7*, Packt Publishing.
- Aust, D., D'Souza, M. G., Gault, D., Gielis, D., Hartman, R., Hichwa, M., Kennedy, S., Kubicek, D., Mattamal, R., McGhan, D., Mignault, F., Nielsen, A. and Scott, J. (2011). *Expert Oracle Application Express*, 1st edn, Apress, Berkely, CA, USA.
- Backus, J. W. (1978). Can Programming Be Liberated From the von Neumann Style? A Functional Style and its Algebra of Programs, *Communications of the Association for Computing Machinery* **21**(8): 613–641.
- Bai, Y. (2003). The design and realization of a common syntax-directed editing system., *IRI*, pp. 85–92.
- Banyasad, O. and Cox, P. T. (2013). Design and Implementation of an Editor/Interpreter for a Visual Logic Programming Language, *International Journal of Software Engineering and Knowledge Engineering* **23**(6): 801–838.
- Barnes, J. G. (1984). *Programming in Ada (2Nd Ed.)*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- Batini, C. (1986). A layout algorithm for data flow diagrams, *IEEE Transactions on Software Engineering* **12**(1): 538–546.
- Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J.,

- Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J. and Thomas, D. (2001). Manifesto for Agile Software Development.
- Bennett, J. (2008). *Practical Django Projects*, Apress, Berkely, CA, USA.
- Bentrad, S. and Meslati, D. (2011). Visual programming and program visualization towards an ideal visual software engineering system, *International Journal on Information Technology* **1**(3): 7.
- Bernini, M. and Mosconi, M. (1994). VIPERS: A Data Flow Visual Programming Environment Based on the Tcl Language, *Proceedings of the workshop on Advanced visual interfaces, AVI '94*, ACM, New York, NY, USA, pp. 243–245.
- Bier, J. C., Goei, E. E., Ho, W. H., Lapsley, P. D., O'Reilly, M. P., Sih, G. C. and Lee, E. A. (1990). Gabriel: A Design Environment for DSP, *IEEE Micro* **10**(5): 28–45.
- Bird, R. and Wadler, P. (1988). *An introduction to functional programming*, Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK.
- Blake, G. and Bly, R. (1993). *The elements of technical writing*, Longman, London, England.
- Boehm, B. W. (1988). A Spiral Model of Software Development and Enhancement, *Computer* **21**(5): 61–72.
- Booch, G. (1994). *Object-oriented Analysis and Design with Applications (2Nd Ed.)*, Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA.

- Booch, G., Rumbaugh, J. and Jacobson, I. (2005). *Unified Modeling Language User Guide, The (2Nd Edition) (Addison-Wesley Object Technology Series)*, Addison-Wesley Professional, Boston, MA, USA.
- Broberg, N. (2012a). A type checker for Haskell as embodied syntactically by the `haskell-src-extends`.
- URL:** <http://hackage.haskell.org/package/haskell-type-exts>
- Broberg, N. (2012b). The Haskell-type-exts package.
- URL:** <http://hackage.haskell.org/package/haskell-type-exts>
- Broberg, N. (2014). A suite of annotable datatypes describing the abstract syntax of Haskell.
- URL:** <http://hackage.haskell.org/package/haskell-src-extends>
- Brockmann, R. (1990). *Writing Better Computer User Documentation: From Paper to Hypertext, Version 2.0*, John Wiley & Sons, Inc., New York, NY, USA.
- Brown, P. S. and Gould, J. D. (1987). An Experimental Study of People Creating Spreadsheets, *ACM Transactions on Information Systems* **5**(3): 258–272.
- Browne, J. C., Hyder, S. I., Dongarra, J., Moore, K. and Newton, P. (1995). Visual Programming and Debugging for Parallel Computing, *IEEE Parallel Distrib. Technol.* **3**(1): 75–83.
- Burbeck, S. (1987). *Applications Programming in Smalltalk-80: How to Use Model-View-Controller (MVC)*, Softsmarts, Incorporated.

