Visual Haskell: A First Attempt*

H. John Reekie
School of Electrical Engineering
University of Technology, Sydney

email: johnr@ee.uts.edu.au

Abstract

This paper presents the Visual Haskell language. Visual Haskell is intended as i) a means of visualising Haskell programs, and ii) as a complementary programming notation to standard Haskell. The syntax of Visual Haskell is specified by first giving a translation from Haskell into an intermediate, textual form; the visual syntax is then specified as a direct translation from the intermediate form into visual representations. Visual Haskell is kept as close to Haskell as possible, in the hope that it could be used in a "two-view" programming system. Several examples of function definitions illustrate how Visual Haskell looks and is used.

Visual Haskell is incomplete, lacking syntax for modules, type declarations, type classes and instances, arrays, and user-defined operators. These are seen as temporary omissions rather than insurmountable drawbacks.

1 Introduction

This paper presents Visual Haskell, my attempt to develop a visual equivalent for the Haskell functional language [4]. Visual Haskell is intended both as a program visualisation tool and as a programming language in its own right. In both of these roles, it should be seen as complementary to Haskell's standard textual form, rather than as a replacement or "improved" notation.

Visual Haskell evolved out of attempts during my Ph.D. research to explain my work to non-functional-programmers. This work involved the application of functional programming (using Haskell) to digital signal processing—in particular, the ways in which functional languages lend themselves to expression and transformation of dataflow-style programs [12].

In signal processing, dataflow-style visual languages have been used as formal programming notations for some years [1, 7]. These systems are generally know as "block-diagram" programming systems, because of their roots in informal diagrammatic representations of signal processing systems. Dataflow visual languages have been used successfully in other fields as well, such as image processing and instrumentation [9]. The success of these systems indicates very strongly that visual programming—if applied to appropriate problem domains and supported by appropriate tools—offers a kind of understanding that is absent in purely-textual languages.

* Published as: Research Report 94.5, Key Centre for Advanced Computing Sciences, University of Technology, Sydney, PO Box 123, Broadway NSW 2007, Australia, August 1994.

Visual Haskell, then, is my effort at bringing together Haskell and the visual programming paradigm. The approach I have taken is to define i) a translation from Haskell into an intermediate form, called VHIF (Visual Haskell Intermediate Form), and ii) visual equivalents for each syntactic production of VHIF. I have kept the syntax of VHIF as close to Haskell as possible, in the hope that the correspondence between visual and textual Haskell programs can be kept as "intuitive" as possible.

I do not claim that visual programming is suitable for all (or even most) ends; nor do I claim that Visual Haskell is a better "block-diagram" language than those offered by existing systems. However, I do believe that Visual Haskell—or something like it—offers some unique advantages:

- Haskell has a number of powerful features not found in visual dataflow systems: polymorphism and higher-order functions are two well-established features of modern functional languages that I have found useful for modelling signal processing systems.

- It is likely to be easier to precisely define the semantics of a visual language in terms of Haskell's semantics, than attempting to define a new semantics from scratch.

- One can easily imagine a "two-view" development system [13], in which either visual or textual notation can be used by the programmer to write or view a function or expression. The choice is made by the programmer, based solely on which view is most descriptive of the program fragment at hand.

- Execution of Visual Haskell programs can take advantage of work on standard Haskell compilers and interpreters; all that is required is a translator from visual to textual syntax.

In its current form, Visual Haskell provides an alternate representation for part of Haskell (notable omissions are class and type declarations). Even so, Visual Haskell is sufficiently well-developed to be useful—its uses in program visualisation, and the prototype graphical program editor, are described in the next section. The language is still evolving; this paper presents a "snapshot" of Visual Haskell as it is now, so that other researchers can use and criticise it.

1.1 A motivating example

I will use the standard prelude function *map* to illustrate the key features of Visual Haskell: Figure 1 is the standard textual definition, while Figure 2 is the Visual Haskell definition. It is immediately apparent that Visual Haskell is a "box-and-arrow" visual language: function application is represented by distinctive graphics, and data flow by directed arcs flowing between triangular "pads." This function has an icon defined for it, displayed next to the "map" title.

The pads on the outside of the main enclosing box indicate that the function has two inputs (and, of course, one output). There are two clauses in this definition, stacked one above the other. Within each clause, the argument patterns are shown immediately inside the enclosing box—in this case, we have a variable pattern (f) a constant pattern ([]), and a list constructor pattern ($x : xs$). The icon for the list constructor pattern (the long thin box) is the same as the icon for constructor application, shown towards the left of the bottom clause.

```
map           :: (a -> b) -> [a] -> [b]
map f []       = []
map f (x:xs) = f x : map f xs
```

Figure 1. The Haskell definition of *map*

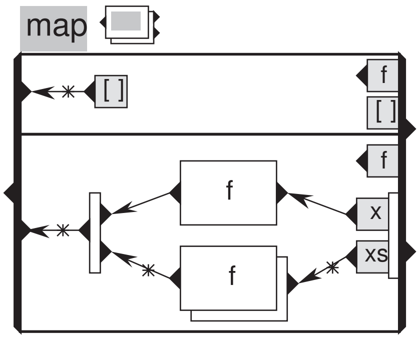


Figure 2. The Visual Haskell definition of *map*

By default, an applied function is shown as a plain box labeled with the function name, as in the application of f. To make visual programs more meaningful, however, we can define icons for specific functions, as done here for *map*. This particular icon has an "argument slot" (see section 3.4) containing the first argument (the function f). (The "icons" I use are rather plain for ease of drawing—more interesting icons would be used in a real graphical development system.)

Notice that list-carrying arcs are decorated with an asterisk-like annotation. These annotations are a useful visual aid to program understanding, although I have not yet been able to formalise their representation.

Incidentally, I am in the habit of drawing Visual Haskell expressions so that the data flow is from right to left. The reason is simple: this is the same direction in which data "flows" in textual programs. For example, in $f \cdot g \cdot h$, data "flows" through h, g, and then f. Visual Haskell, however, does not care about the direction, and all of the diagrams in this paper could be mirrored without any change in meaning.

1.2 Related work

Visual notations of the "box-and-arrow" style are often used to illustrate or explain functional programs: Reade [10], for example, explains function application using diagrams of this kind; Kelly [6] illustrates networks of streams and processes written using an annotated pure functional language; Waugh *et al* [15] illustrate the effect of program transformation on parallelism. However, these diagrams serve an expository purpose and are not programming notations in themselves. (Other types of diagram are used to explain functional language implementations—for example, [5]—but these are not our concern here.)

In other fields, diagrams have been used as formal programming notations for some time. Signal processing, for example, makes extensive use of the metaphor of modular units interconnected by signals (that is, streams). The Ptolemy system is an excellent example of this type of system [7], and a number of systems based on this idea have been successful, both in signal processing and in other fields such as image processing and instrumentation [9].

In the field of visual language research, Visual Haskell is perhaps closest to that of Najork and Golin [8] on Show-and-Tell. Show-and-Tell is also declarative in nature, and utilises a polymorphic type system similar to Haskell's. Visual Haskell's argument slots were inspired by Show-and-Tell's typing slots—their type notation would be a good starting-point for Visual Haskell's missing typing notation.

2 Examples of Visual Haskell

The ultimate goals of Visual Haskell are:

1. To provide a visual equivalent for all of standard Haskell, so that any Haskell program can be represented either textually or visually.

2. To keep the notation simple and uncluttered, since (formal) visual program notations can easily become cluttered and obscure program meaning. The fact that Haskell programs are quite concise anyway has certainly helped this goal.

3. To make the concrete visual syntax extensible, so a programmer can define new graphical elements for application-specific datatypes and functions. This is important to give programs instantly-recognisable meaning in particular application domains—signal processing, for example.

In this section, I give examples of the use of Visual Haskell, to illustrate how Visual Haskell can be used, and to what degree Visual Haskell as it currently stands fulfills these goals. I have attempted to include examples that illustrate short-comings and incomplete parts of Visual Haskell, as well as favourable examples.

2.1 Illustrating program translation

Visual Haskell uses similar visual elements for program constructs with similar purpose, even though standard Haskell may use different syntax. This can often highlight similarities between programs that are less obvious in a textual representation.

For example, consider the translation of a function to the enriched lambda calculus, as described in [5]. Translation of *map* (Figure 1) into a single λ-abstraction using *case*-expressions gives:

```
map = \f xs -> case xs of
                 []   -> []
                 y:ys -> f y : map f ys
```

Figure 3 shows this definition in Visual Haskell. The similarities to the original definition (Figure 2) are, I think, clearer than with the textual forms.

Taking an additional step in the translation, we can translate the *case*-expression to give

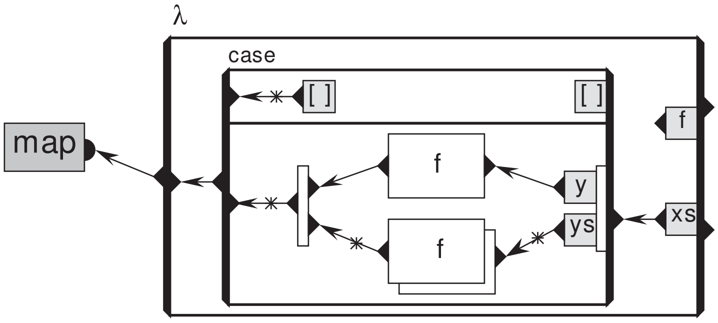


Figure 3. *map* translated using a *case*-expression

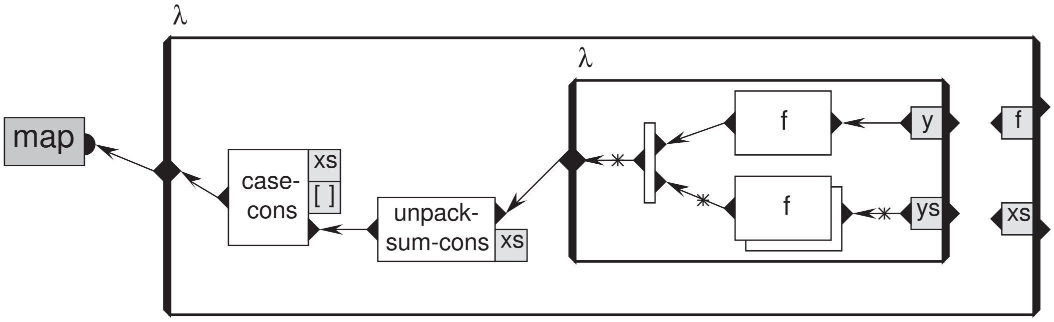


Figure 4. *map* after removing the *case*-expression

```
map = \f xs -> CASE-CONS
                 xs
                 []
                 UNPACK-SUM-CONS (\y ys -> CONS (f y) (map f ys)) xs
```

Figure 4 shows this definition; in this case, the visual representation does not give much improvement in understanding.

