Experimenting with an interactive visual functional programming environment

Arjan Boeijink

University of Twente, Enschede

NL FP Day 2016

About the Viskell project

- Creating readable and compact visualisations for FP
- Direct feedback and interaction based on the use of types
- Using multi-touch as the primary input method
- Addressing the scalability issues of larger visual programs
- Raising the level of visual abstractions using FP methods

History of the project

Philip Hölzenspies initiated work on multi-touch and type feedback
Too interesting to let it just gather dust after Philip left academia
Subject of student projects (Derk, Jan-Jelle, Kristel, Martijn, Richard)

Ongoing work

- Frank: Visual components for pattern matching and guards
- Wander: Larger scale structures and application to hardware

Haskell like language and visual programming

Simple core language and purity

Only limited number of visual constructs can be used in practice
Mutable state and updating variables are hard to visualize well

Compact expressiveness

Minimizing the amount of visual elements is required for scaling
Use expressiveness of first class functions and FP abstractions

No explicit evaluation order

Lack of linear execution order fits better with 2D visualizations
Working with a graph makes concurrent modifactions more natural

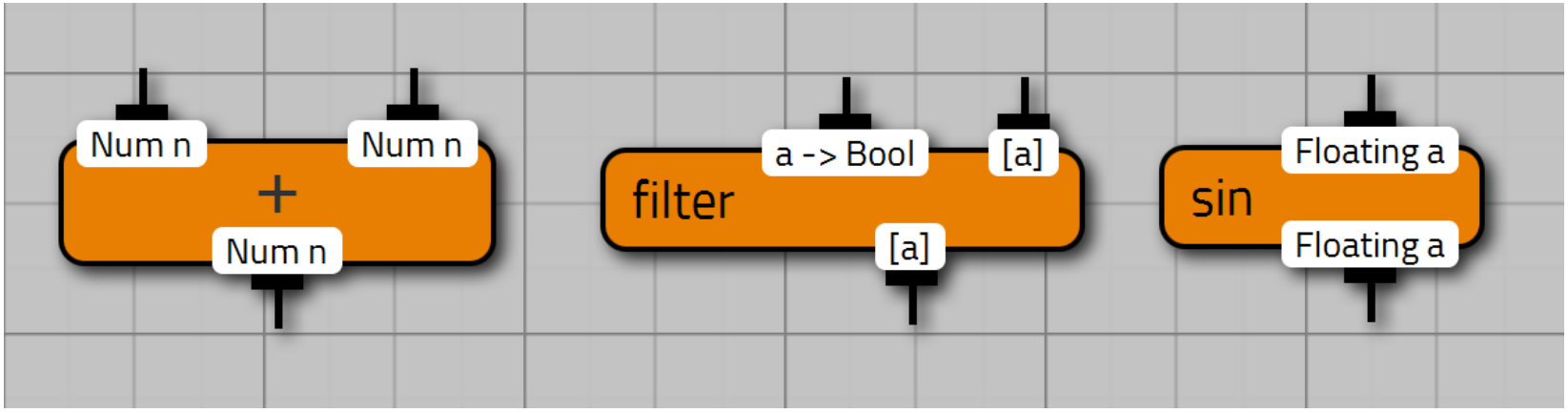
Advanced type system

Exploiting types helps minimizing the number of input actions
Direct type feedback can be a big help to every FP programmer