- Burnett, M., Atwood, J., Walpole Djang, R., Reichwein, J., Gottfried, H. and Yang, S. (2001). Forms/3: A First-order Visual Language to Explore the Boundaries of the Spreadsheet Paradigm, *Journal of Functional Programming* **11**(2): 155–206.
- Burnett, M. M. and Ambler, A. L. (1993). An Interactive Approach to Visual Data Abstraction for Declarative Visual Programming Languages, *Technical report*, Corvallis, OR, USA.
- Burstall, R. (2000). Christopher Strachey—Understanding Programming Languages, *Higher-Order and Symbolic Computation* **13**(1-2): 51–55.
- Card, S. K., Newell, A. and Moran, T. P. (1983). *The Psychology of Human-Computer Interaction*, L. Erlbaum Associates Inc., Hillsdale, NJ, USA.
- Carr, M. E. (2011). Thoughts on LabVIEW.
URL: <http://www.paleotechnologist.net/?p=1502>
- Carrier, L. M., Cheever, N. A., Rosen, L. D., Benitez, S. and Chang, J. (2009). Multitasking across generations: Multitasking choices and difficulty ratings in three generations of Americans, *Computer Human Behavior* **25**(2): 483–489.
- Caserta, P. and Zendra, O. (2011). Visualization of the static aspects of software: A survey., *IEEE Transactions on Visualization and Computer Graphics* **17**(7): 913–933.
- Caspi, P., Pilaud, D., Halbwachs, N. and Plaice, J. A. (1987). LUSTRE: A

- Declarative Language for Real-time Programming, *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, ACM, New York, NY, USA, pp. 178–188.
- Chakravarty, M. M. T. and Keller, G. (2004). The Risks and Benefits of Teaching Purely Functional Programming in First Year, *Journal of Functional Programming* **14**(1): 113–123.
- Chambers, C., Chen, S., Le, D. and Scaffidi, C. (2012). The Function, and Dysfunction, of Information Sources in Learning Functional Programming, *Journal of Computing Sciences in Colleges* **28**(1): 220–226.
- Chang, S.-K. (1987). Visual Languages: A Tutorial and Survey, *IEEE Software* **4**(1): 29–39.
- Chitil, O. (2001). Compositional explanation of types and algorithmic debugging of type errors, *SIGPLAN Notices* **36**(10): 193–204.
- Chitil, O. and Luo, Y. (2007). Structure and properties of traces for functional programs, *Electron. Notes Theor. Comput. Sci.* **176**(1): 39–63.
- Christofides, N. (1975). *Graph Theory: An Algorithmic Approach (Computer Science and Applied Mathematics)*, Academic Press, Inc., Orlando, FL, USA.
- Claessen, K. and Hughes, J. (2011). Quickcheck: A lightweight tool for random testing of haskell programs, *SIGPLAN Not.* **46**(4): 53–64.

- Claessen, K., Hughes, J., Palka, M., Smallbone, N. and Svensson, H. (2010). Ranking programs using black box testing, *Proceedings of the 5th Workshop on Automation of Software Test*, AST '10, ACM, New York, NY, USA, pp. 103–110.
- Coleman, D., Arnold, P., Bodoff, S., Dollin, C., Gilchrist, H., Hayes, F. and Jeremaes, P. (1994). *Object-oriented Development: The Fusion Method*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA.
- Cooper, A., Reimann, R. and Cronin, D. (2007). *About face 3: the essentials of interaction design*, John Wiley & Sons, Inc., New York, NY, USA.
- Coulouris, G. F. and Dollimore, J. (1988). *Distributed systems: concepts and design*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- Courtney, A. and Elliott, C. (2001). Genuinely Functional User Interfaces, *Proceedings of the 2001 Haskell Workshop*.
- Cox, P. T. and Gauvin, S. (2011). Controlled Dataflow Visual Programming Languages, *Proceedings of the 2011 Visual Information Communication - International Symposium*, VINCI '11, ACM, New York, NY, USA, pp. 9:1–9:10.
- Cox, P. T. and Mulligan, I. J. (1985). Compiling the Graphical Functional Language PROGRAPH, *Proceedings of the 1985 ACM SIGSMALL Symposium on Small Systems*, SIGSMALL '85, ACM, New York, NY, USA, pp. 34–41.
- Cox, P. T. and Nicholson, P. K. (2008). Unification of Arrays in Spreadsheets