2.2 Illustrating process networks

A visual representation is particularly useful for programs that can be conceived of in a *structural* way, such as networks of communicating processes. Kelly [6] and Waugh *et al* [15], for example, use diagrams to illustrate process networks and the effect of program transformation on them.

The work referred to above uses lists to model streams—that is, communications channels between processes. I prefer to use a distinct datatype, *Stream*, to more clearly distinguish streams from other kinds of lists. I also use stream versions of higher-order functions such as *map*, called *smap* and so on. In Visual Haskell, streams are indicated by circular type annotations; functions such as *smap* have circular icons.

Consider a function that performs FFTs (Fast Fourier Transforms) on successive 16-sample sections of a signal xs. (This function and the following code fragments are considered in detail in [11].) Each element of the output stream is a vector containing the complex spectrum of a 16-sample portion of the real input signal:[1]

```
fftnet :: (Num a) => Stream a -> Stream (Vector (Complex a))
fftnet = smap fft . chop 16 . smap (\x -> x :+ 0)
```

This function contains three processes, as should be clear from Figure 5a, which shows the right-hand side of the definition only. To increase the degree of parallelism, we can apply algebraic program transformation techniques, such as described by Bird [2]. One way to do so is to distribute FFTs to different processors:

```
fftnet' = interleave . map (smap fft) . distribute 4
          . chop 16 . smap (\x -> x :+ 0)
```

where *distribute p* generates a list of streams, and *interleave* combines elements from a list of streams into a single stream. This resulting program is like a restricted form of "processor farm," and is illustrated in Figure 5b.

A slightly more sophisticated attempt to increase parallelism converts the FFT into a pipeline, thus:

```
fftnet'' = smap bitrev
           . smap (stage 2) . smap (stage 4) . smap (stage 8) . smap (stage 16)
           . chop 16 . smap (\x -> x :+ 0)
```

The pipeline can clearly be seen in Figure 5c.

This example highlights a difficulty with visual representations: I resorted to giving values of variables which really should be arguments to the various version of *fftnet*—the size of the FFT (16), and the number of parallel streams (4). Without knowing these values, it is not possible to produce diagrams such as Figure 5b and Figure 5c. In section 5.4, I suggest one way in which these kinds of diagrams could be produced without requiring known values.

(Note that Visual Haskell can easily represent pipelined FFTs of arbitrary length without difficulty; the difficulty is in "unfolding" these diagrams to illustrate the *structure* of the pipelined or parallel processes.)

2.3 Standard prelude examples

We have already seen the Visual Haskell representation of the standard prelude function *map*. This section shows some other standard prelude functions.

2.3.1 $(+)_{Complex\ a}$

The addition function for the *Complex* type is a simple example which works well. However, it shows up the lack of type and class syntax in Visual Haskell. For now, I use a subscript to indicate the type to which an overloaded function belongs. Figure 6 shows the Visual Haskell version of $(+)_{Complex\ a}$.

$(+)_{Complex\ a}$ is defined by:

```
(x:+y) + (x':+y') = (x+x') :+ (y+y')
```

[1] I should point out that this is a simplified example, and will give neither good spectral estimation, nor an efficient realisation.

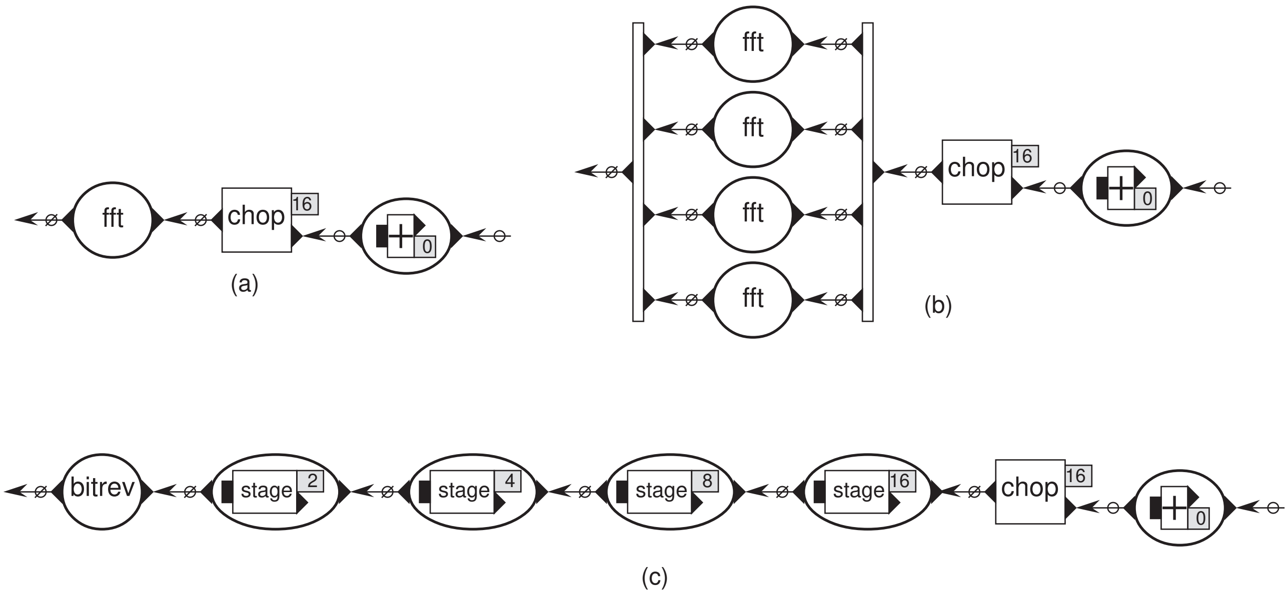


Figure 5. Examples of process networks

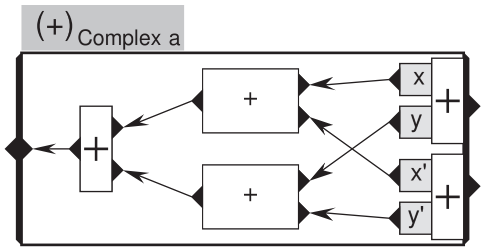


Figure 6. $(+)_{Complex\ a}$

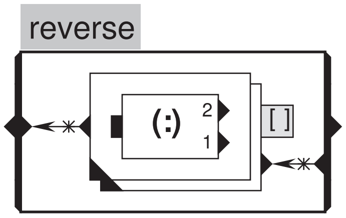


Figure 7. *reverse*

2.3.2 *reverse*

reverse (Figure 7) demonstrates that, even with "invisible" patterns, the augmenting pattern rules, and permuted application (these things are described in section 4 and section 5), some functions just do not lend themselves to visual representation. *reverse* is defined as:

```
reverse     :: [a] -> [a]
reverse     = foldl (flip (:)) []
```

2.3.3 (^)

The power operator, (^), illustrates nested function definitions quite well. It is also a function for which a visual representation provides little aid to understanding. Figure 8 shows the visual representation. The textual definition is

```
(^)         :: (Num a, Integral b) => a -> b -> a
x ^ 0       = 1
x ^ (n+1)   = f x n x
              where
                f _ 0 y = y
                f x n y = g x n
                    where
                    g x n | even n    = g (x*x) (n 'quot' 2)
                          | otherwise = f x (n-1) (x*y)
```

2.4 A Visual Haskell editor

Ken Dawson has constructed a prototype graphical editor for Visual Haskell, to explore the feasibility of using Visual Haskell to write programs [3]. The prototype editor allows the user to write simple function definitions in Visual Haskell, and then generate (textual) Haskell code for compilation. Although the functionality of the editor is limited, it is adequate to define functions such as *map*. Figure 9 shows the screen after using the editor to draw the definition of *map* and asking the editor to generate the equivalent textual definition (the screen bitmap was kindly supplied by Ken Dawson). The prototype system should provide valuable feedback on whether Visual Haskell really is useable as a programming notation. It will also be interesting to see how programs written in Visual Haskell differ in style from programs written in standard Haskell.

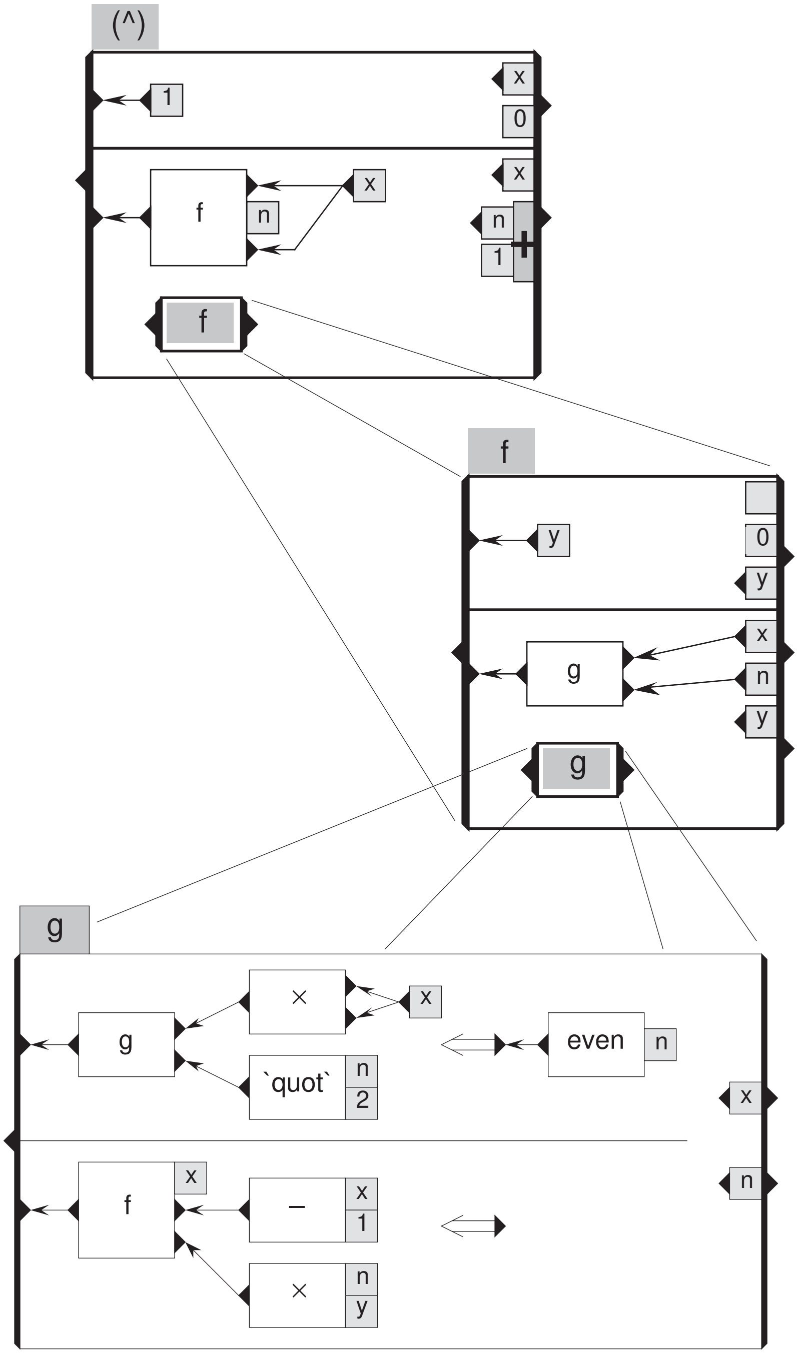


Figure 8. (ˆ)