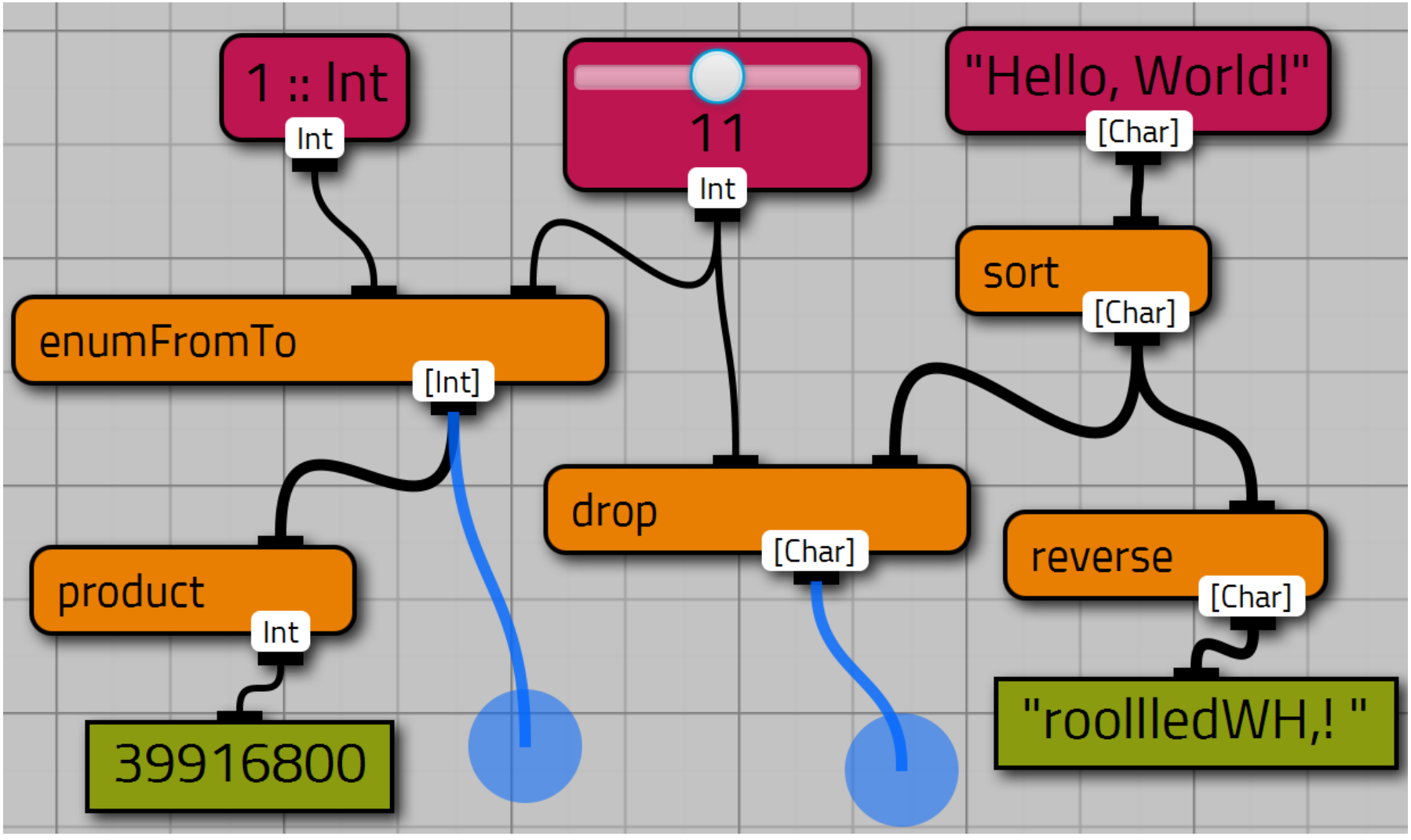
Blocks with names and types in a graph

Design choices made for the blocks elements

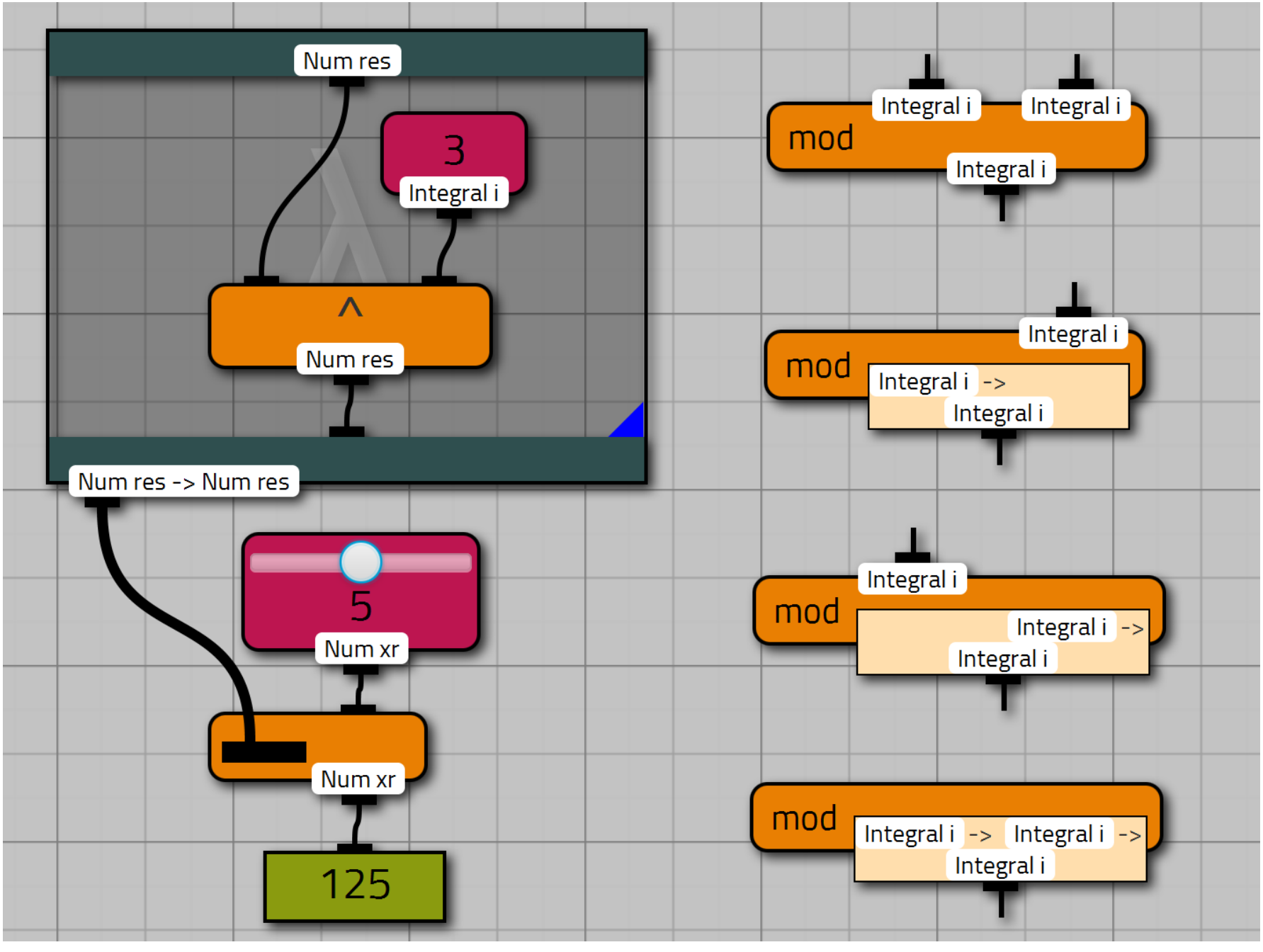
- No common language for icons, thus most blocks have names
- Names/types horizontal (inputs on top, outputs on bottom)
- Types are shown locally as assistance to programmer
- No arrows, but implicit topdown flow (follows gravity)
- Touch-sensitive areas where to draw connections to others
- Types are on the edge, so they can hide smoothly



