Visual Functional Programming - An Approach Based On Interaction Nets

Miguel Vilaça

September 12, 2008

Chapter 5

Visual Functional Languages

This chapter presents an overview of the work done in visual functional programming. As some of the features are not specific of the functional setting, occasionally results in the (more general) visual programming area will be mentioned. However, visual programming in its broader sense is outside the scope of this thesis.

The motivation to move from the textual to the visual setting resides in results from cognitive and human brain sciences that state that "for the human brain it is easier and faster to understand and comprehend images than text".

As pointed out in [NPC01], the pictorial aspect of visual languages improves the mental understanding done by a programmer, compared with textual programming. This allows programmers to have a better formulation of the semantic relationships existing in programs, which much facilitates the ability to develop new program patterns.

With this intention in mind, several attempts to define visual programming languages have been and continue to be made, which cover all the main programming paradigms: imperative, object-oriented, logic and functional.

One fundamental feature of functional programming is the presence of higher-order constructs. A function f can take function g as an argument and g can then be applied within the body of f. Expressing this feature is easy if variables are used but some purely visual solutions exist as well, as

will be seen below.

A second difficulty arising from the higher-order nature of functional programs is that a (curried) function of two arguments may receive only its first argument and return as result a function. In a box-based representation this means that it must be possible for a box to lose its input ports one by one – a complicated process.

Work in this area has addressed different aspects of visual programming; while some projects propose new visual programming languages, others simply introduce graphical visualisations of textual languages. The current chapter presents an overview of some of those languages.

5.1 VEX

VEX [CHZ95] is a completely visual representation of the pure untyped λ-calculus. The motivation of its authors was to overcome the difficulties that students have learning aspects of the λ-calculus.

VEX uses containment (boxes inside boxes), geometric placement (boxes place side-by-side for instance for function application) and linking (between boxes) as graphical ingredients.

Application is represented by circles externally tangent to each other and with one arrow pointing the argument. Abstraction is represented by circles containing their body definitions inside, and the abstracted parameters are represented by inner circles tangent to the abstraction circle. For instance, considering the Y combinator defined as the λ-term $\lambda f.\,(\lambda x.f\ (x\,x))\ (\lambda x.f\ (x\,x))$, the λ-term $Y\,e$ is represented in VEX by the visual expression given in Figure 5.1.

The reduction process is mimicked in VEX moving the argument of an application (which is depicted as a circle) to the inner circle of the abstraction (that corresponds to the bound variable). Although this way of mimicking reduction is intuitive, a big disadvantage is that the size of the representation of an expression is changed, making it difficult to track if a sub-expression has been modified or not. Figure 5.2 shows one step of the reduction of the expression $Y\,e$.

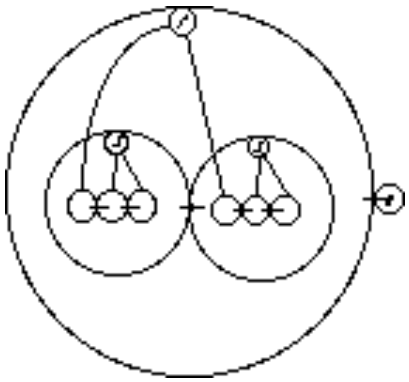

Figure 5.1: VEX expression for $Y\,e$ (taken from [CHZ95])

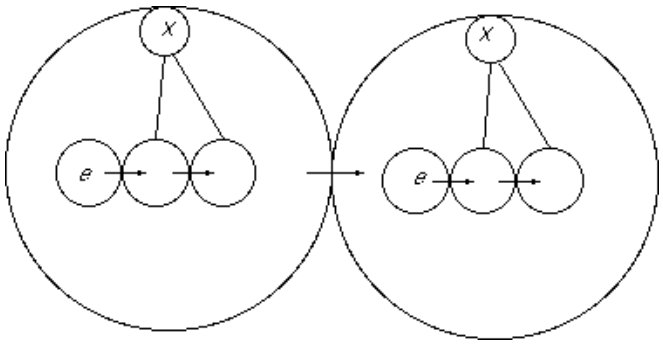

Figure 5.2: VEX expression after reduction of $Y\,e$ (taken from [CHZ95])

5.2 Pivotal

The Pivotal project [Han02] offers a visual notation (and Haskell programming environment) for data structures, but not programs. It is more an interpreter with visualisation capabilities; the user types textual Haskell code in a regular Haskell file. When loaded in the Pivotal interpreter the functional values which the interpreter knows how to display are shown graphically. Other values and program definitions remain only textual. An example of a file loaded in the interpreter is given in Figure 5.3 where it can be seen that *tree* is shown textually and visually.

