Interacting with conditionals in Viskell

F Wibbelink
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
f.wibbelink@student.utwente.nl

ABSTRACT

Viskell is a new attempt at a visual programming language and complementing touchscreen-oriented editor designed for functional programming. However, the language currently lacks one of the most useful mechanisms in functional programming: pattern matching. Pattern matching allows a developer to concisely and clearly write conditional code in a program. The goal of this research is to investigate how conditionals can be implemented and visualized in a visual functional language and how a user can interact with them. We present an implementation in Viskell and discuss shortcomings and specialisations of the chosen notation.

Keywords

conditional pattern matching visual functional language Viskell multi-touch

1. INTRODUCTION

Recently, an initial version of a *visual programming language* (VPL) based on the function language Haskell has been developed at the University of Twente. One of the shortcomings of this system is the lack of support for conditionals. Among other shortcomings are the lack of general recursion, higher-order type expressions and function grouping and the environment is currently limited to predefined functions. Extending the system Viskell with all these features would increase the usability substantially. However, this work will investigate to what extent it is possible to support conditionals and recursion only.

Arguments favouring visual programming over conventional textual programming are the multi-dimensional way humans processes visual information âĂŞwhereas textual programming is largely one-dimensionalâĂŞ the often lesser emphasis on syntax and the often easier representation of concurrent and real-time software.[**?**]

Recent developments in the ubiquity of new user interfaces, mainly touch screens, and the ever-increasing processing powers of computers bring new opportunities to VPLs. Whereas earlier VPL programming environments relied on a mouse and keyboard, which makes a VPL hardly more usable than a conventional language from an interaction standpoint, VPLs can now be developed that benefit from the ease of a touch interface and gestures.

The reason to choose Viskell for this research, is because it is built for a functional language with an environment designed for various interfaces, including multi-touch screens. The abstractions in functional languages[**?**] lend themselves to clean visual code. Whereas procedural VPLs pass around control flow and either hide relations between values or become an entangled mess of wires, functional and dataflow VPLs pass around values and abstract from control flow. This allows the user to reason about the transformations of data instead of having to focus on how the program is run. Moreover, whereas dataflow VPLs in general lack first-class functions and may keep internal state, a functional VPL does the opposite, achieving more reusability and purity.

This paper will first discuss a brief history of visual programming, functional programming and an overview of the target language, Viskell. Secondly, a set of requirements is discussed to which conditionals must adhere. This includes both language features and means of user interaction. Thirdly, the design, implementation and evaluation of conditionals in Viskell is discussed. Lastly, conclusions and recommendations of this study are discussed.

2. BACKGROUND AND RELATED WORK

2.1 Visual programming

Visual programming has been proposed as an alternative to programming in a textual language. In this paper *visual programming languages* (VPLs) concern only programming languages in which both representation and interaction are performed visually, rather than an editor that can generate a visual representation while requiring the user to interact with a text program.
An ongoing effort to create VPLs and environments since the 60s has lead to numerous VPLs in nearly all language paradigms[**?**]. Earlier attempts often focussed on user interaction using the mouse or keyboard. This can be advantageous in domain-specific languages in which the interface and language is tailored to the interaction medium, though it is believed that general-purpose VPLs allow for significantly less productivity than *integrated development environments* (IDEs) for text languages. However, with the introduction of multi-touch screens to the general public, a new way of interacting with VPLs can be explored, aiding users with gestures and collaborative programming.

Most languages discussed in [**?**] are domain-specific and are built to visualize a specific kind of task, so they can be used by people without a thorough understanding of programming. However, within general visual languages there are three major paradigms: procedural, dataflow and functional.

24th Twente Student Conference on IT January 22nd, 2016, Enschede, The Netherlands.

Procedural VPLs mostly aim to visualise or teach procedural and object-oriented programming in text. Control flow and the sequencing of actions is emphasised in this paradigm. The visual notation usually consists of blocks that interlock like puzzle pieces. The sequential nature allows for a time dimension in a routine, e.g. waiting for a number of seconds. Examples include Scratch and LEGO Mindstorms.

The most abundant class of VPLs is that of dataflow languages. These languages resemble assembly lines and are often parallelised. Usually each node in a program is a self-contained process and edges between nodes transfer data. Nodes can also keep internal state –visualised as memory or a feedback loop– to process the same input differently a second time. Similar to procedural VPLs, they allow for a time dimension naturally. Examples include LabVIEW, SimuLink and Blender's compositor.