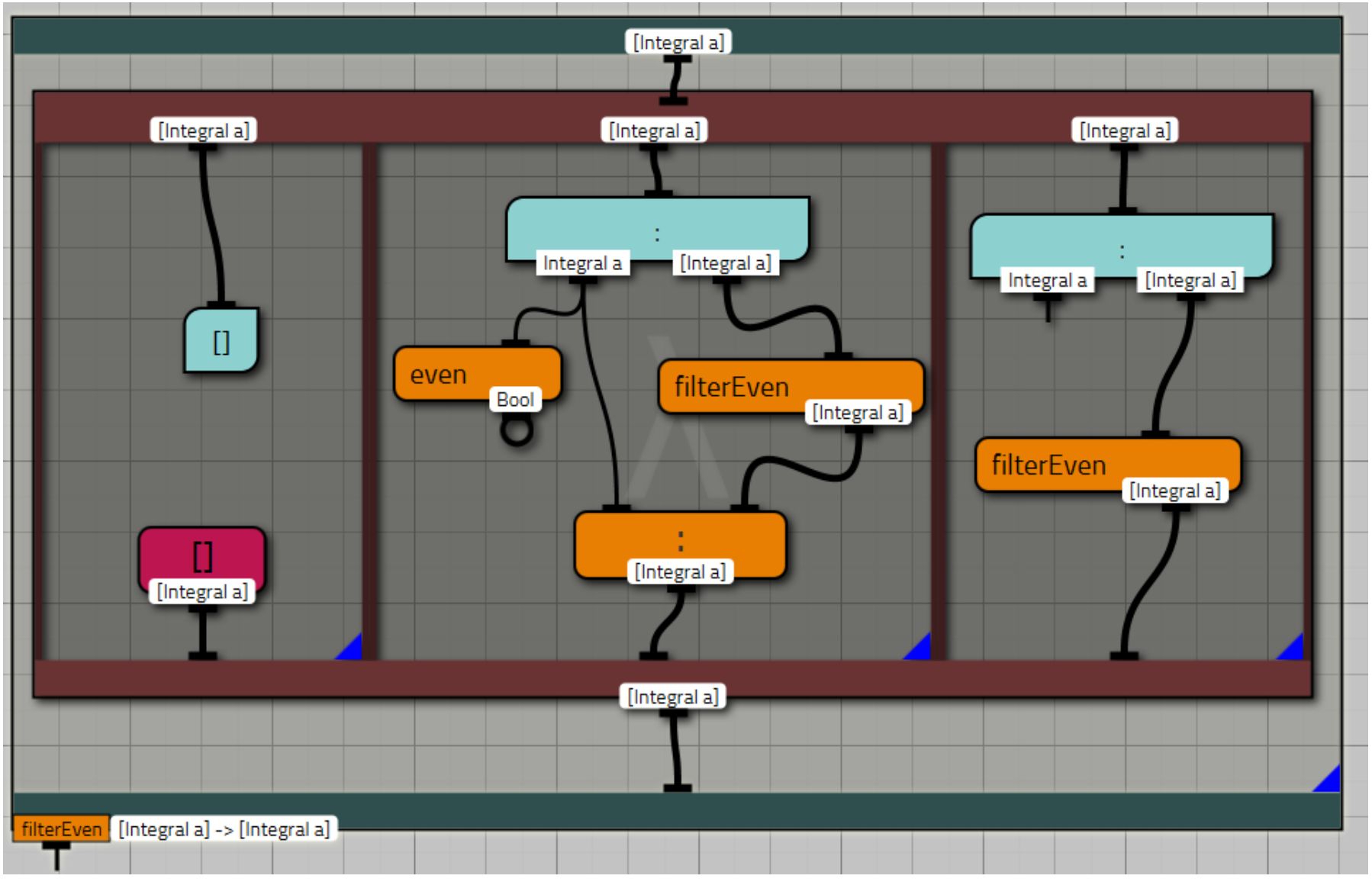
Programming anywhere in an (incomplete) graph