- with Logic Programming, in P. Hudak and D. S. Warren (eds), *PADL*, Vol. 4902 of *Lecture Notes in Computer Science*, Springer, Netherlands, pp. 100–115.
- Cox, P. T., Plimmer, B. and Rodgers, P. J. (eds) (2012). *Diagrammatic Representation and Inference - 7th International Conference*, Vol. 7352 of *Lecture Notes in Computer Science*, Springer, Netherlands.
- Crenshaw, D. (2008). *The Myth of Multitasking: How "Doing It All" Gets Nothing Done*, The Jossey-Bass business & management series, John Wiley & Sons, Inc., New York, NY, USA.
- Cypher, A. (ed.) (1993). *Watch What I Do: Programming by Demonstration*, MIT Press, Cambridge, MA, USA.
- Dahl, O. J., Dijkstra, E. W. and Hoare, C. A. R. (eds) (1972). *Structured Programming*, Academic Press Ltd., London, UK, UK.
- Darlington, J., Henderson, P. and Turner, D. (1982). *Functional Programming and Its Applications: An Advanced Course*, CREST advanced courses, Cambridge University Press, Cambridge, UK.
- Denicolo, P. and Becker, L. (2012). *Developing Research Proposals, Success in Research*, SAGE Publications, London, UK.
- Downs, E., Clare, P. and Coe, I. (1988). *Structured Systems Analysis and Design Method: Application and Context*, Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK.

Eberts, R. E. (1994). *User interface design*, Prentice Hall international series in industrial and systems engineering, Prentice Hall, Englewood Cliffs, NJ, USA.

Ebrahimi, A. (1994). Novice programmer errors: language constructs and plan composition, *International Journal Human-Computer Studies* **41**(4): 457–480.

Elliott, C. (2008). Trimming inputs in functional reactive programming.

URL: <http://conal.net/blog/posts/>

Elliott, C. and Hudak, P. (1997). Functional Reactive Animation, in S. Peyton Jones, M. Tofte and A. M. Berman (eds), *ICFP*, ACM, New York, NY, USA, pp. 263–273.

Erwig, M. and Meyer, B. (1995). Heterogeneous Visual Languages-integrating Visual and Textual Programming, *Proceedings of the 11th International IEEE Symposium on Visual Languages, VL '95*, IEEE Computer Society, Washington, DC, USA, pp. 318–.

URL: <http://dl.acm.org/citation.cfm?id=832276.834317>

Fekete, J.-D. and Beaudouin-Lafon, M. (1996). Using the multi-layer model for building interactive graphical applications, *Proceedings of the 9th annual ACM symposium on User interface software and technology, UIST '96*, ACM, New York, NY, USA, pp. 109–118.

Findler, R. B., Clements, J., Flanagan, C., Flatt, M., Krishnamurthi, S., Steck-

- ler, P. and Felleisen, M. (2002). DrScheme: A Programming Environment for Scheme, *Journal of Functional Programming* **12**(2): 159–182.
- Fowler, M. (2002). *Patterns of Enterprise Application Architecture*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- Freeman, A. (2012). *Pro ASP.NET MVC 4*, Apress Series, Apress, Berkely, CA, USA.
- Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- Gibbons, J. (2002). Towards a Colimit-Based Semantics for Visual Programming, *Proceedings of the 5th International Conference on Coordination Models and Languages*, COORDINATION '02, Springer-Verlag, London, UK, UK, pp. 166–173.
- Giorgidze, G. and Nilsson, H. (2008). Switched-On Yampa: Declarative Programming of Modular Synthesizers, *Proceedings of the 10th International Conference on Practical Aspects of Declarative Languages*, PADL'08, Springer-Verlag, Berlin, Heidelberg, pp. 282–298.
- Gittins, D. (1986). Icon-based human-computer interaction., *International Journal of Man-Machine Studies* **24**(6): 519–543.
- Gordon, M. (2000). From LCF to HOL: a short history, *in* G. D. Plotkin,

- C. Stirling and M. Tofte (eds), *Proof, Language, and Interaction*, MIT Press, Cambridge, MA, USA, pp. 169–186.
- Green, T. R. G. (1990). Programming languages as information structures, in J.-M. Hoc, T. R. G. Green, R. Samurcay and D. J. Gilmore (eds), *Psychology of Programming*, London: Academic Press.
- Grossman, T., Fitzmaurice, G. and Attar, R. (2009). A survey of software learnability: Metrics, methodologies and guidelines, *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '09*, ACM, New York, NY, USA, pp. 649–658.
- Grune, D. (1977). Towards the design of a super-language of algol 68 for the standard prelude (excerpt), *Proceedings of the Strathclyde ALGOL 68 Conference*, ACM, New York, NY, USA, pp. 78–81.
- Gurd, J. R., Kirkham, C. C. and Watson, I. (1985). The manchester prototype dataflow computer, *Communications of the Association for Computing Machinery* **28**(1): 34–52.
- Haiduc, S., Aponte, J., Moreno, L. and Marcus, A. (2010). On the use of automated text summarization techniques for summarizing source code, *Proceedings of the 2010 17th Working Conference on Reverse Engineering, WCRE '10*, IEEE Computer Society, Washington, DC, USA, pp. 35–44.
- Halbert, D. C. (1984). *Programming by Example*, PhD thesis. AAI8512843.