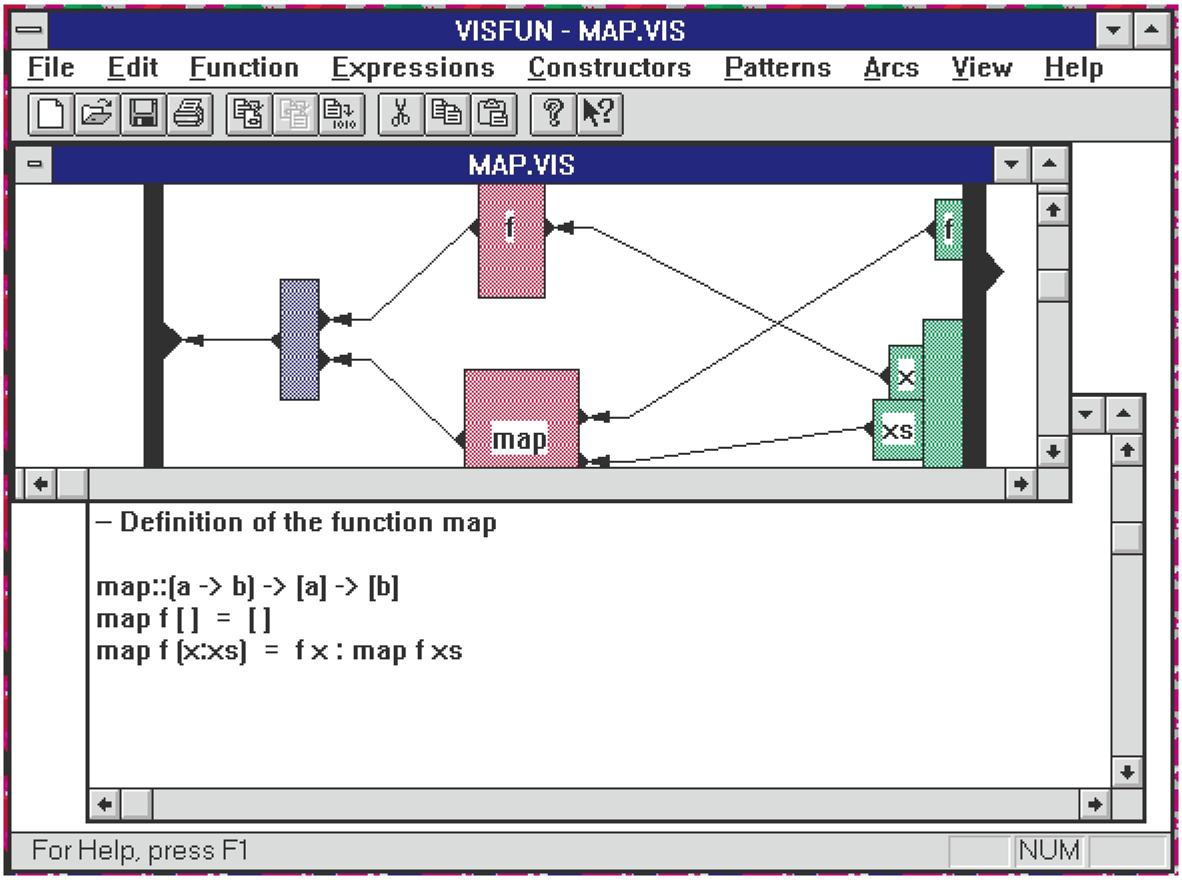


Figure 9. A screen dump of the prototype editor

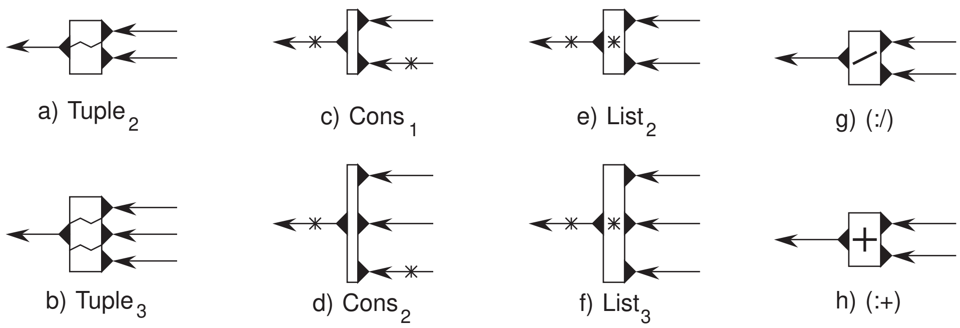


Figure 10. Icons for some standard prelude data constructors

3 The Visual Representation of Programs

There is a certain tension between the goals of precise representation of Haskell programs, and uncluttered visual programs. Visual Haskell attempts to resolve this conflict by distinguishing between levels of (concrete) syntax. The *core syntax* describes a canonic visual representation that serves to specify the essential visual constructs required to represent a program. This gives Visual Haskell programs a precise correspondence with Haskell programs. The *extended syntax* allows a given syntactic production to take on different appearances, thus allowing visual programs to take on forms that lend themselves to "intuitive" understanding. The extended syntax is also the mechanism by which a user of a Visual Haskell development environment would tailor programs to suit a particular application domain—by creating suggestive icons for application of certain functions, for example. The extended syntax could be viewed as visual "syntactic sugar."

As an example of extended syntax, an arc could be displayed as a straight line, a curved line, or a series of perpendicular lines. It could even be broken, if it is impossible to route the arc without crossing other arcs. There are two types of extended syntax:

Static A static syntax, once defined, is always used. The icon for the *map* function is an example—this icon is always used when *map* is applied.

Dynamic A dynamic syntax can be selected by the programmer for each circumstance in which alternatives arise—for example, the way a particular arc is displayed.

The remainder of this section presents the syntax of Visual Haskell in an informal manner. The formal syntax will be presented in section 4.

3.1 Standard icons

Figure 10 shows the icons I use for some common data constructors (constructors such as $Tuple_2$ are generated by translation rules of Figure 23.) These icons are used both for application and in patterns; I have shown connecting arcs as they would appear in an application.

Figure 11 shows icons for several of the standard prelude list functions. The grey boxes in the higher-order function icons are argument slots (see section 3.4). Note that the input and output pads will change sides if data flow is drawn from left to right, but the icons themselves will not change.

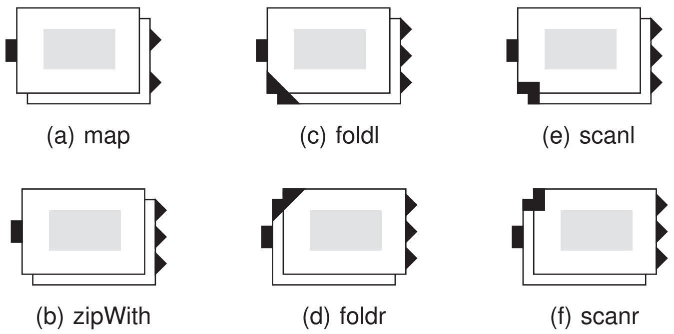


Figure 11. Icons for some standard prelude functions

Figure 12. Source and sink pads

3.2 Aggregation, connection, and attachment

A visual program (or rather, module) is a collection of visual objects. Visual objects can be related to other visual objects in three ways:

1. Aggregation

   Aggregation is a rather loose term used to describe a group of objects which can be treated as an atomic unit. Usually, an aggregation corresponds to a syntactic production of VHIF. Examples of aggregations are a function application icon and its input and output pads, a *let*-expression including output pad and all internal objects, and a function definition.

2. Connection

   Two objects are connected by drawing an arc from a source pad of one object to a sink pad of the other. Figure 12 shows source and sink pads. There are two types of arc: data arcs are used for function application, and are drawn with an arrow (Figure 13a); binding arcs are used in patterns, and are drawn without an arrow (Figure 13b).

3. Attachment

   Two objects are "attached" if they are physically located such that at least one point of each is at the same physical location (Figure 13c).

   Attachment is often a special form of connection: if an arc has zero length, then the two objects it connects are attached and the pads become invisible. Two types of arcs can have zero length: (a) data arcs emanating from a variable or constant, and (b) all binding arcs.

3.3 Shared objects

In the core syntax, each occurrence of a variable or named pattern has a unique box; each such box has one arc connected to its output pad. For example, Figure 14a shows the expression

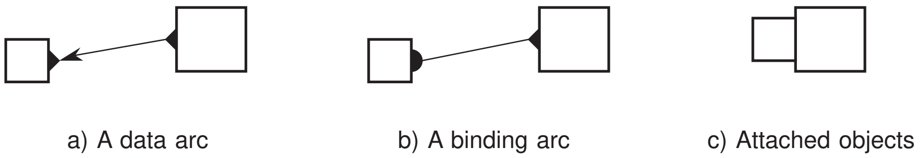


Figure 13. Connection and attachment

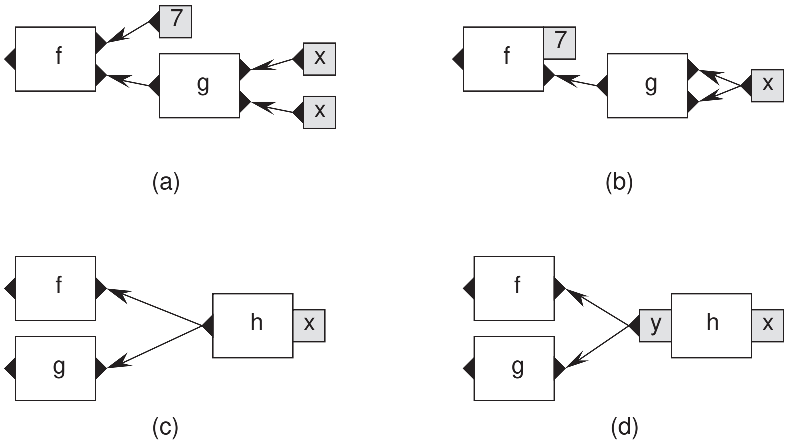


Figure 14. Illustrating sharing and zero-length arcs

$f\ 7\ (g\ x\ x)$ in core syntax. In the extended syntax, data arcs connected to the output of a variable, constant, or named pattern, can have zero length; in effect, the two objects on either end of the arc become attached. For example, in Figure 14b, the constant 7 is attached to f.