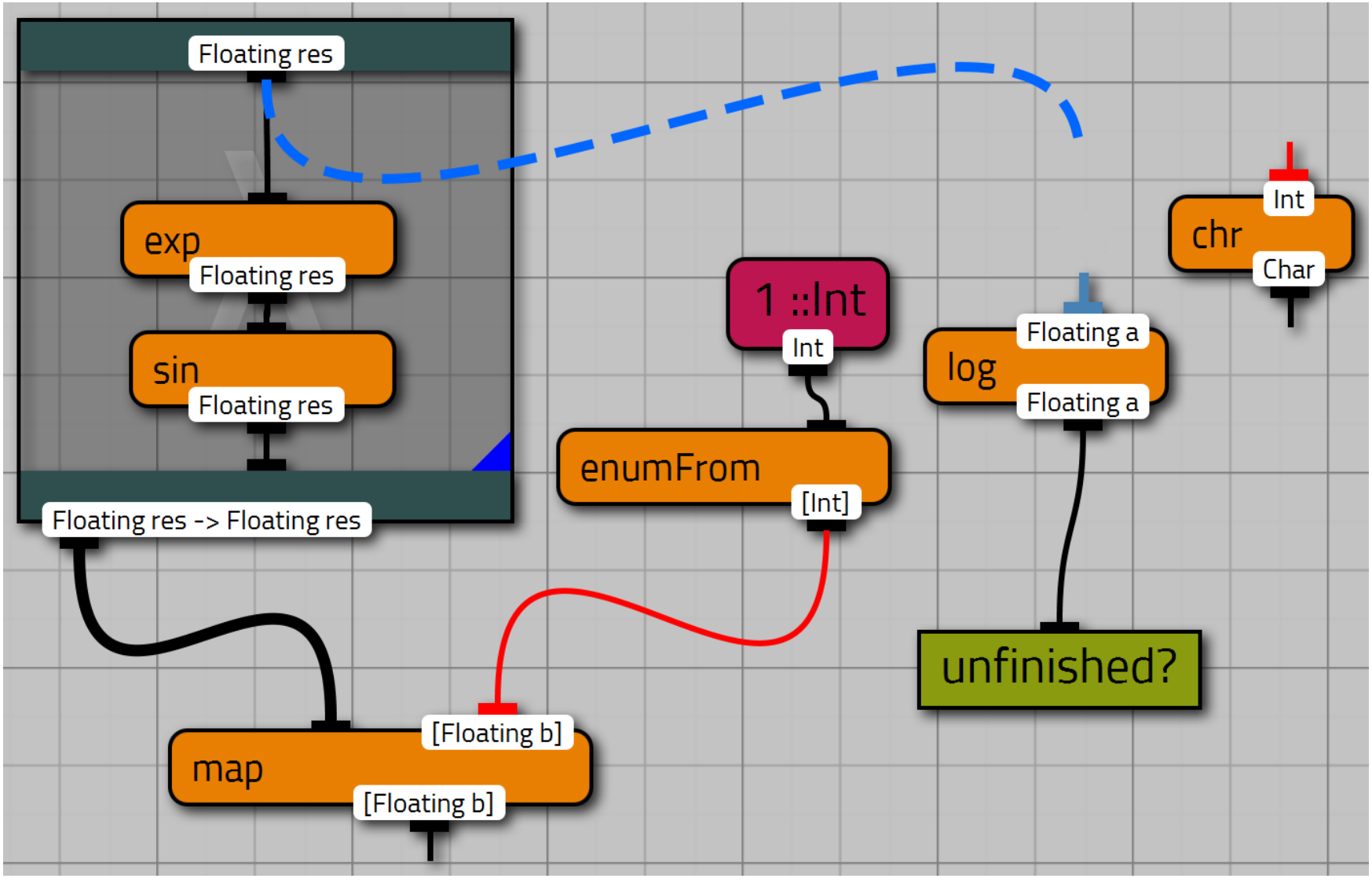
Lambdas, application, and currying



Case expression (wrapped in a named lambda)