Haskell-Cafe (2015). The Haskell-Cafe Archives.

URL: <https://mail.haskell.org/pipermail/haskell-cafe/>

Haskell Website - Base (n.d.). Source Code of Functor, Applicative and Monad.

URL: <https://hackage.haskell.org/package/base-4.8.0.0/docs/src/GHC-Base.html>

Haskell Website - reactive-banana (n.d.). Source Code of reactive-banana.

URL: <https://hackage.haskell.org/package/reactive-banana-0.7.0.1/docs/>

Haskell Website - wxHaskell (n.d.). Source Code of wxHaskell.

URL: <https://hackage.haskell.org/package/wx-0.12.1.6/docs/>

Himmelstrup, D. (2006). Interactive debugging with ghci, *Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell*, Haskell '06, ACM, New York, NY, USA, pp. 107–107.

Hofer, B., Ribeiro, A., Wotawa, F., Abreu, R. and Getzner, E. (2013). On the empirical evaluation of fault localization techniques for spreadsheets, *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering*, FASE'13, Springer-Verlag, Berlin, Heidelberg, pp. 68–82.

Horwitz, S. and Teitelbaum, T. (1986). Generating Editing Environments Based on Relations and Attributes, *ACM Trans. Program. Lang. Syst.* **8**(4): 577–608.

- Hubbell, T. J., Langan, D. D. and Hain, T. F. (2006). A Voice-activated Syntax-directed Editor for Manually Disabled Programmers, *Proceedings of the 8th International ACM SIGACCESS Conference on Computers and Accessibility*, Assets '06, ACM, New York, NY, USA, pp. 205–212.
- Hudak, P. (1989). Conception, Evolution, and Application of Functional Programming Languages, *ACM Computing Surveys* **21**(3): 359–411.
- Hudak, P. (2000). *The Haskell School of Expression: Learning Functional Programming Through Multimedia*, Cambridge University Press.
- Hughes, J. (1989). Why functional programming matters, *The Computer Journal*, *Special issue on Lazy functional programming* **32**(2): 98–107.
- Hughes, J. (2000). Generalising Monads to Arrows, *Science of Computer Programming* **37**(1-3): 67–111.
- Hughes, J. K. (1986). *PL/I Structured Programming*, John Wiley & Sons, Inc., New York, NY, USA.
- Hundhausen, C. D., Farley, S. and Lee Brown, J. (2006). Can Direct Manipulation Lower the Barriers to Programming and Promote Positive Transfer to Textual Programming? An Experimental Study, *Proceedings of the Visual Languages and Human-Centric Computing*, VLHCC '06, IEEE Computer Society, Washington, DC, USA, pp. 157–164.
- Jensen, K. and Wirth, N. (1974). *PASCAL User Manual and Report*, Springer-Verlag New York, Inc., New York, NY, USA.

- Johnson, G. W. (1997). *LabVIEW Graphical Programming: Practical Applications in Instrumentation and Control*, 2nd edn, McGraw-Hill Education, Berkshire, UK.
- Joosten, S., Berg, K. V. D. and Hoeven, G. V. D. (2008). Teaching functional programming to first-year students, *Journal of Functional Programming* **3**: 49–65.
- Kay, A. C. (1996). History of programming languages—ii, ACM, New York, NY, USA, chapter The Early History of Smalltalk, pp. 511–598.
- Kimura, T., Choi, J. and Mack, J. (1986). *A Visual Language for Keyboardless Programming*, Technical report, Washington University, Department of Computer Science, St. Louis, MO, USA.
- Ko, A. J., Abraham, R., Beckwith, L., Blackwell, A., Burnett, M., Erwig, M., Scaffidi, C., Lawrance, J., Lieberman, H., Myers, B., Rosson, M. B., Rothermel, G., Shaw, M. and Wiedenbeck, S. (2011). The state of the art in end-user software engineering, *ACM Computing Surveys* **43**(3): 21:1–21:44.
- Ko, A. J. and Myers, B. A. (2004). Designing the whyline: a debugging interface for asking questions about program behavior, *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '04*, ACM, New York, NY, USA, pp. 151–158.
- Ko, A. J., Myers, B. A. and Aung, H. H. (2004). Six Learning Barriers in End-User Programming Systems, *Proceedings of the 2004 IEEE Symposium*