Functional VPLs are similar to their dataflow counterpart, except that nodes lack internal state. As functional languages, they often introduce functions as data and referential transparency. Unlike dataflow and procedural languages they do not allow for a time dimension easily, although this can be achieved with *functional reactive programming*. Examples include Visual Haskell, Viskell and older versions of SubText.

2.2 Pattern matching

Pattern matching is a technique in Haskell[**?**], among other functional languages, to branch a function based on the constructor of an argument. For example, the integer type has a constructor for each value, so a function can contain a branch for any specific number to evaluate a different expressions than for other numbers. This is shown in the factorial example below. Likewise, a list either contains no items, using the empty constructor, or contains an item and a another list, using the *cons* constructor.
Moreover, when matching on a constructor that requires parameters, the values inside can be bound to a name or yet another constructor.

2.3 General conditions

Haskell also allows for boolean guards in addition to patterns. In this case, an expression (often using the arguments of the function) is evaluated and the corresponding branch is chosen if the expression evaluates to `True`. This is shown in the fibonacci example below. Haskell allows to combine these two to create a very powerful and concise method of branching, as shown in the `filter` example. `filter` iterates over a list and builds a new list containing only the values for which the predicate function is true.

```
factorial 0 = 0
factorial n = n * factorial (n-1)

fib n | n == 0    = 0
      | n == 1    = 1
      | otherwise = fib (n-1) + fib (n-2)

filter :: (a -> Bool) -> [a] -> [a]
filter pred [] = []
filter pred (x:xs)
        | pred x    = (x : filter pred xs)
        | otherwise = filter pred xs
```

Haskell has accumulated multiple extensions to conditions over the years. Among others are pattern guards, which can take the spot of a boolean guard but try to match an expression to a pattern instead of `True`, and view patterns, which replace any parameter name with a pattern guard using the value of that parameter. However, all these forms of pattern matching, along with if-then-else, can be reduced to `case` expressions, as is customary in haskell compilers. It is believed that Viskell will not benefit from transferring these extensions directly to extra syntax, although the conditional blocks may include an analogue for some of the concepts.

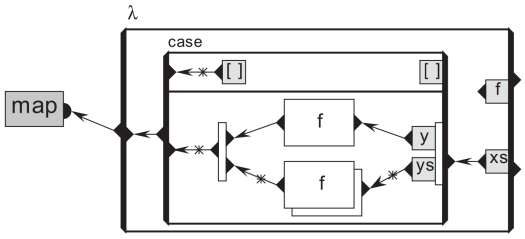

Figure 1. The `map` function in Visual Haskell

2.4 Conditionals in visual programming

Conditionals have been implemented in VPLs in different ways. The *Aardappel* language[**?**] presents conditionals in the form of pattern matching, in which every pattern plus function body is seperated by a thick horizontal bar. Guards can be emulated by surrounding the cases, in a lambda function that evaluates the guards, and matching on truth values. Since every function has its own window, the different patterns are well recognizable. On the other hand, the language is highy inaccessible due to the way functions are composed and the way parameters are represented by their values instead of their type, a symbol or a name. Moreover, there is no visualization of the structure of the whole program.

Another visual language, *PHF* as defined by [**?**], presents conditionals by splitting the wire of an incoming variable and feeding it to an expression in multiple containers. It then executes the functions in the container for which the expression is true. This means that this technique is limited to boolean guards and cannot deconstruct parameters.

In the visual language *Subtext*[**?**] conditionals are presented as decision tables. The columns of these tables allow a guard for each of the arguments and local variables of the function; the rows contain arguments, local variables and additional space for function calls. Functions in the cells are then executed based on whether their column is valid during a certain call. This allows the user to switch between specialized and generic code at any point during a function. The editor continuously checks whether parallel functions form a partition and shows the missing or overlapping cases. It should be remarked, however, that later versions of Subtext dropped the use of decision tables altogether.