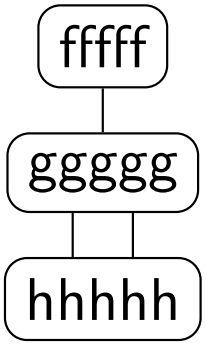


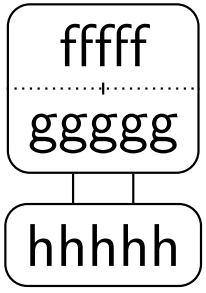
Direct feedback and type guided programming

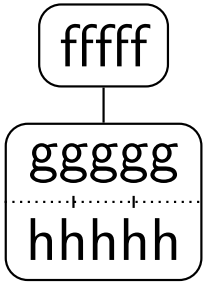


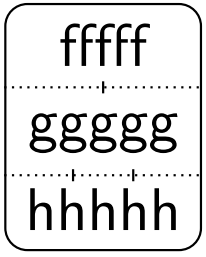
Sketch of stacking and nesting blocks

Vertical composition of functions/blocks with compact form

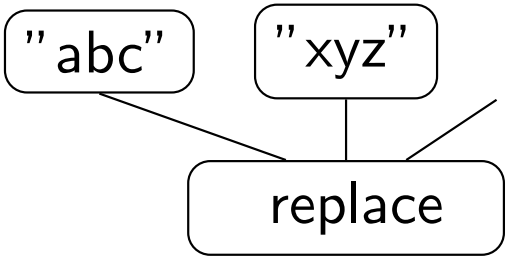


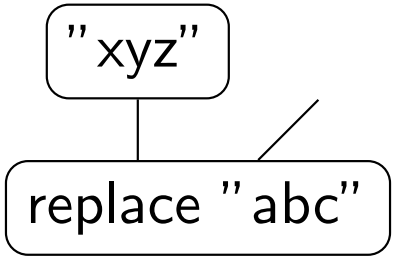


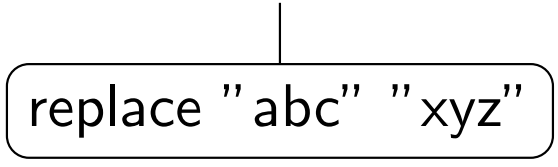




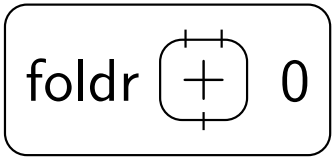
Application of constants can be compacted horizontally