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 14b, the variable x is shared; in Figure 2, the patterns x and xs are shared.

Sharing is subject to some constraints on arc placement (see section 4.6 for the visual syntax of scopes and clauses):

– An arc cannot enter or exit a clause,

– No arc can connect to the patterns of a guarded clause,

– An arc cannot go from one guard or expression of a guarded clause to another.

Unnamed function results cannot be shared. For example, Figure 14c is an invalid expression; it can be corrected by naming the function result, as in Figure 14d. (This restriction is imposed by the need for a correspondence between Haskell and Visual Haskell. It may be possible to work around it with "invisible" variables (section 4.5.1)).

3.4 Argument slots

Any function application icon can have one or more argument slots. If an actual supplied to a pad which has a slot is "able to fit into" the slot, then the actual is displayed in the slot and the corresponding pad is not displayed.

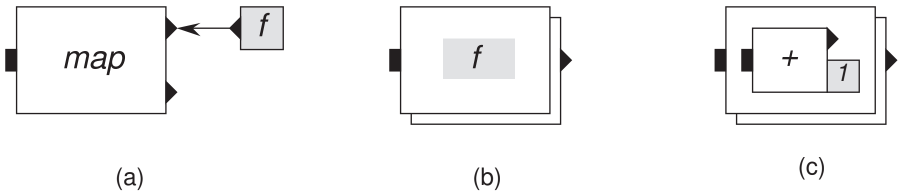


Figure 15. Illustrating argument slots

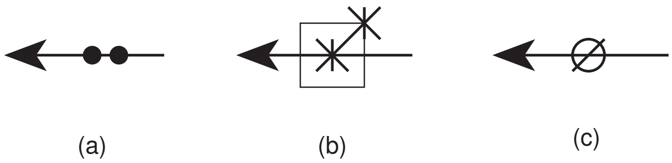


Figure 16. Illustrating type annotations: a) `(a,b)`; b) `[[a]]`; c) `Stream (Vector a)`

Figure 15a shows the application *map f* in core syntax; Figure 15b shows the application using *map*'s icon and with the name of the argument, *f*, placed into the icon's slot. Larger arguments can also fit, Figure 15c shows *map* (+1). Section 5.3 suggests an extension of this idea useful for illustrating program transformation.

3.5 Type annotations

Arcs are optionally decorated to indicate their types. In this paper, only list-carrying arcs are so decorated, with an asterisk-like symbol. In other work [11] I have used a more extensive set of type annotations. Figure 16 illustrates a few annotated arcs to give the flavour of annotations.

A second kind of annotation is used to indicate function values: the output pad of an expression producing a function-valued result is rectangular rather than triangular, as shown in Figure 15. I have found this variation to be useful, but have not incorporated it into the formal syntax.

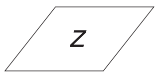
4 The Visual Syntax

The syntax of Visual Haskell is specified in two stages:

1. By a translation from Haskell into the Visual Haskell intermediate form (VHIF)

   The translation is given by syntactic translation rules in the style of [5]. The translations are named $\mathcal{V}$ (for definitions), $\mathcal{VE}$ (for expressions), and $\mathcal{VP}$ (for patterns).

2. By concrete visual syntax for each production of VHIF. The parallelogram in

   represents a non-terminal called z. All other visual elements are terminals. Two visual meta-functions are also used:

   – *name*(x) gives the name of object x as a displayable object.

$$\begin{aligned} \mathcal{VE}[\![\, v \,]\!] &\equiv v && (1)\\ \mathcal{VE}[\![\, k \,]\!] &\equiv k && (2)\\ \mathcal{VE}[\![\ (\ e\ )\ ]\!] &\equiv (\ \mathcal{VE}[\![\, e \,]\!]\ ) && (3) \end{aligned}$$

Figure 17. Translation rules for variables, constants, and parenthesised expressions

$exp \rightarrow var \mid const$
where there is an icon defined for
var or *const*

$exp \rightarrow var \mid const$
where there is no icon defined for
var or *const*

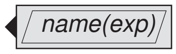
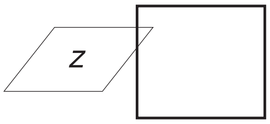


Figure 18. Visual syntax for variables and constants

– $icon(x)$ gives the displayable icon of object x.

An overlapping of a (visual) non-terminal with another visual object indicates that the two objects are attached. For example, z is attached to the box in



4.1 Variables and constants

The concrete syntax for a variable, ground constant, or string, is either an icon, or a shaded box containing the variable or constant's "name." The name of variables is the string representation of its identifier; the name of numeric constants is the string representation of its value. There are a number of special constants: `()` has the name "()"; `[]` has the name "[ ]"; _ has the name " ".

Variables and constants have one output pad. Figure 17 lists the translation rules for variables and constants; Figure 18 shows the visual syntax.

4.2 Function application

In Visual Haskell, application of a function of arity n causes all n input pads to be shown, even if not all are connected. Each function application graphic has one or more input pads and one output pad.

Figure 19 lists translation rules for function application; these ensure that every application is of one of the functions $apply_n$ or $applyp_{is}$, where $apply_n$ applies a function of arity n to $m \leq n$ arguments (see the next section for $applyp$). This tactic is used to make arity information explicit in VHIF, so the correct number of input pads can be displayed.

$$
\begin{aligned}
\mathcal{VE}[\![e_0 \cdots e_m]\!] &\equiv \texttt{apply}_\texttt{n}\ \mathcal{VE}[\![e_0]\!] \cdots \mathcal{VE}[\![e_m]\!] && (4)\\
&\quad \text{if } \mathcal{VE}[\![e_0]\!] \text{ is a } \lambda\text{-abstraction, variable, or data constructor of arity}\\
&\quad n, \text{ and } e_0 \text{ has exactly } m \text{ arguments supplied}\\
&\equiv \texttt{apply}_\texttt{n+1}\ \texttt{apply}_\texttt{n}\ \mathcal{VE}[\![e_0]\!] \cdots \mathcal{VE}[\![e_m]\!] && (5)\\
&\quad \text{if } \mathcal{VE}[\![e_0]\!] \text{ is not a } \lambda\text{-abstraction, variable, or data constructor,}\\
&\quad \mathcal{VE}[\![e_0]\!] \text{ has arity } n, \text{ and } e_0 \text{ has exactly } m \text{ arguments supplied}\\
\mathcal{VE}[\![e_1\ op\ e_2]\!] &\equiv \texttt{apply}_2\ op\ \mathcal{VE}[\![e_1]\!]\ \mathcal{VE}[\![e_2]\!] && (6)\\
\mathcal{VE}[\![\texttt{(}\ e\ op\ \texttt{)}]\!] &\equiv \texttt{(}\ \texttt{apply}_2\ op\ \mathcal{VE}[\![e]\!]\ \texttt{)} && (7)\\
\mathcal{VE}[\![\texttt{(}\ op\ e\ \texttt{)}]\!] &\equiv \texttt{(}\ \texttt{applyp}_{[2,1]}\ op\ \mathcal{VE}[\![e]\!]\ \texttt{)} && (8)\\
\mathcal{VE}[\![\texttt{-}e]\!] &\equiv \texttt{apply}_1\ \texttt{-}\ \mathcal{VE}[\![e]\!] && (9)\\
\mathcal{VE}[\![\texttt{flip}\ e\ e_1\ e_2]\!] &\equiv \texttt{apply}_2\ \mathcal{VE}[\![e]\!]\ \mathcal{VE}[\![e_2]\!]\ \mathcal{VE}[\![e_1]\!] && (10)\\
\mathcal{VE}[\![\texttt{flip}\ e\ e_1]\!] &\equiv \texttt{applyp}_{[2,1]}\ \mathcal{VE}[\![e]\!]\ \mathcal{VE}[\![e_1]\!] && (11)\\
\mathcal{VE}[\![\texttt{flip}\ e]\!] &\equiv \texttt{applyp}_{[2,1]}\ \mathcal{VE}[\![e]\!] && (12)
\end{aligned}
$$

Figure 19. Translation rules for function application

Figure 20 shows the visual syntax for function application. The kernel of an application, $\texttt{apply}_\texttt{n}\ e_0$, has been factored out to account for three possible situations. Note that rule 5 ensures that e_0 *must* be a function or data constructor name, or a λ-abstraction. Notice also that operators are not parenthesised by the translation rules (rule 6). This ensures that Visual Haskell maintains the distinction between the expressions $x + y$ and $(+)\ x\ y$.

We have already seen simple examples of function application. Figure 21 shows some more complex examples, which should clarify the purpose of rule 4 and rule 5. Note that the output pads in these examples use the "function-carrying" pad shape—this difference is not made explicit in the translation rules.

4.3 Permuted application order

In Visual Haskell, it is easy to apply arguments in any order. The family of permuted application functions $applyp_{is}$ apply a function f of arity $n = |is|$ to $m \leq n$ arguments, where $[i_1, \ldots, i_n] = is$ and the output pad of the j'th argument is connected to input pad i_j of f. Permuted application is built into the translation rules for operator sections and the $flip$ function. In a graphical development environment, permuted application would, I expect, be used much more than in standard Haskell, because the visual representation does not impose an order of arguments in the way a textual representation does.

The function application graphics for $applyp_{is}$ are the same as for standard function application, with numeric labels attached to indicate the order of argument application. The labels can be omitted if $m = n$ or $m = n - 1$.

4.4 Data constructors

Constant data constructors are treated like constants; non-constant data constructors are treated like functions.

$exp \rightarrow app_n\ e_1 \cdots e_m$
where the output pad of e_i is joined to input pad i of app_n. (If $m < n$, some input pads are left open.)

$app_n \rightarrow \mathtt{apply_n}\ e_0$
where e_0 is a λ-abstraction.

$app_n \rightarrow \mathtt{apply_n}\ e_0$
where e_0 is a function name and e_0 has an icon.

$app_n \rightarrow \mathtt{apply_n}\ e_0$
where e_0 is a function name and e_0 has no icon.