2.5 Visual Haskell

An early attempt to visualize functional programming has been performed in [**?**]. This approach uses the box-and-wire model, in which boxes represent functions and wires represent argument passing, as depicted in Figure **??**. This particular notation also allows arguments to be named, such that they can be used elsewhere in a function without connecting it by a line. The downside is that naming is required for multiple uses of an output and that this feature is easily misused to detach nearly all wires, thereby obscuring program flow. Another pecularity about the notation is that it has a special line styles for the most common types and node styles for the most common data con-

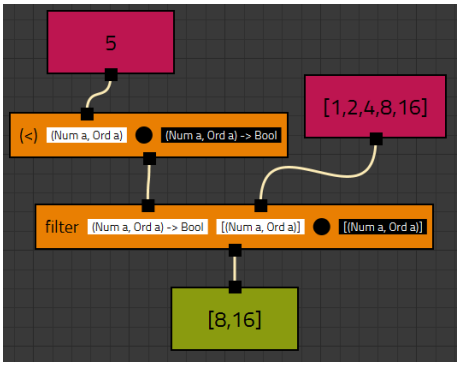

Figure 2. An example program in Viskell

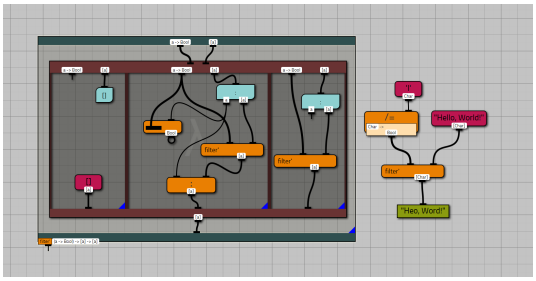

Figure 3. The `filter` function as envisioned

structors, but otherwise hides the type information from the user.

Conditionals are represented by boxes with the desired patterns and guards stuck to the input side, which are then stacked together in a lambda block. Patterns can contain (inverse) data constructors and constants and can be nested by attaching another pattern to the output of an existing one. It is unclear how the author envisioned the user to choose data constructors or turn them into patterns.

3. VISKELL

A very recent attempt at a VPL and complementary editor is Viskell[**?**], performed as a project at the CAES chair at the University of Twente. This project was developed in the first half of 2015 and focuses on a multi-touch interface, collaborative programming and online typechecking. Types play a central role in Viskell. Based on Haskell, it currently supports all functions in the Prelude. Additionally, there are nodes to produce values and show results, which are evaluated real-time to assist in programming by example. In contrast to all investigated languages, Viskell allows for partial application of functions by moving the knot between input and output arguments leftwards, as depicted in the compare function in Figure **??** that is by default a function of two arguments returning a boolean.

In important aspect of Viskell is the online typechecking. Every anchor on a block shows its type information. Whenever a user connects an input and an output the typechecker updates the type information on all the relevant anchors.

4. (USER) REQUIREMENTS

This section will discuss the ways in which users would interact with pattern matching in a general visual environment. The specific case for Viskell and the corresponding implementation will be discussed in the next section.

In general, choice is modeled as a node with a condition and multiple edges outwards to other nodes, of which one is selected based on the condition and current state. This pattern is most recognizable in flow charts, where branches usually leave in different directions. Whereas this general approach works well for procedural and dataflow programming, choice in functional programming also requires all the branches to have a well-defined output to return to whichever node asked for the choice. This restricts choice to a fork-and-join pattern in a function VPL.

In cases where program flow has a general direction, parallel computation, e.g. evaluating arguments of a binary function, naturally tends to be laid out perpendicular to the program flow. Similarly, branches must be laid out perpendicular to the program flow, as they share a single entry and exit point and perform independent calculations.

5. APPROACH

This section discusses the design decisions for conditionals.

Firstly, the choice has been made to translate only the structure of pattern guards to Viskell. This means that a function, when compiled to Haskell, will always have one definition with simple parameters. As the program flow in Viskell is oriented vertically, the alternatives in a conditional are layed out horizontally.

Secondly, to reduce wire clutter, a conditional block should allow for input anchors which are copied to each alternative as output anchors. However,

There are multiple ways to visualize pattern matching on values. The main contenders are a conditional block with every

The choice has been made to not make seperate constructs for boolean choices and pattern matching, but to combine them into one construct where the results of expressions can be matched against the `True` constructor.

6. IMPLEMENTATION

6.1 Translation to Haskell

A function block has the following properties for translation to Haskell:

$$
\begin{aligned}
node_{a,out,1} &= var \\
node_{a,lhs} &= node_{a,out,1} \\
node_{a,rhs} &= f\ node_{k_1,out,o_1}\ node_{k_2,out,o_2}\ ..\ node_{k_n,out,o_n}
\end{aligned}
$$

where f is the name of the function.
The lefthand and righthand side are the binding identifier and expression respectively. Within a let expression these will be combined with an equals sign ($=$) and within a guard these will be combined with a leftward arrow ($\leftarrow$).