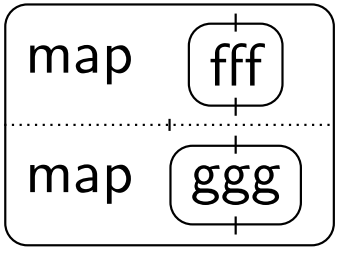


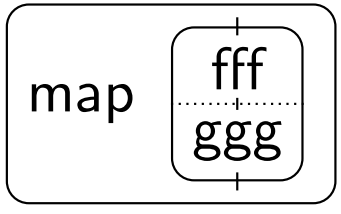




Using function as constants to higher order functions

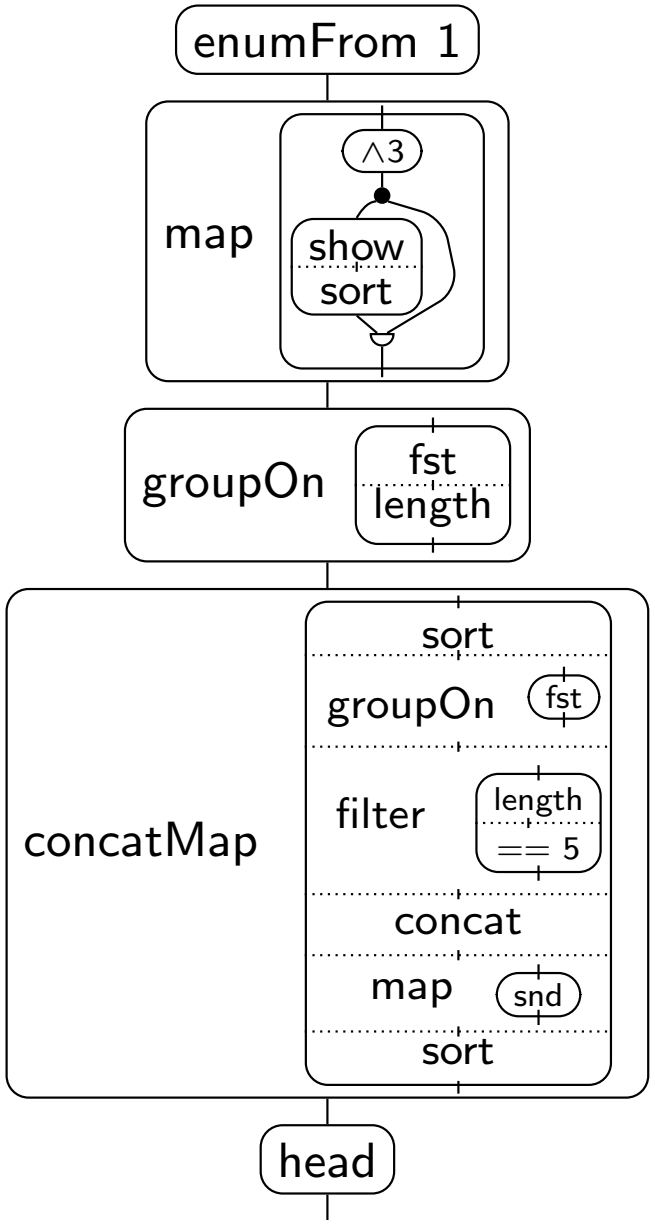






Structure to keep larger programs readable

- On the right a sketch of extreme use of stacking and nesting
- Most of the wires can be eliminated and most blocks grouped together
- Having multiple layers of structure is important for readability
- Coding style also makes big difference in visuals programming
- We are experimenting with subtle variations in wire style and coloring
- Grouped blocks in regions that can be named and collapsed is planned