- on *Visual Languages - Human Centric Computing*, VLHCC '04, IEEE Computer Society, Washington, DC, USA, pp. 199–206.
- Kochan, S. (2009). *Programming in Objective-C 2.0*, 2nd edn, Addison-Wesley Professional, Boston, MA, USA.
- Korfhage, R. (1984). Query Enhancement by User Profiles, *SIGIR*, pp. 111–121.
- Krasner, G. E. and Pope, S. T. (1988). A Cookbook for Using the Model-view Controller User Interface Paradigm in Smalltalk-80, *Journal of Object Oriented Programming* **1**(3): 26–49.
- Lammers, S. (1986). *Programmers at work*, number v. 1 in *The At Work Series*, Microsoft Press.
- Larman, C. (2004). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*, 3rd edn, Prentice Hall PTR, Upper Saddle River, NJ, USA.
- Lau, K.-K., Bush, V. J. and Jinks, P. J. (1994). Towards an Introductory Formal Programming Course, *SIGCSE Bulletin* **26**(1): 121–125.
- Lazar, J., Jones, A., Hackley, M. and Shneiderman, B. (2006). Severity and impact of computer user frustration: A comparison of student and workplace users, *Interact. Comput.* **18**(2): 187–207.
- Le Guernic, P., Gautier, T., Le Borgne, M. and Le Maire, C. (1991). Pro-

- gramming Real-Time Applications with Signal, *Proceedings of the IEEE* **79**(9): 1321–1336.
- Leijen, D. (2004). wxHaskell: a portable and concise GUI library for Haskell, *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell, Haskell '04*, ACM, New York, NY, USA, pp. 57–68.
- Leijen, D. (2014). wxHaskell is a portable and native GUI library for Haskell.
URL: <http://hackage.haskell.org/package/wx>
- Lewis, C. and Norman, D. A. (1995). Human-computer interaction, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, chapter Designing for error, pp. 686–697.
- Lewis, C. and Olson, G. (1987). Can principles of cognition lower the barriers to programming?, *Empirical studies of programmers: second workshop*, Ablex Publishing Corp., Norwood, NJ, USA, pp. 248–263.
- Lipovaa, M. (2011). Learn You a Haskell for Great Good!
URL: <http://learnyouahaskell.com/>
- Lodding, K. (1983). Iconic Interfacing, *IEEE Computer Graphics and Applications* **3**: 11–20.
- Mandel, T. (1997). *The elements of user interface design*, John Wiley & Sons, Inc., New York, NY, USA.
- Marcus, A. (1992). *Graphic design for electronic documents and user interfaces*, ACM, New York, NY, USA.

- Marques, B. R. C., Levitt, S. P. and Nixon, K. J. (2012). Software visualisation through video games, *Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference, SAICSIT '12*, ACM, New York, NY, USA, pp. 206–215.
- Mayhew, D. J. (1999). *The usability engineering lifecycle : a practioner's handbook for user interface design*, Morgan Kaufmann, San Francisco, CA, USA.
- McCormack, J. and Asente, P. (1988). An overview of the x toolkit, *Proc. of the 1st Symposium on User Interface Software*, Banff, Canada, pp. 46–55.
- McLaughlin, B. D., Pollice, G. and West, D. (2006). *Head First Object-Oriented Analysis and Design: A Brain Friendly Guide to OOA&D (Head First)*, O'Reilly Media, Inc., Sebastopol, CA, USA.
- McNiff, J. (1988). *Action Research: Principles and Practice*, Routledge, Oxon, UK.
- Medina, J. (2010). *Brain Rules: 12 Principles for Surviving and Thriving at Work, Home, and School*, Pear Press, Seattle, WA, USA.
- Meyers, S. (1991). Difficulties in Integrating Multiview Development Systems, *IEEE Software* **8**(1): 49–57.
- Microsoft Corporation (n.d.). VPL Documentation. last checked: 15.07.2014.
URL: <http://msdn.microsoft.com/en-us/library/bb483088.aspx>