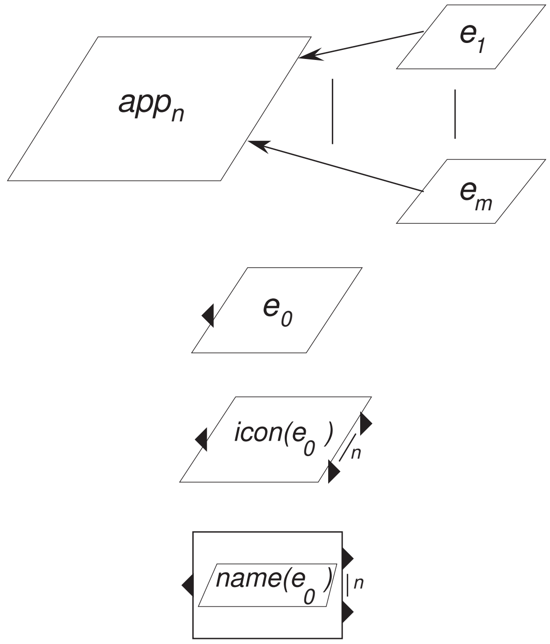


Figure 20. Visual syntax for function application

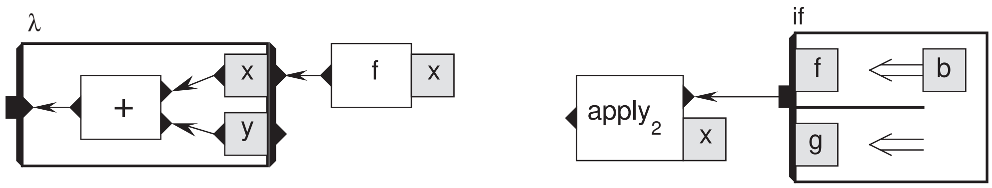
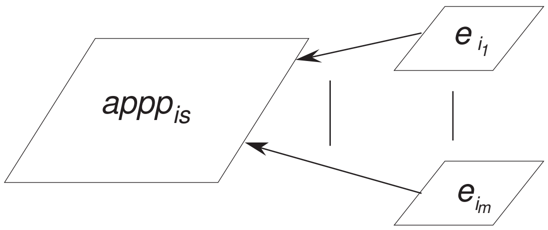
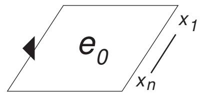


Figure 21. Examples of function application: a) `(\ x y -> x + y) (f x)`; b) `(if b then f else g) x`

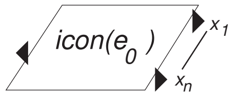
$exp \rightarrow appp_{is}\ e_1 \cdots e_m$
where $[i_1, \cdots, i_n] = is$ and the output pad of e_j is joined to input pad i_j of $appp_{is}$. (If $m < n$, some input pads are left open.)



$appp_{is} \rightarrow \mathtt{applyp_{is}}\ e_0$
where e_0 is a λ-abstraction and $x_k = name(j)$ where $i_j = k$.



$appp_{is} \rightarrow \mathtt{applyp_{is}}\ e_0$
where e_0 is a function name and e_0 has an icon and $x_k = name(j)$ where $i_j = k$.



$appp_{is} \rightarrow \mathtt{applyp_{is}}\ e_0$
where e_0 is a function name and e_0 has no icon and $x_k = name(j)$ where $i_j = k$.

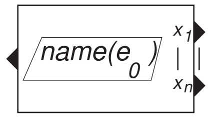


Figure 22. Visual syntax for permuted function application

$$\mathcal{VE}[\![\, e_1 \texttt{:} \cdots \texttt{:} e_n \texttt{:} e \,]\!] \equiv \mathtt{apply_{n+1}}\ \mathtt{Cons_n}\ \mathcal{VE}[\![\, e_1 \,]\!] \cdots \mathcal{VE}[\![\, e_n \,]\!]\ \mathcal{VE}[\![\, e \,]\!] \tag{13}$$

$$\mathcal{VE}[\![\ \texttt{(}\ e_1 \texttt{,} \cdots \texttt{,} e_n\ \texttt{)}\ ]\!] \equiv \mathtt{apply_n}\ \mathtt{Tuple_n}\ \mathcal{VE}[\![\, e_1 \,]\!] \cdots \mathcal{VE}[\![\, e_n \,]\!] \tag{14}$$

$$\mathcal{VE}[\![\ \texttt{[}\ e_1 \texttt{,} \cdots \texttt{,} e_n\ \texttt{]}\ ]\!] \equiv \mathtt{apply_n}\ \mathtt{List_n}\ \mathcal{VE}[\![\, e_1 \,]\!] \cdots \mathcal{VE}[\![\, e_n \,]\!] \tag{15}$$

Figure 23. Special translation rules for data constructors

Haskell's syntax for tuples and lists is translated into application of one of the $Tuple_n$, $Cons_n$, or $List_n$ series of constructors. See Figure 23.

4.5 Patterns

Visually, a pattern is rather like the corresponding expression, but connected "backwards." Every pattern has one binding pad, which looks like this:

(A binding pad is a sink pad.) Patterns have one or more output (source) pads—indeed, patterns are the only kind of object which can have more than one output pad. Patterns also have a naming pad, a special pad used for *as*-patterns. Figure 24 lists translation rules for patterns; Figure 25 gives the visual syntax. Figure 26 gives some examples of patterns.

$$
\begin{aligned}
\mathcal{VP}[\![\, v \,]\!] &\equiv v && (16)\\
\mathcal{VP}[\![\, k \,]\!] &\equiv k && (17)\\
\mathcal{VP}[\![\, \$v \,]\!] &\equiv \$v && (18)\\
\mathcal{VP}[\![\, \$\$ \,]\!] &\equiv \$\$ && (19)\\
\mathcal{VP}[\![\, v\,\texttt{@}\,p \,]\!] &\equiv v\,\texttt{@}\,\mathcal{VP}[\![\, p \,]\!] && (20)\\
\mathcal{VP}[\![\; \texttt{[ ]} \;]\!] &\equiv \texttt{Null} && (21)\\
\mathcal{VP}[\![\; \texttt{( )} \;]\!] &\equiv \texttt{Void} && (22)\\
\mathcal{VP}[\![\; \texttt{(}\, p \,\texttt{)} \;]\!] &\equiv \texttt{(}\, \mathcal{VP}[\![p]\!] \,\texttt{)} && (23)\\
\mathcal{VP}[\![\, p_1\texttt{:} \cdots \texttt{:}p_n\texttt{:}\, p \,]\!] &\equiv \texttt{Cons}_\texttt{n}\ \mathcal{VP}[\![\, p_1 \,]\!] \cdots \mathcal{VP}[\![\, p_n \,]\!]\ \mathcal{VP}[\![\, p \,]\!] && (24)\\
\mathcal{VP}[\![\; \texttt{(}\, p_1 \texttt{,} \cdots \texttt{,}\, p_n \,\texttt{)} \;]\!] &\equiv \texttt{(}\ \texttt{Tuple}_\texttt{n}\ \mathcal{VP}[\![\, p_1 \,]\!] \cdots \mathcal{VP}[\![\, p_n \,]\!]\ \texttt{)} && (25)\\
\mathcal{VP}[\![\; \texttt{[}\, p_1 \texttt{,} \cdots \texttt{,}\, p_n \,\texttt{]} \;]\!] &\equiv \texttt{(}\ \texttt{List}_\texttt{n}\ \mathcal{VP}[\![\, p_1 \,]\!] \cdots \mathcal{VP}[\![\, p_n \,]\!]\ \texttt{)} && (26)\\
\mathcal{VP}[\![\, p_1\ op\ p_2 \,]\!] &\equiv op\ \mathcal{VP}[\![\, p_1 \,]\!]\ \mathcal{VP}[\![\, p_2 \,]\!] && (27)\\
\mathcal{VP}[\![\, con\ p_1 \cdots p_n \,]\!] &\equiv con\ \mathcal{VP}[\![\, p_1 \,]\!] \cdots \mathcal{VP}[\![\, p_n \,]\!] && (28)
\end{aligned}
$$

Figure 24. Translation rules for patterns

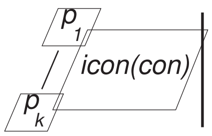
4.5.1 Invisible patterns

Variable names that begin with the **$** symbol are treated specially in Visual Haskell. Instead of displaying a box, these variables have no display at all: as a result, just an output pad appears in their place.

This mechanism is used in special circumstances to add arcs where otherwise there would be none, as an aid to making visual representation of higher-order definitions understandable. Its use is described in section 4.6.3.

The special pattern **$$** is used to denote an empty pattern list. It is used for *let*-expressions, *if-then-else* expressions, and pattern bindings.

As noted in section 3.2, bindings arcs can—and usually do—have zero length. For example, in the case of constructor patterns with an icon, the syntax would look more like this:

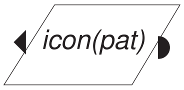


4.6 Scoping expressions

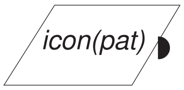
Haskell has several means of introducing a new scope: *let*-expressions, *case*-expressions, λ-abstractions, and pattern and function bindings. In Visual Haskell, all of these, as well as conditional expressions, are dealt with by *scoping expressions*, or *scopes* for short.

A scoping expression is essentially just an enclosing box; the box is equivalent to a lexical scope in a textual language. Each scope has one or more patterns; the distinguished variable **$$** is called the *null pattern*, and is used for scoping expressions that do not "really" have

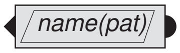
pat → *var*
where there is an icon defined for *var*



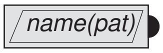
pat → *const*
where there is an icon defined for *const*



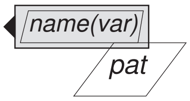
pat → *var*
where there is no icon defined for *var*



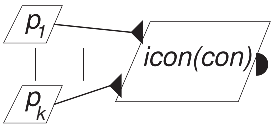
pat → *const*
where there is no icon defined for *const*



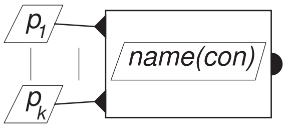
pat → *var* @ *pat*



pat → *con* $p_1 \cdots p_k$
where $k >= 1$ and there is an icon defined for *con*



pat → *con* $p_1 \cdots p_k$
where $k >= 1$ and there is no icon defined for *con*



pat → $*var*
(Used for unnamed arguments and function composition.)

pat → $$
(The null pattern variable, which is not displayed at all.)

Figure 25. Visual syntax for patterns

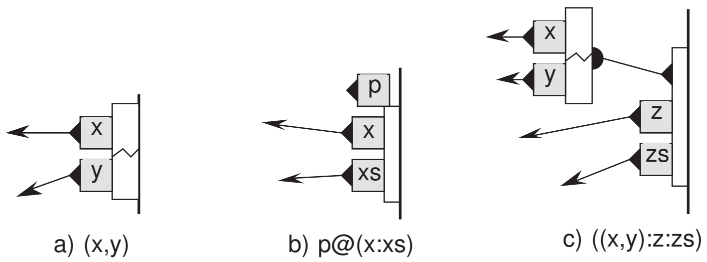


Figure 26. Some examples of patterns

patterns.[2]

There are two forms of scope: guarded and unguarded. Unguarded scopes (top of Figure 27) contain an expression and an optional set of declarations; guarded scopes (bottom of Figure 27) contain a series of expressions guarded by predicates and an optional set of declarations.

