HasView : A Visual Editor for Haskell

Brandon Wang and Kaushik Iyer

CS264 Final Project
May 14, 2011

I. Introduction

HasView is a visual language, similar to LabView. However, rather than being a completely new visual language, HasView uses Haskell as a base. The user can choose to use the dataflow visual programming links, simply code in Haskell, or participate in any level in-between. HasView draws from work on visualizing Haskell, namely from [Ree94] and [Hug00]. These visualizations were generated from existing Haskell code; we seek to generate Haskell from visualizations. However, our project is not to completely replace Haskell with a functional interface. We provide a visual aspect to data flow, and the user may control the extent they utilize it to, in addition to arbitrary text-based Haskell.

II. Nodes

HasView's layout is familiar to the visual programming language, LabView. The fundamental operational block is a node, a literal "black" (our editor shows them as white) box with inputs and outputs. This is a HasView node, and is the building block to creating programs. A basic node does nothing; we extend it to contain lexical scope for its children, or create primitives, such as the Figure 1, which is a function apply.

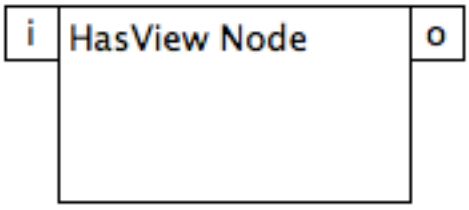

Figure 1: A HasView node with input "i" and output "o"

Although Haskell has many advanced language features, we found that its usage of lexical scoping and nested forms to be its means of transferring data. A user-defined container node specifies a lexical scope. A container node holds outer inputs and outputs, as well as a complementary set of inner inputs and outputs. Additionally, it may contain any other type of node in its lexical scope, as its children. The inner inputs and outputs are only accessible within the container node. This means that the binding of an inner output or input is only defined inside of a container node. The outer input and output ports serve as an interface to allow a higher level of abstraction to show that his is a black box.

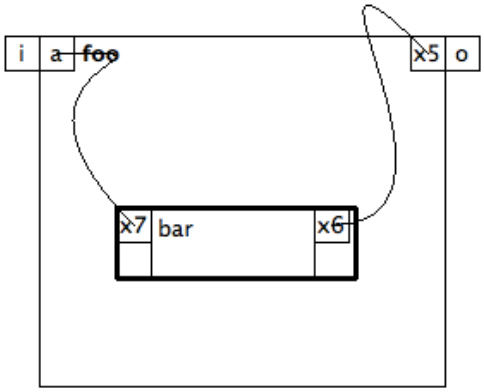

Figure 2: A HasView container node

The user can utilize HasView's linking system or not. We discovered that the actual linking can be annoying to more seasoned programmers from our poll of the CS264 class. As such, we allowed for variable binding in enclosures, and specify to use that variable. This allows the user to decide what requires the visual language, and what doesn't.

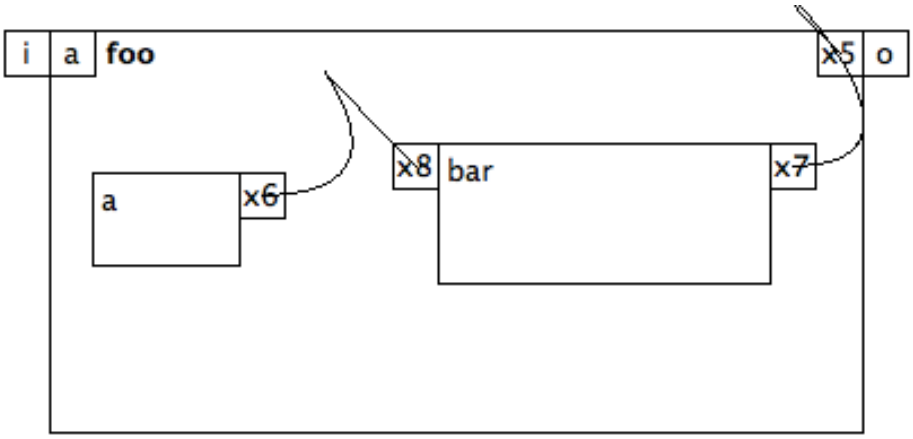

Figure 3: An equivalent to Figure 2

Furthermore, Haskell utilizes pattern matching for overloaded definitions. This is achieved in HasView by using a split container node. This split container node functions similar to a container node, but may contain multiple function bodies as well. An example can be seen in our Fibonacci program below.