- Milner, R. (1984). A Proposal for Standard ML, *LISP and Functional Programming*, Department of Computer Science, University of Edinburgh, UK, pp. 184–197.
- Minor, S. (1991). *Interacting with Structure-Oriented Editors*, Department of Computer Science, Lund University, Lund, Sweden, Lund University, Department of Computer Science, Lund, Sweden.
- Mow, I. (2008). Issues and difficulties in teaching novice computer programming, in M. Iskander (ed.), *Innovative Techniques in Instruction Technology, E-learning, E-assessment, and Education*, Springer-Verlag, Berlin, Heidelberg, pp. 199–204.
- Myers, B. A. (1990). Taxonomies of visual programming and program visualization, *Journal of Visual Languages and Computing* **1**(1): 97–123.
- Nardi, B. A. (1993). *A Small Matter of Programming: Perspectives on End User Computing*, 1st edn, MIT Press, Cambridge, MA, USA.
- Nardi, B. A. and Miller, J. R. (1990). An ethnographic study of distributed problem solving in spreadsheet development, *Proceedings of the 1990 ACM conference on Computer-supported cooperative work, CSCW '90*, ACM, New York, NY, USA, pp. 197–208.
- Nassi, I. and Shneiderman, B. (1973). Flowchart techniques for structured programming, *ACM SIGPLAN Notices* **8**(8): 12–26.
- Newby, M. and Nguyen, T. (2010). Using the same problem with different tech-

- niques in programming assignments: An empirical study of its effectiveness, *Journal of Information Systems Education* **21**(4): 375–382.
- Nielsen, J. (1992). The Usability Engineering Life Cycle, *IEEE Computer* **25**(3): 12–22.
- Nielsen, J. (1993). *Usability engineering*, Academic Press, Boston, MA, USA.
- Norman, D. A. (1989). Perspectives on the computer revolution, Ablex Publishing Corp., Norwood, NJ, USA, chapter The trouble with UNIX (1981), pp. 243–260.
- Ohshima, Y., Lunzer, A., Freudenberg, B. and Kaehler, T. (2013). Kscript and ksworld: A time-aware and mostly declarative language and interactive gui framework, *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, ACM, New York, NY, USA, pp. 117–134.
- O’Sullivan, B., Goerzen, J. and Stewart, D. (2008). *Real World Haskell*, 1st edn, O’Reilly Media, Inc., Sebastopol, CA, USA.
- Pane, J. F. (1998). Designing a Programming System for Children with a Focus on Usability, *CHI 98 Conference Summary on Human Factors in Computing Systems*, CHI ’98, ACM, New York, NY, USA, pp. 62–63.
- Pane, J. F., Myers, B. A. and Miller, L. B. (2002). Using HCI techniques to design a more usable programming system, *IEEE*, pp. 198–206.

- Panko, R. R. (1998). What We Know About Spreadsheet Errors, *Journal of End-User Computing* **10**(2): 15–21.
- Panko, R. R. (2000). Spreadsheet Errors: What We Know. What We Think We Can Do., *Proceedings of the Spreadsheet Risk Symposium* .
- Patrick, T., Roman, S., Petrusha, R. and Lomax, P. (2006). *Visual Basic 2005 - in a nutshell: a desktop quick reference*, 3rd edn, O’Reilly, Sebastopol, CA, USA.
- Pembeci, I., Nilsson, H. and Hager, G. (2002). System Presentation – Functional Reactive Robotics: An Exercise in Principled Integration of Domain-Specific Languages, *Principles and Practice of Declarative Programming*, Pittsburgh, Pennsylvania, USA, pp. 168–179.
- Peterson, J., Hager, G. and Hudak, P. (1999). A Language for Declarative Robotic Programming, *International Conference on Robotics and Automation*, pp. 1144–1151.
- Petre, M. and Blackwell, A. F. (1999). Mental imagery in program design and visual programming, *Int. J. Hum.-Comput. Stud.* **51**(1): 7–30.
- Peyton Jones, S. (ed.) (2002). *Haskell 98 Language and Libraries: The Revised Report*, <http://haskell.org/>.
- URL:** <http://haskell.org/definition/haskell98-report.pdf>
- Peyton Jones, S., Gordon, A. and Finne, S. (1996). Concurrent Haskell, *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of*