An additional set of rules, $\mathcal{VS}$, is used to translate scoping expressions (Figure 28). These just recursively re-apply the rules to their components. Note also that:

- The graphics for d are omitted if the optional *where* clause is not present,
- A variable bound in p_1 to p_n is crossed out if it conflicts with a variable bound in d,
- The syntax for an `otherwise` guard is just blank space.

4.6.1 Augmenting patterns

A second rule, $\mathcal{VS}_n$, is used by the translation of λ-abstractions, pattern bindings, and function bindings (Figure 29). Its purpose is to add patterns to a pattern list, so that higher-order function definitions are changed into first-order function definitions. This causes arcs to be drawn which aid in understanding the visual function definition.

Because of $\mathcal{VS}_n$, for example, the function definition

```
f = id
```

will be translated into

```
f $t1 = id $t1
```

4.6.2 Expressions

Figure 30 and Figure 31 list the translation rules for Haskell expressions that will use VHIF scoping expressions. In all cases, the $\mathcal{VS}$ translation is used by the right-hand side—part of the result right-hand side will always match $gscx$ or $uscx$. Figure 32 shows the visual syntax of these expressions.

[2] This is a clumsy device. It would probably be better simply to have a *list* of patterns; the empty list can then be used instead of this "null pattern."

$$\begin{array}{lcl} scx & \rightarrow & gscx \\ & \mid & uscx \end{array}$$

$$\begin{array}{lcl} uscx & \rightarrow & p_1 \cdots p_n \\ & & \texttt{->}\ e\ [\texttt{where}\ d] \end{array}$$

$$\begin{array}{lcl} gscx & \rightarrow & p_1 \cdots p_n \\ & & \texttt{|}\ g_1\ \texttt{->}\ e_1 \\ & & \cdots \\ & & \texttt{|}\ g_k\ \texttt{->}\ e_k \\ & & [\texttt{where}\ d] \end{array}$$

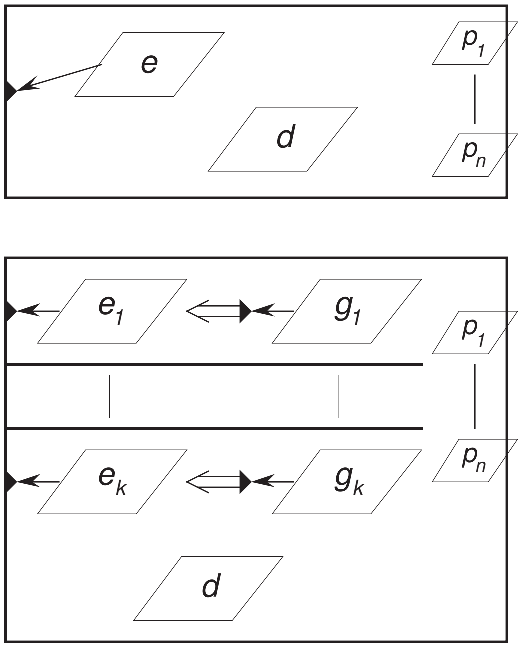


Figure 27. Visual syntax of scoping expressions

$$\begin{aligned} &\mathcal{VS}[\![\, p_1 \cdots p_m\ \texttt{->}\ e\ [\texttt{where}\ d]\,]\!] \\ &\quad\equiv\ \mathcal{VP}[\![\, p_1 \,]\!] \cdots \mathcal{VP}[\![\, p_m \,]\!]\ \texttt{->}\ \mathcal{VE}[\![\, e \,]\!]\ [\texttt{where}\ \mathcal{V}[\![\, d \,]\!]] \end{aligned} \tag{29}$$

$$\begin{aligned} &\mathcal{VS}[\![\, p_1 \cdots p_m\ \texttt{|}\ g_1\ \texttt{->}\ e_1; \cdots;\ \ \texttt{|}\ g_k\ \texttt{->}\ e_k\ [\texttt{where}\ d]\,]\!] \\ &\quad\equiv\ \mathcal{VP}[\![\, p_1 \,]\!] \cdots \mathcal{VP}[\![\, p_m \,]\!] \\ &\qquad\qquad \texttt{|}\ \mathcal{VE}[\![\, g_1 \,]\!]\ \texttt{->}\ \mathcal{VE}[\![\, e_1 \,]\!] \\ &\qquad\qquad \cdots \\ &\qquad\qquad \texttt{|}\ \mathcal{VE}[\![\, g_k \,]\!]\ \texttt{->}\ \mathcal{VE}[\![\, e_k \,]\!]\ [\texttt{where}\ \mathcal{V}[\![\, d \,]\!]] \end{aligned} \tag{30}$$

Figure 28. Translation rules for scoping expressions

$$\begin{aligned}
&\mathcal{VS}_n[\![\, p_1 \cdots p_m \; \texttt{->} \; e \; [\texttt{where} \; d] \,]\!] \\
&\quad \equiv \; \mathcal{VS}[\![\, p_1 \cdots p_m \; \texttt{->} \; e \; [\texttt{where} \; d] \,]\!], \text{ if } m = n && (31)\\
&\quad \equiv \; \mathcal{VS}[\![\, p_1 \cdots p_m \; \texttt{\$t}_{\texttt{m+1}} \cdots \texttt{\$t}_{\texttt{n}} \\
&\qquad\qquad \texttt{->} \; \texttt{(} \; e \; \texttt{)} \; \texttt{\$t}_{\texttt{m+1}} \cdots \texttt{\$t}_{\texttt{n}} \; [\texttt{where} \; d] \,]\!], \text{ if } m < n && (32)
\end{aligned}$$

$$\begin{aligned}
&\mathcal{VS}_n[\![\, p_1 \cdots p_m \; \texttt{|} \; g_1 \; \texttt{->} \; e_1; \cdots; \; \texttt{|} \; g_k \; \texttt{->} \; e_k \; [\texttt{where} \; d] \,]\!] \\
&\quad \equiv \; \mathcal{VS}[\![\, p_1 \cdots p_m \; \texttt{|} \; g_1 \; \texttt{->} \; e_1; \cdots; \; \texttt{|} \; g_k \; \texttt{->} \; e_k \; [\texttt{where} \; d] \,]\!], \text{ if } m = n && (33)\\
&\quad \equiv \; \mathcal{VS}[\![\, p_1 \cdots p_m \; \texttt{\$t}_{\texttt{m+1}} \cdots \texttt{\$t}_{\texttt{n}} \\
&\qquad\qquad \texttt{|} \; g_1 \; \texttt{->} \; \texttt{(} \; e_1 \; \texttt{)} \; \texttt{\$t}_{\texttt{m+1}} \cdots \texttt{\$t}_{\texttt{n}} \\
&\qquad\qquad \texttt{...} \\
&\qquad\qquad \texttt{|} \; g_k \; \texttt{->} \; \texttt{(} \; e_k \; \texttt{)} \; \texttt{\$t}_{\texttt{m+1}} \cdots \texttt{\$t}_{\texttt{n}} \; [\texttt{where} \; d] \,]\!], \text{ if } m < n && (34)
\end{aligned}$$

Figure 29. Translation rules for augmenting scoping expressions

- *let*-expressions

  A *let*-expression is treated as an unguarded scoping expression with a null pattern (it is thus always "enabled"). The use of the null pattern can be seen to be merely a mechanism to fit all of the above-mentioned Haskell constructs into a single visual construct.

- λ-abstractions

  A λ-abstraction contains one unguarded scoping expression. λ-abstractions look similar to unguarded, single-clause function bindings; however, they have no label, and can of course appear within expressions. A special rule (rule 37) simplifies the common case in which a λ-abstraction contains a *let*-expression.

  Note how the $\mathcal{VS}_n$ rule is used to add unstated arguments to the pattern list—these will be displayed as pads.

- *case*-expressions

  Each alternative of a *case*-expression is translated into a guarded or unguarded scoping expression; the scoping expressions are "stacked" on top of each other.

- Conditionals

  One or a series of conditionals (*if-then-else*) is converted into a guarded scoping expression with a null pattern. This treatment makes a series of *if-then-else* expressions visually simple. (Translation into *case*-expressions, as in [4], would be too cumbersome visually.)

4.6.3 Bindings

Figure 33 and Figure 34 list the translation rules for pattern and function bindings. Again, part of the right-hand side always matches *scx*, *gscx*, or *uscx*. Figure 35 shows the visual syntax.

$$\mathcal{VE}[\![\,\texttt{let}\ d\ \texttt{in}\ e\,]\!] \equiv \texttt{let}\ \mathcal{VS}[\![\,\texttt{\$\$ ->}\ e\ \texttt{where}\ d\,]\!] \tag{35}$$

$$\mathcal{VE}[\![\,\backslash\ p_1 \cdots p_m\ \texttt{->}\ e\,]\!] \equiv \backslash\ \mathcal{VS}_n[\![\,p_1 \cdots p_m\ \texttt{->}\ e\,]\!] \tag{36}$$

$$\mathcal{VE}[\![\,\backslash\ p_1 \cdots p_m\ \texttt{->}\ \texttt{let}\ d\ \texttt{in}\ e\,]\!] \equiv (\ \backslash\ \mathcal{VS}_n[\![\,p_1 \cdots p_m\ \texttt{->}\ e\ \texttt{where}\ d\,]\!]\ ) \tag{37}$$

e has arity n

Figure 30. Translation rules for *let*-expressions and λ-abstractions.

$$\mathcal{VE}[\![\,\texttt{if}\ g_1\ \texttt{then}\ e_1\ \texttt{else if}\ \cdots\ \texttt{else}\ e_k\,]\!]$$

$$\equiv\ \texttt{ifcase}\ \mathcal{VS}[\![\,\texttt{\$\$}\ \ |\ g_1\ \texttt{->}\ e_1; \cdots ;\ \ |\ \texttt{otherwise}\ \texttt{->}\ e_k\,]\!] \tag{38}$$

$$\mathcal{VE}[\![\,\texttt{case}\ e\ \texttt{of}\ scx_1; \cdots ; scx_k\,]\!]$$

$$\equiv\ \texttt{case}\ \mathcal{VE}[\![\,e\,]\!]\ \texttt{of}\ \mathcal{VS}[\![\,scx_1\,]\!]; \cdots ; \mathcal{VS}[\![\,scx_k\,]\!] \tag{39}$$