The last fundamental node is the HasScript node. The HasScript node is assigned inputs and outputs with inner lexical scoping similar to the container node, but cannot contain node children. Instead, the user can type arbitrary Haskell, and explicitly utilize the lexically scoped variables. Its LabView equivalent is the MathScript node.

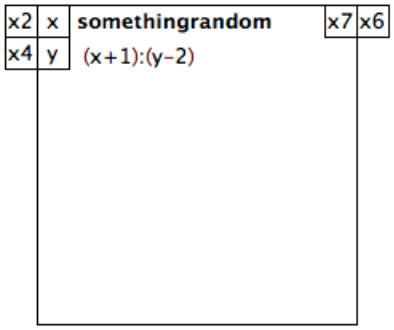

Figure 4: A HasScript node

We have some primitive nodes defined for convenience. It is trivial to extend these nodes, as we created them to show that they can exist without too much effort. Our two at the moment are constant nodes and functional nodes, which output a constant and function apply, respectively. Because these are primitives, they require no serialization and only assist in maintaining links.

III. Fibonacci

As an example, here is the canonical Fibonacci number program, written in a HasView style.

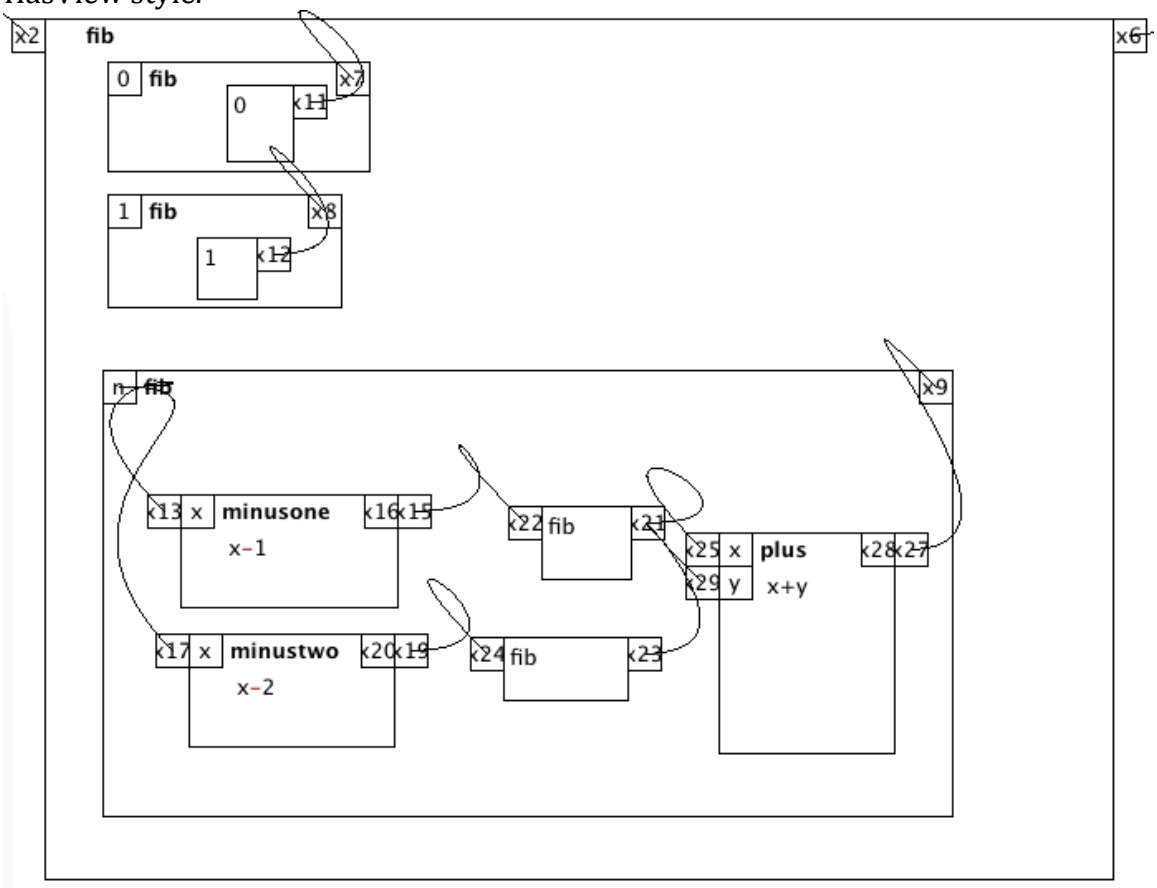

Figure 5: Fibonacci Numbers in HasView

The visualization can be compiled into Haskell with our editor. The output is quite verbose, but only because we maintain connections in terms of link variables.