The interpreter even allows that some of the elements visualised can be graphically edited, and reflects those changes in the textual expression.

Pivotal and Vital (its predecessor) support an useful subset of Haskell.

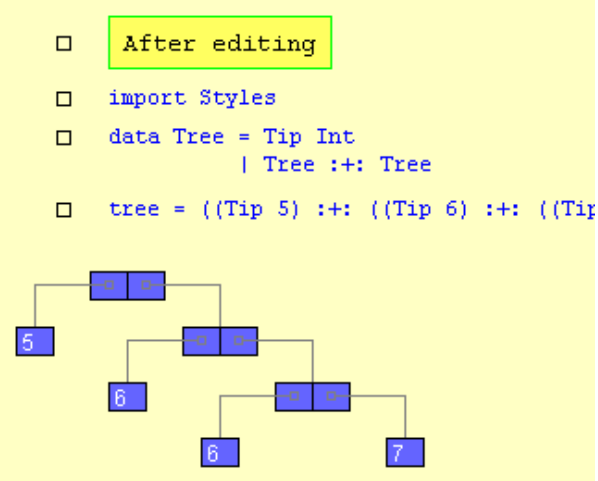

Figure 5.3: Displayed tree allowing editing (taken from [Vit])

Both tools are appealing as visualisers, but their interest is limited to certain pre-defined types, in particular those whose values can be visually edited. Although, the user have the possibility to define how to visualise new datatypes.

They can hardly be considered to be visual programming tools, since they do not allow for programs to be graphically created.

5.3 Visual Haskell

Visual Haskell [Ree95] more or less stands on the opposite side of the spectrum of possibilities when compared with Pivotal: this is a dataflow-style visual notation for Haskell programs, which allows programmers to define their programs visually and then have them translated automatically to Haskell code. It is a two purpose project; on one hand it defines a visualisation tool for a subset of Haskell and on the other hand it intends to be a visual programming language on its own.

In Visual Haskell, functions are represented as boxes with input ports for the arguments and exactly one output port for the result; the contents of each box corresponds to the body of the function. Inside boxes named variables are used to refer to function parameters. As an example, the Visual Haskell

representation of factorial (fact) function is presented in Figure 5.4.

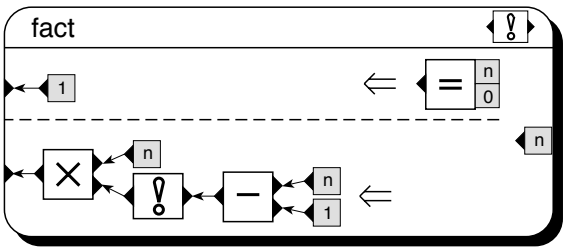

Figure 5.4: Visual Haskell representation of factorial (taken from [Ree95])

As function definitions use variables, functions that receive functions as arguments are accepted, and thus partial support for higher-order programming exists.

The current status of the project is that a prototype visual editor for the language has been produced, but no notion of reduction in Visual Haskell has been implemented yet.

5.4 VFPE

The Visual Functional Programming system [Kel02] is a complete environment that allows functional programs to be defined visually, and then reduced step by step.

VFP uses a notation without boxes, more inspired by the traditional representations of functional programs used in implementation-oriented abstract machines [Jon87]. In particular, it allows for named functions and variables (used for arguments, as in Visual Haskell). An example program is given in Figure 5.5.

VFP supports a wide part of the Haskell syntax tree. Since explicit abstraction and application nodes are used, higher-order programming is supported.