- Programming Languages*, POPL '96, ACM, New York, NY, USA, pp. 295–308.
- Price, J. and Apple Computer, I. (1984). *How to Write a Computer Manual: a Handbook of Software Documentation*, Benjamin/Cummings Publishing Company, CA, USA.
- Rasure, J. and Williams, C. S. (1991). An integrated data flow visual language and software development environment, *Journal of Visual Languages and Computing* **2**(3): 217–246.
- Read, M. (1996). Specifying Direct Manipulation within Program Editors, *Proceedings of the 6th Australian Conference on Computer-Human Interaction (OZCHI '96)*, OZCHI '96, IEEE Computer Society, Washington, DC, USA, pp. 346–.
- Reekie, J. H. (1994). Visual Haskell: A first attempt, *Technical Report 94.5*, Key Centre for Advanced Computing Sciences, University of Technology, Sydney, PO BOX 123, Broadway, NSW 2007, Australia.
- Reid, A., Peterson, J., Hudak, P. and Hager, G. (1999). Prototyping Real-Time Vision Systems: An Experiment in DSL Design, *Proceedings of ICSE 99: International Conference on Software Engineering*, ACM, New York, NY, USA.
- Reiss, S. P. (1985). PECAN: Program Development Systems that Support Multiple Views, *IEEE Trans. Software Eng.* **11**(3): 276–285.

- Reiss, S. P. (1986). An Object-oriented Framework for Graphical Programming (Summary Paper), *Proceedings of the 1986 SIGPLAN Workshop on Object-oriented Programming, OOPWORK '86*, ACM, New York, NY, USA, pp. 49–57.
- Reiss, S. P. (1987). Visual languages and the GARDEN system, *Psychology. Selected contributions on Visualization in programming*, Springer-Verlag, London, UK, pp. 178–198.
- Robins, A., Rountree, J. and Rountree, N. (2003). Learning and Teaching Programming: A Review and Discussion, *Computer Science Education* **13**(2): 137–172.
- Roff, J. T. (2003). *UML: A Beginner's Guide*, 1st edn, McGraw-Hill, Inc., New York, NY, USA.
- Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorenzen, W. (1991). *Object-oriented Modeling and Design*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA.
- Ruparelia, N. B. (2010). Software Development Lifecycle Models, *SIGSOFT Software Engineering Notes* **35**(3): 8–13.
- Russell, D. (2001). *FAD: A Functional Analysis and Design Methodology*, PhD thesis, Computing Laboratory, University of Kent at Canterbury.
URL: <http://www.cs.kent.ac.uk/pubs/2001/1152>
- Ryder, C. and Thompson, S. J. (2005). Software Metrics: Measuring Haskell,

- Trends in Functional Programming*, Vol. 6 of *Trends in Functional Programming*, Intellect, UK/The University of Chicago Press, USA, pp. 31–46.
- Scaffidi, C., Brandt, J., Burnett, M. M., Dove, A. and Myers, B. A. (2012). SIG: end-user programming, in J. A. Konstan, E. H. Chi and K. Höök (eds), *CHI Extended Abstracts*, ACM, New York, NY, USA, pp. 1193–1996.
- Segal, J. (1994). Empirical studies of functional programming learners evaluating recursive functions, *Instructional Science* **22**: 385–411.
- Segal, J. (2007). Some Problems of Professional End User Developers, *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, IEEE Computer Society, Washington, DC, USA, pp. 111–118.
- S’erot, J. (2000). CAMLFLOW: a CAML to data-flow graph translator, in S. Gilmore (ed.), *2nd Scottish Functional Programming Workshop*, Vol. 2, Intellect, St. Andrews, Scotland, pp. 129–144. ISBN 1-84150-058-5.
- Sheard, T. and Jones, S. P. (2002). Template meta-programming for Haskell, *SIGPLAN Notices* **37**(12): 60–75.
- Shneiderman, B. (1986). Seven Plus or Minus Two Central Issues in Human-Computer Interaction, *Proceedings of CHI-86*, Boston, MA, USA, pp. 343–349.
- Shneiderman, B. and Plaisant, C. (2004). *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, 4th edn, Pearson Addison Wesley, Reading, MA, USA.

- Shu, N. C. (ed.) (1988). *Visual programming*, Van Nostrand Reinhold Co., New York, NY, USA.
- Smith, D. C. (1993). Watch what i do, MIT Press, Cambridge, MA, USA, chapter Pygmalion: An Executable Electronic Blackboard, pp. 19–48.
- Smith, S. and Mosier, J. (1986). *Guidelines for Designing User Interface Software*, Mitre Corporation, Bedford, MA, USA.
- Software, R. (1994). *Getting Started with Rational Rose Revision 2.5*, Rational Rose.
- Sommerville, I. (1995). *Software Engineering (5th Ed.)*, Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.
- Spohrer, J. C., Soloway, E. and Pope, E. (1985). A goal/plan analysis of buggy pascal programs, *Human-Computer Interaction* **1**(2): 163–207.
- SPSS Inc. (2007). *SPSS base 16.0 user's guide*, SPSS Inc., Chicago, IL, USA.
- Steele Jr., G. L. and Sussman, G. J. (1978). The Revised Report on Scheme, a Dialect of LISP, *Technical Report 452*, Massachusetts Institute of Technology, Cambridge, MA, USA.
- Stoughton, A. (2008). A Functional Model-View-Controller Software Architecture for Command-Oriented Programs, *Proceedings of the ACM SIGPLAN workshop on Generic programming, WGP '08*, ACM, New York, NY, USA, pp. 1–12.