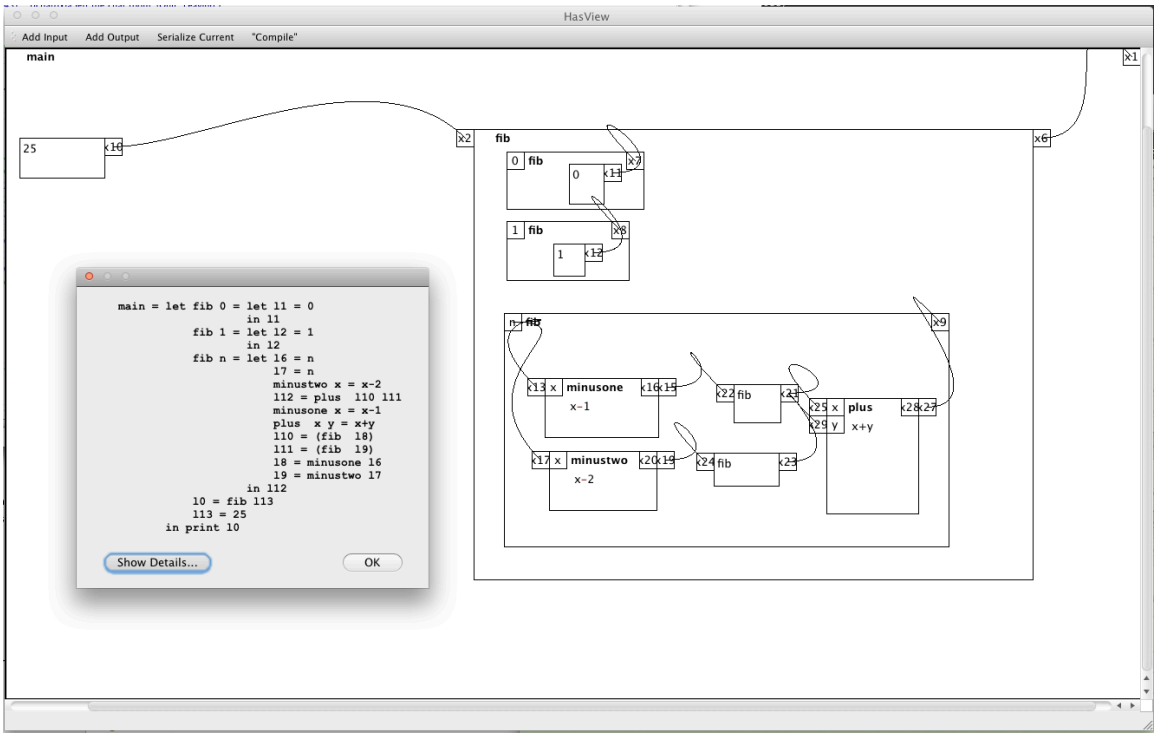

Figure 6: The complete Fibonacci program, with compiled output

The Haskell "compiled" code can be run in GHC.

```
main = let fib 0 = let l1 = 0
                   in l1
           fib 1 = let l2 = 1
                   in l2
           fib n = let l6 = n
                       l7 = n
                       minustwo x = x - 2
                       l12 = plus l10 l11
                       minusone x = x - 1
                       plus x y = x + y
                       l10 = (fib 18)
                       l11 = (fib 19)
                       l8 = minusone l6
                       l9 = minustwo l7
                   in l12
           l0 = fib l13
           l13 = 25
       in print l0
```

IV. Implementation

Our implementation of compilation is a two-step process. The first step, dubbed serialization, essentially creates lexical scopes and function definitions. The second step, named resolution, determines the order of execution within each enclosure. For our purposes, every node is a child of the main container node, which forces a print on its single output.