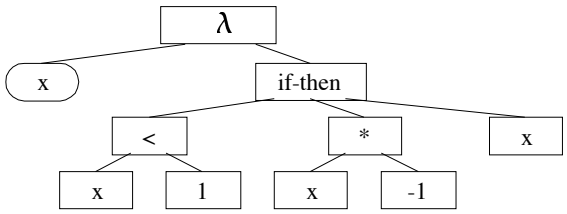

Figure 5.5: VFPE example (taken from [Kel02])

5.5 Visual Lambda

VisualLambda [DV96] is a formalism based on graph-rewriting: programs are represented as a kind of hierarchical graphs whose reduction mimics the execution of a functional program. Functions are represented by boxes with input ports for the arguments and exactly one output port for the result. As in Visual Haskell, the contents of the box correspond to the body of the function, although the notations differ in that Visual Haskell uses named variables to refer to function arguments while in VisualLambda a purely visual notation with arrows is used.

One interesting aspect of this language is the purely visual treatment of higher-order notions; in VisualLambda a special box (depicted in grey) would be used as a placeholder for a function g which is an argument of f (in the body of f) to be instantiated later, and an arrow would link an input port in the box of f to the box of g. Figure 5.6 exemplifies this grey box.

As an example the λ-term $(\lambda f.\lambda x.f\ (f\ x))\ (\lambda x.x + 1)\ 23$ is represented as in Figure 5.7.

As a full language, it has a notion of reduction defined inside the formalism, which is implemented by the VisualLambda tool.

5.6 Final Remarks

VEX is a visual language with a notion of reduction but it is restricted to the λ-calculus. The Pivotal project offers a visual notation (and Haskell programming environment) for data-structures, but not programs. Visual

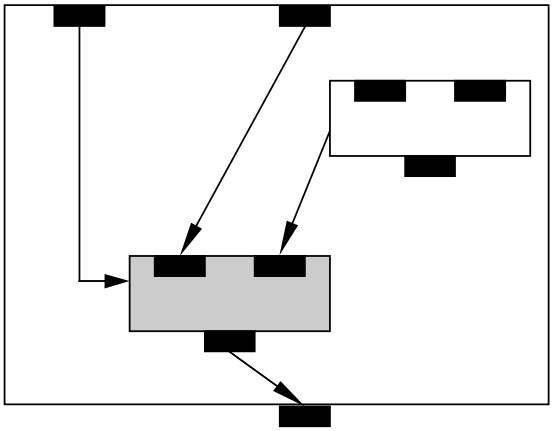

Figure 5.6: VisualLambda example of grey box (taken from [DV96])

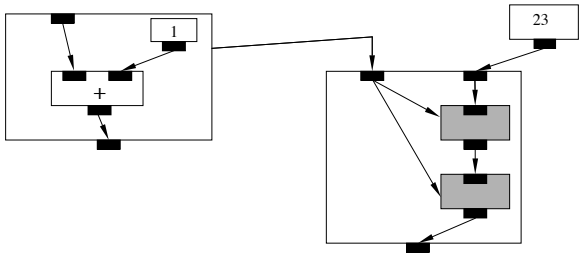

Figure 5.7: VisualLambda example for λ-term $(\lambda f.\lambda x.f\ (f\ x))\ (\lambda x.x + 1)\ 23$ (taken from [DV96])

Haskell more or less stands at the opposite side of the spectrum of possibilities: this is a dataflow-style visual notation for Haskell programs, which allows programmers to define their programs visually (with the assistance of a tool) and then have them translated automatically to Haskell code. Kelso's VFP system is a complete environment that allows to define functional programs visually and then reduce them step by step. Finally, VisualLambda is a formalism based on graph-rewriting: programs are defined as graphs whose reduction mimics the execution of a functional program.

Visual Haskell and VisualLambda have in common the fact that functions are represented as boxes with input ports for the arguments and an output port for the result. They differ in that Visual Haskell uses named variables

to refer to function arguments, while VisualLambda uses a graphical notation based on arrows. VFP uses a notation without boxes, inspired by the representations used in implementation-oriented graph-rewriting machines. In particular, it allows for named functions but also for λ-abstractions, and an explicit application node exists. Variables are used for arguments, as in Visual Haskell.

In what concerns higher-order, Visual Haskell and VFP support it through the use of variables; in VisualLambda a special box would be used as a placeholder for g (in the body of f) to be instantiated later.

As far as we know none of these systems is widely used in practice.