- Stroustrup, B. (2000). *The C++ Programming Language*, 3rd edn, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- Szlenk, M. (2011). Metamodel and uml profile for functional programming languages, *Dependable Computer Systems*, Springer-Verlag, Berlin, Heidelberg, pp. 233–242.
- Tamassia, R., Di Battista, G. and Batini, C. (1988). Automatic graph drawing and readability of diagrams, *IEEE Transactions on Systems, Man and Cybernetics* **18**(1): 61–79.
- Tanimoto, S. L. and Glinert, E. P. (1990). Designing Iconic Programming Systems: Representation and Learnability, in E. P. Glinert (ed.), *Visual Programming Environments: Applications and Issues*, IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 330–336.
- Tate, B. and Hibbs, C. (2006). *Ruby on Rails: Up and Running*, O’Reilly Media, Inc., Sebastopol, CA, USA.
- Trudeau, R. J. (1993). *Introduction to Graph Theory*, Dover Publications, New York, NY, USA.
- Uustalu, T. and Vene, V. (2006). The Essence of Dataflow Programming, in Z. Horvth (ed.), *Central European Functional Programming School*, Vol. 4164 of *Lecture Notes in Computer Science*, Springer, Berlin Heidelberg, pp. 135–167.

- Van Tassel, D. (1974). *Program style, design, efficiency, debugging, and testing*, Prentice-Hall, Englewood Cliffs, NJ, USA.
- Vangheluwe, H. and de Lara, J. (2003). Foundations of multi-paradigm modeling and simulation: Computer automated multi-paradigm modelling: Metamodeling and graph transformation, *Proceedings of the 35th Conference on Winter Simulation: Driving Innovation*, WSC '03, Winter Simulation Conference, New Orleans, Louisiana, USA, pp. 595–603.
- Wadge, W. W. and Ashcroft, E. A. (1985). *LUCID, the Dataflow Programming Language*, Academic Press Professional, Inc., San Diego, CA, USA.
- Wadler, P. (1995). Monads for Functional Programming, *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, Springer-Verlag, London, UK, UK, pp. 24–52.
- Wadler, P. (1997). How to Declare an Imperative, *ACM Computing Surveys* **29**(3): 240–263.
- Wadler, P. (1998). Why no one uses functional languages, *SIGPLAN Notices* **33**(8): 23–27.
- Wakeling, D. (2001). A design methodology for functional programs, *Proceedings of the 2Nd International Conference on Semantics, Applications, and Implementation of Program Generation*, SAIG'01, Springer-Verlag, Berlin, Heidelberg, pp. 146–161.

- Wallis, C. (2006). The Multitasking Generation, *TIME* .
URL: <http://www.time.com/time/magazine/article/0,9171,1174696,00.html>
- Wan, Z. and Hudak, P. (2000). Functional Reactive Programming from First Principles, *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, ACM, New York, NY, USA, pp. 242–252.
- Waters, R. C. (1984). The Programmer's Apprentice: Knowledge Based Program Editing, in D. R. Barstow, H. E. Shrobe and E. Sandewall (eds), *Interactive Programming Environments*, McGraw-Hill, New York, NY, USA, pp. 464–486.
- West, S. and Kahl, W. (2009). A Generic Graph Transformation, Visualisation, and Editing Framework in Haskell, *ECEASST* **18**.
- Whitley, K. N. and Blackwell, A. F. (1997). Visual programming: the outlook from academia and industry, *Papers presented at the seventh workshop on Empirical studies of programmers*, ESP '97, ACM, New York, NY, USA, pp. 180–208.
- Wickens, C. D. and Hollands, J. G. (1999). *Engineering Psychology and Human Performance*, 3rd edn, Prentice Hall, Englewood Cliffs, NJ, USA.
- Wolfram, S. (2003). *The Mathematica book*, 5th edn, Wolfram Media Inc., Champaign, IL, USA.
- Zhang, K. (2007). *Visual languages and applications*, Springer, Netherlands.