Serialization places each enclosure in the form of:

```
[function name] [arguments] = let [nested serializations]
                                  [resolutions]
                              in [output variables]
```

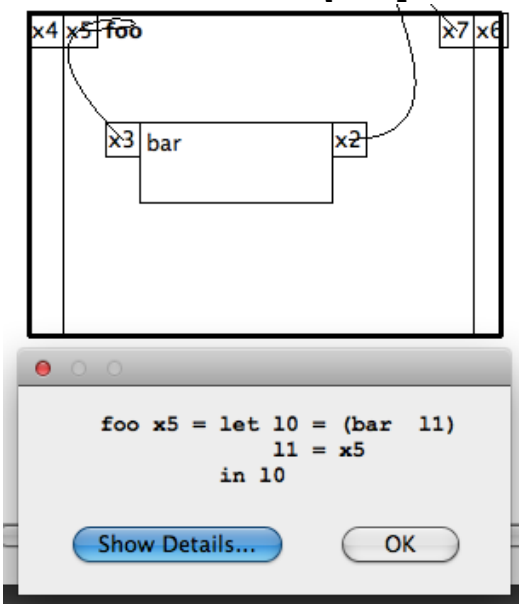

Figure 7: Serialized code

Serialization is a recursive procedure, and will continue to create nested serializations until it hits a serialization without children that require serialization.

Resolution follows the link backwards from a function outputs. It will assign a variable name to each link that it discovers, and defines the link in terms of a function invocation with its respective linked inputs and outputs. The resolution of Figure 7 is simply:

```
[x5] = foo [x4]
```

where x5 and x4 are the names of the link variables connected to x5 and x4, respectively.

V. Conclusion

Our implementation took about three weeks to conceive and create. This is not without bugs and incompleteness. We have the overall idea and working gears

to compile and run these programs, however we are lacking several features because of the short time span allotted to design this program.

Our biggest limitation is that we don't support non-functionality. Each resolution is determined and assumed to connect to the output of a container. This model is fine for functional programs, but cannot express variables or code that does not affect the direct output, or calls of monads. However, introducing code to resolve links not terminating at the output is not difficult; each link can be resolved independently instead of following our method of traversing the output to input. We only chose this method because of time limitations. Additionally, we do not yet have the support to delete inputs, outputs, or nodes.

We have multiple bugs, most of which are visual. It was difficult to learn PyQT as well as program something meaningful in a short period of time. The children of nodes can be dragged outside of nodes, and nodes can overlap, creating graphical ambiguity. It is up to the user to maintain a nice graphical layout. We also have an issue of the ordering of serializations, where we have a bug with sorting a list of serializations. This is an almost trivial issue and can be fixed given more time. We also had plans for multiple outputs for functions, and as such, there is half-implemented code, but it does not necessarily function yet. We also feel that the interface needs overall polish to be more usable.

Overall, we believe we attained the goal of our CS264 project – to make a visual extension of Haskell to allow for a simpler interface to programs in which data-flow may be difficult to maintain in a text-based environment. Our limitations and issues can be overcome given additional time. We don't seek to replace the text-based language completely. Rather, we create a hybrid, where the difficult to visualize areas are maintained in the visual aspect of a language, and the Haskell more fit for the text interface is maintained in text.

Our BitBucket repository can be accessed at https://bitbucket.org/brandonwang/hasview/overview.

VI. References

[Hug00] John Hughes, *Generalising Monads to Arrows*, May 2000. http://www.cse.chalmers.se/~rjmh/Papers/arrows.pdf

[Ree94] John Reekie, *Visual Haskell: A First Attempt,* August 1994*.* http://ptolemy.eecs.berkeley.edu/~johnr/papers/postscript/visual-haskell.ps.gz