Abstraction, lifting functor and applicative

Lifting functions into something is a very general abstraction

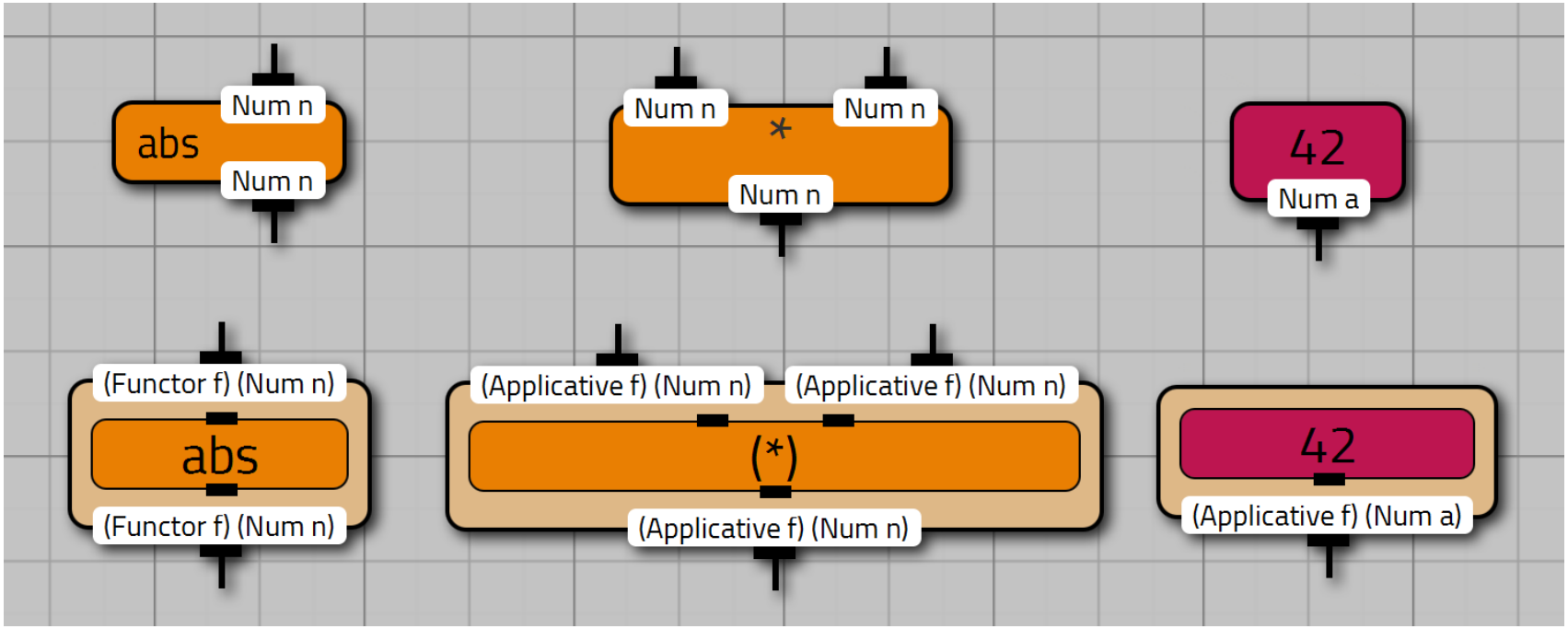
Generalized map: $fmap :: Functor\ f => (a \to b) \to f\ a \to f\ b$

Visually represented as a function block wrapped into another

Functions not having one argument become Applicative functors

Lifting is a standard user interaction on almost every block

The meaning of the lift depends on inner and outer block



Many interpretations of blocks and wires

Many interesting and useful abstactions are based around the concepts of composition, parallel composition, and lifting.
Some support more extras like choice, application and feedback
The relative position of blocks does matter for some abstractions

Examples of interpretations and translations

- Pure functions (without any special interpretation)
- Arrows and variants, including monads as Kleisli arrows
- Dataflow: blocks as actors and wires as channels
- Hardware description languages, especially asynchronous
- Concatenative/stack based languages (typed pure Forth)
- Lenses, where each block is some kind of projection

Future work

Going beyond a demo version

- Improved graphical design and more powerful user interactions
- Complete set of language constructs and a proper typesystem
- Better compiler (GHC) integration (using haskell-ide-engine?)
- Trying browser based user interface plus Haskell on a server
- Exploring other new input methods and display technologies
- Integrated refactoring and interactive debugging

Potential future directions

- As a FP introduction teaching tool, focus on interaction
- Start of a new language with more abstractions and types
- Programming for engineers: signal processing, hardware descriptions, modelling control systems

My observations

- Typed FP is the best basis for visual programming
- Live type feedback is more important than live execution
- Making larger visual programs readable is a solvable problem
- Interactions with a program matter more than its syntax
- Future visual languages will designed for new input methods
- Programming with blocks and wires is also an abstraction

The exploratory journey into visual FP has barely begun

We have no clue where this is going and so many ideas to try out
Viskell could be a good platform for experimental improvements

You are all invited to give feedback and suggestions

Questions?

Piqued your interest?

Try it yourself on the multi-touch table during the break.

Even more curious?

Pull the code from: https://github.com/wandernauta/viskell
Requires Java 8 with JavaFX, and GHC installed