Figure 31. Translation rules for *if-then-else* and *case*-expressions

$exp \rightarrow$ **let** $uscx$

$exp \rightarrow \backslash\ uscx$

$exp \rightarrow$ **case** e **of** scx_1 $\cdots$ scx_k

$exp \rightarrow$ **ifcase** $gscx$

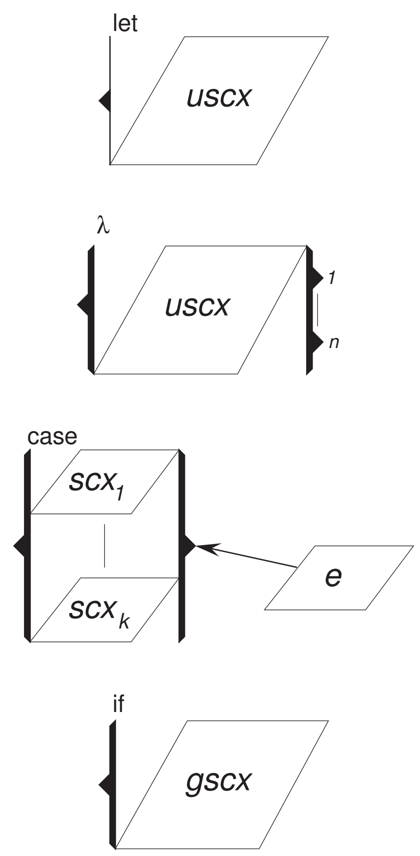


Figure 32. Visual syntax of expressions containing scoping expressions

$$\mathcal{V}[\![\, p \texttt{ = } e \,]\!] \equiv \mathcal{VP}[\![\, p \,]\!] \texttt{ = } \mathcal{VE}[\![\, e \,]\!],\ \text{if } n = 0 \tag{40}$$

$$\equiv p\ \mathcal{VS}[\![\, \texttt{\$t}_1 \cdots \texttt{\$t}_n \texttt{ -> (} e \texttt{) } \texttt{\$t}_1 \cdots \texttt{\$t}_n \,]\!],\ \text{if } n > 0 \tag{41}$$

where p has arity n

$$\mathcal{V}[\![\, p \texttt{ = } e \texttt{ where } d \,]\!]$$

$$\equiv \mathcal{VP}[\![\, p \,]\!] \texttt{ = } \mathcal{VS}[\![\, \texttt{\$\$ -> where } d \,]\!],\ \text{if } n = 0 \tag{42}$$

$$\equiv p\ \mathcal{VS}[\![\, \texttt{\$t}_1 \cdots \texttt{\$t}_n \texttt{ -> (} e \texttt{) } \texttt{\$t}_1 \cdots \texttt{\$t}_n \,]\!] \texttt{ where } d,\ \text{if } n > 0 \tag{43}$$

where p has arity n

$$\mathcal{V}[\![\, p \ \texttt{|}\ g_1 \texttt{ = } e_1; \cdots;\ \texttt{|}\ g_k \texttt{ = } e_k\ [\texttt{where } d] \,]\!]$$

$$\equiv \mathcal{VP}[\![\, p \,]\!] \texttt{ = } \mathcal{VS}[\![\, \texttt{\$\$ | } g_1 \texttt{ -> } e_1; \cdots;\ \texttt{|}\ g_k \texttt{ -> } e_k\ [\texttt{where } d] \,]\!],\ \text{if } n = 0 \tag{44}$$

$$\equiv p\ \mathcal{VS}[\![\, \texttt{\$t}_1 \cdots \texttt{\$t}_n$$

$$\texttt{|}\ g_1 \texttt{ -> (} e_1 \texttt{) } \texttt{\$t}_1 \cdots \texttt{\$t}_n$$

$$\cdots$$

$$\texttt{|}\ g_k \texttt{ -> (} e_k \texttt{) } \texttt{\$t}_1 \cdots \texttt{\$t}_n\ [\texttt{where } d] \,]\!],\ \text{if } n > 0 \tag{45}$$

where p has arity n

Figure 33. Translation rules for pattern bindings

– Pattern bindings

A simple pattern binding, of the form $p = e$, is handled simply by making a binding connection from e to p. Pattern bindings with auxiliary declarations or with a guarded right-hand side, however, must be translated into a scoping expression.

Note that function-valued pattern bindings are treated as function bindings, not as pattern bindings. This keeps value and function definitions visually distinct.

– Function bindings

Each clause of a function binding becomes a scoping expression; the scoping expressions are "stacked" on top of each other.

As for λ-abstractions, the $\mathcal{VS}_n$ rule is used to add unstated arguments to the pattern list—these will be displayed as pads.

4.7 List comprehensions

The list comprehension syntax is a relatively new addition to Visual Haskell, and should be considered experimental. In the simple examples I have tried so far, list comprehensions do not benefit from visual representation. No matter—the goal of being able to represent all of standard Haskell makes it worthwhile to explore syntaxes for list comprehensions. The programs I have worked with in a visual representation up to now tend to use functions such

$$
\begin{aligned}
&\mathcal{V}[\![\, f\ p_1 \cdots p_m \ \texttt{=}\ e\ [\texttt{where}\ d]\,]\!] \\
&\quad \equiv \quad f\ \mathcal{VS}_n[\![\, p_1 \cdots p_m \texttt{->}\ e\ [\texttt{where}\ d]\,]\!] \\
&\qquad\quad \text{where } f \text{ has arity } n
\end{aligned}
\tag{46}
$$

$$
\begin{aligned}
&\mathcal{V}[\![\, f\ p_1 \cdots p_m\ \ \texttt{|}\ g_1\ \texttt{=}\ e_1; \cdots;\ \ \texttt{|}\ g_k\ \texttt{=}\ e_k\ [\texttt{where}\ d]\,]\!] \\
&\quad \equiv \quad f\ \mathcal{VS}_n[\![\, p_1 \cdots p_m\ \texttt{|}\ g_1\ \texttt{->}\ e_1; \cdots;\ \ \texttt{|}\ g_k\ \texttt{->}\ e_k\ [\texttt{where}\ d]\,]\!] \\
&\qquad\quad \text{where } f \text{ has arity } n
\end{aligned}
\tag{47}
$$

Figure 34. Translation rules for function bindings

$decl \rightarrow p = e$

$decl \rightarrow p = scx$

$decl \rightarrow \ f\ scx_1$
$\qquad\qquad \cdots$
$\qquad\qquad f\ scx_k$

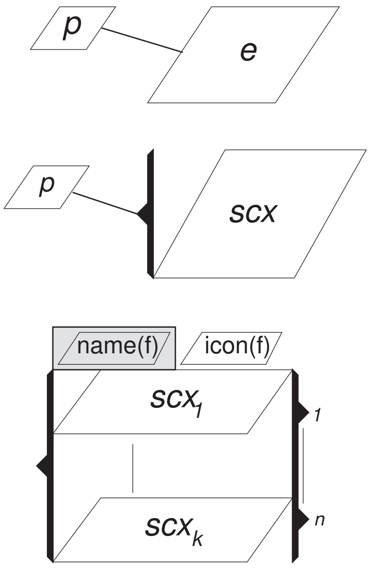


Figure 35. Visual syntax of bindings

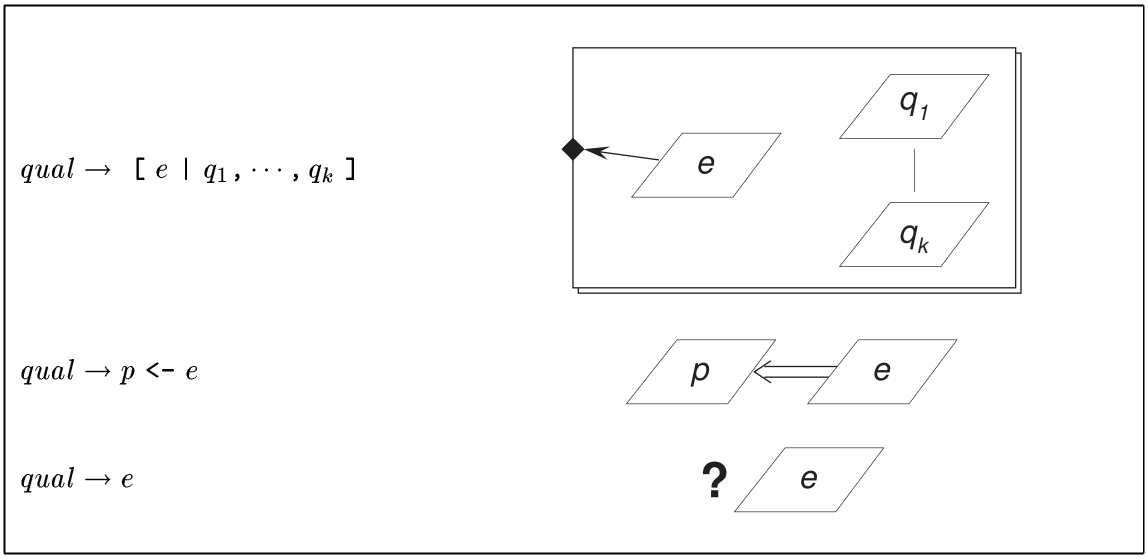


Figure 36. Visual syntax of list comprehensions

as *map* and *filter* in preference to comprehensions. I cannot say whether or not this is due to a different viewpoint one gains from visual (as opposed to textual) program representation.

Since list comprehensions are the same in VHIF as in Haskell, I have not supplied rules for them. Figure 36, however, is the visual syntax for comprehensions.

5 Advanced Visual Representations

When using Visual Haskell, one seems continually to be inventing new syntax that makes a particular piece of Haskell code visually simpler. This is unfortunate; nonetheless, I hope that this will no longer be necessary once Visual Haskell is refined a bit more. In this section, I present some extensions that have proven useful.

5.1 Function composition

A short-coming of visual program representations is that they do not lend themselves well to higher-order functions. This is why, for example, the translation rules for λ-abstractions and function bindings sometimes add "invisible" patterns. In effect, a higher-order definition is converted into a first-order definition.

Function composition suffers a similar problem. Although it is a higher-order function, we would like the visual representation to show arcs indicating the "flow of data" through a composition. Argument slots (section 3.4) do not fulfil this requirement.

The following translation rule converts one or a series of function compositions into a form that is visually more meaningful :

$$\mathcal{VE}[\![\, f_1 \cdot \;\ldots\; \cdot f_k \,]\!] \equiv \mathcal{VE}[\![\, \texttt{\$}x \texttt{ -> } f_1(\cdots(f_k\ \texttt{\$}x)\cdots) \,]\!] \qquad (48)$$

A series of compositions is thus converted into a λ-abstraction, with an invisible variable as argument, and containing a series of function applications. To illustrate, Figure 37 shows the expression $f \cdot g \cdot h$.