A deconstructor block or match block has the following properties:

$$
\begin{aligned}
node_{a,out,m} &= var_m\ \forall 1 \leq m \leq n \\
node_{a,lhs} &= s(C\ node_{a,out,0}\ node_{a,out,1}\ ..\ node_{a,out,n}) \\
node_{a,rhs} &= node_{k_1,out,o_1}
\end{aligned}
$$

where C is the name of the constructor and s one of $\{\tilde{},!,\sqcup\}$ to indicate whether a match is strict or not.

The lambda container or function definition has the fol-

lowing properties:

$$
\begin{aligned}
node_{a,out,1} &= var \\
node_{a,lhs} &= node_{a,out,1} \\
node_{a,rhs} &= let\{ \\
&\quad node_{k_1,lhs} = node_{k_1,rhs}; \\
&\quad node_{k_2,lhs} = node_{k_2,rhs}; \\
&\quad ...; \\
&\quad node_{k_n,lhs} = node_{k_n,rhs}; \\
&\quad var_2 = node_{k_n,out,l} \\
&\} \; in \; var_2
\end{aligned}
$$

The container accumulates all the nodes within itself and adds them seperated by semicolons.

The choice block or alternative block has the following properties:

$$
\begin{aligned}
node_{a,out,1} &= var \\
node_{a,lhs} &= node_{a,out,1} \\
node_{a,rhs} &= case \; () \; of \; \{ \\
&\quad alt_{k_1}; alt_{k_2}; ..; alt_{k_n}; \\
&\} \\
\end{aligned}
$$

The block accumulates the expressions of the alternatives within, seperated by semicolons. The primary expression is simply the unit (`()`) and all matching is done within the pattern guards, as shown below.

A lane or alternative within a choice block has the following properties:

$$
\begin{aligned}
alt_a = & ()| \\
& node_{k_1,lhs} \leftarrow node_{k_1,rhs}, \\
& node_{k_2,lhs} \leftarrow node_{k_2,rhs}, \\
& .., \\
& node_{k_n,lhs} \leftarrow node_{k_n,rhs}, \\
& True \rightarrow node_{k_n,out,o_1}
\end{aligned}
$$

Like the lambda container, an alternative accumulates all the internal nodes. However, in this case they are topologically sorted, because identifiers must be defined in a guard before they can be used in an expression.

6.2 Evaluation

7. CONCLUSION

8. FUTURE RECOMMENDATIONS

9. REFERENCES

[1] J. Edwards. No ifs, ands, or buts: Uncovering the simplicity of conditionals. *SIGPLAN Not.*, 42(10):639–658, Oct. 2007.

[2] A. Fukunaga, W. Pree, and T. D. Kimura. Functions as objects in a data flow based visual language. In *Proceedings of the 1993 ACM Conference on Computer Science*, CSC '93, pages 215–220, New York, NY, USA, 1993. ACM.

[3] T. Green. Cognitive dimensions of notations. *A. Sutcliffe & L. Macaulay (1989) People and Computers V*, pages 443–460, 1989.

[4] E. Hosick. Visual Programming Languages - Snapshots. "`http://web.archive.org/web/20150814141754/http://blog.interfacevision.com/design/design-visual-progarmming-languages-snapshots/`", 2014. Accessed: 2015-09-05.

[5] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[6] J. K. W. N. D. S. M. Bruning, K. Hartsuiker. Viskell technical report. Technical report, University of Twente, 2015.

[7] S. Marlow et al. Haskell 2010 language report. *Available online http://www.haskell.org/ (May 2011)*, 2010.

[8] D. Moody. The "physics" of notations: Toward a scientific basis for constructing visual notations in software engineering. *Software Engineering, IEEE Transactions on*, 35(6):756–779, Nov 2009.

[9] B. A. Myers and B. A. Myers. Taxonomies of visual programming, 1989.

[10] H. J. Reekie. Visual haskell: A first attempt. Technical report, 1994.

[11] W. Van Oortmerssen. *Concurrent tree space transformation in the aardappel programming language.* PhD thesis, University of Southampton, 2000.