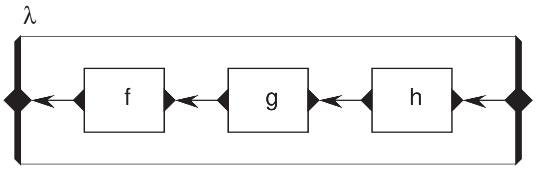


Figure 37. Function composition: $f \cdot g \cdot h$

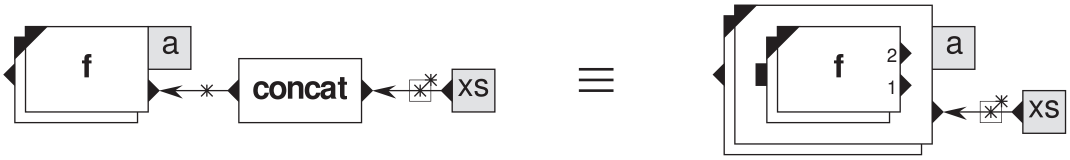


Figure 38. Illustrating fold promotion

5.2 Permuted arguments

In my work using the algebraic style of program transformation on higher-order functions such as *map*, *foldl* and *foldr*, and *scanl* and *scanr*, I have needed to introduce λ-abstractions solely for the purpose of "re-ordering" function arguments.

This is somewhat cumbersome textually, but is easy to do in a visual notation. The following rule removes this kind of λ-abstraction and replaces them with a permuted application:

$$\mathcal{VE}[\![\setminus p_1 \cdots p_m \texttt{ -> } f\ p_{1'} \cdots p_{m'}]\!] \equiv \texttt{applyp}_{\texttt{is}}\ \mathcal{VE}[\![f]\!] \tag{49}$$

where $is = [1'\texttt{,} \cdots \texttt{,} m']$ is some permutation of $[1 \cdots m]$. I am assuming here that the λ-abstraction has all arguments specified—in other words, that rule 36 or rule 37 has already been applied.

For example, a variant of Bird's "fold promotion" law [2] using Haskell's "left-handed" directed reduction function, *foldl*, is

$$foldl\ f\ a\ (concat\ xs)\ \equiv\ foldl\ (foldl\ f)\ a\ xs \tag{50}$$

If we wish to use *foldr* instead, we must either introduce an auxiliary function or use a λ-abstraction, like this:[3]

$$foldr\ f\ a\ (concat\ xs)\ \equiv\ foldr\ (\setminus\ ys\ y \texttt{ -> } foldr\ f\ y\ ys)\ a\ xs \tag{51}$$

Figure 38 illustrates identity 51. See [14] for a more realistic example of why rule 49 is needed.

5.3 Rubber icons

Rubber icons, and the unfolded functions of the next section, are an aid to program visualisation. They help to illustrate the *structure* of expressions built with higher-order functions.

[3] In this particular case, we can use *flip*, but in general this is not so.

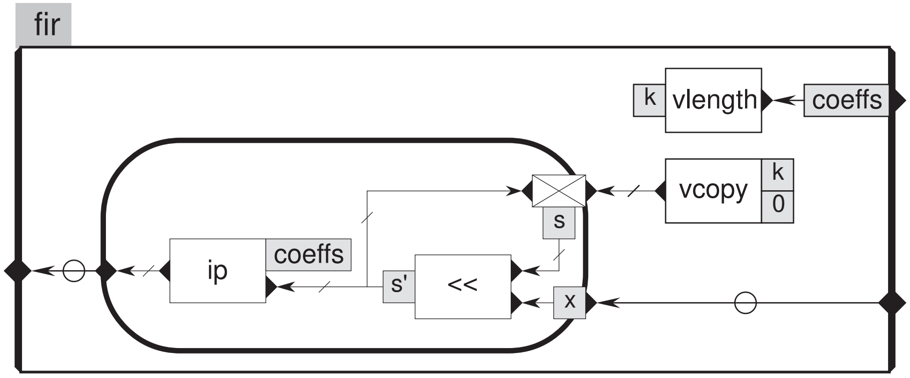


Figure 39. An FIR filter, using a rubber *sstate* icon

A “rubber” icon is one that can stretch to contain more complex graphics than supported by argument slots. This is used to illustrate the computation that occurs within each “iteration” of higher-order functions such as *map* and *foldl*. To illustrate, Figure 39 shows a transversal filter after program transformation has been used to combine two processes into one. The textual definition is:

```
fir coeffs = sstate
                 (\s x -> let s' = s << x in (s', ip coeffs s'))
                 (vcopy k 0)
             where
                 k = vlength coeffs
```

The rounded-rectangle graphic represents the *sstate* function on streams; for each input stream element, *sstate* also takes a state value, and produces an output value and the next state. If you consider the crossed rectangle as though it were a mutable state variable, the visual representation reveals the computation performed by this application of *sstate* much more readily than by examining the textual definitions. [14] shows the complete series of steps in transforming the fir function into a pipeline of processes.

To formally express the translation into rubberised representation, we would need to translate the argument drawn inside the icon into a λ-abstraction, and be able to express the connection of this function to the pads on the *inside* of the icon.

5.4 Unfolded higher-order functions

Another aid to visualisation is to unfold higher-order functions. This is valuable in visualising the *pattern of computation* captured by a higher-order function. For example, Figure 40 shows unfolded representations of *map*, *foldl*, and *scanl*—the relationship between these functions can be visualised in a very “structural” way. In Figure 5c of section 2.2, by comparison, I unfolded the function definition with a known argument to achieve a similar effect; if the unfolded representation just described can be formalised, the need to do this can be eliminated.

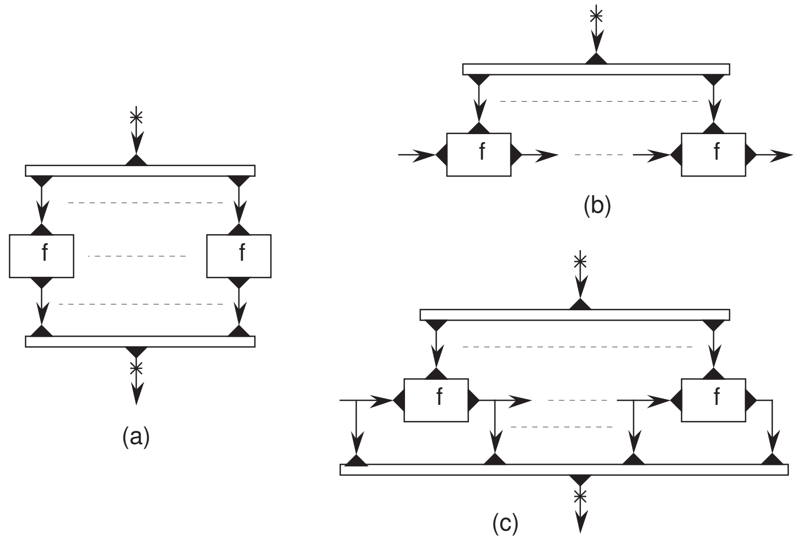


Figure 40. Unfolded higher-order functions: a) *map*; b) *foldl*; c) *scanl*

6 Conclusions

There is, I believe, significant value in a precise visual syntax for Haskell, which can replace the *ad-hoc* notations that are sometimes used. In this paper, I have presented my suggestions for such a notation.

There is a substantial amount of work yet to do. As a critical reader will have noticed, I have glossed over a great many details, and treated others with only a very informal description. The missing syntax—modules, type declarations, type classes and instances, and arrays, among others—needs to be supplied if Visual Haskell is to be used as a formal programming notation. A translation from VHIF back to Haskell also needs to be given—this would define the semantics of Visual Haskell in terms of Haskell. Although I have used Visual Haskell mainly for program visualisation so far, the idea of a "two-view" development system is intriguing. Dawson's prototype visual editor indicates that such a system is feasible.

7 Acknowledgments

I would like to thank my academic supervisors, Professor Warren Yates, of the University of Technology, Sydney, and Dr. John Potter, of the Microsoft Institute, for their advice and support. Ken Dawson wrote the prototype Visual Haskell editor, and in the process stimulated development of Visual Haskell from an *ad-hoc* notation into something that might become a useful language. This work was supported by an Australian Post-graduate Research Award.

References

[1] J. C. Bier, E. E. Goei, W. H. Ho, P. D. Lapsley, M. P. O'Reilly, G. C. Sih, and E. A. Lee. Gabriel: A design environment for DSP. *IEEE Micro*, pages 28–45, October 1990.

[2] R. Bird. A calculus of functions for program derivation. Technical Report Technical Monograph PRG-64, Oxford Computing Laboratory Programming Research Group, Oxford, 1987.

[3] Ken Dawson. Visual Haskell editor and parser. Technical report, School of Electrical Engineering, University of Technology, Sydney, 1993. Undergraduate thesis report.

[4] Paul Hudak et al. Report on the functional programming language Haskell, a non-strict purely-functional language, version 1.2. *SIGPLAN Notices*, May 1992.

[5] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

[6] Paul Kelly. *Functional Programming for Loosely-coupled Multiprocessors.* Research Monographs in Parallel and Distributed Computing. Pitman, 1989.

[7] Edward A. Lee and David G. Messerschmitt et al. An overview of the Ptolemy project. Available by anonymous ftp from the Ptolemy distribution site, ptolemy.eecs.berkeley.edu, March 1994.

[8] M.A. Najork and E. Golin. Enhancing Show-and-Tell with a polymorphic type system and higher-order functions. In *IEEE Workshop on Visual Languages, Skokie, Illinois*, pages 215–220, October 1990.

[9] John Rasure and Mark Young. Dataflow visual languages. *IEEE Potentials*, 11(2):30–33, April 1992.

[10] Chris Reade. *Elements of Functional Programming.* Addison Wesley, 1989.

[11] H. John Reekie. Towards effective programming for parallel digital signal processing. Technical Report 92.1, Key Centre for Advanced Computing Sciences, University of Technology, Sydney, May 1992.

[12] H. John Reekie. *Functional and Dataflow Programming for Realtime Distributed Multiprocessors.* PhD thesis, School of Electrical Engineering, University of Technology, Sydney, 1994. In preparation.

[13] John Reekie. Integrating block-diagram and textual programming for parallel DSP. In *ISSPA '92*, August 1992.

[14] John Reekie and John Potter. Process network transformation. In David Arnold, editor, *Parallel Computing and Transputers (PCAT-93)*, pages 376–383. IOS Press, November 1993.

[15] Kevin Waugh, Patrick McAndrew, and Greg Michaelson. Parallel implementations from functional prototypes—a case study. Technical Report 90/4, Heriot-Watt University, 1990.