A Type-Level Approach To Component Prototyping

Luís Barbosa Jácome Cunha Joost Visser

SYANCO 2007

3-4 September

Overview

Introduction

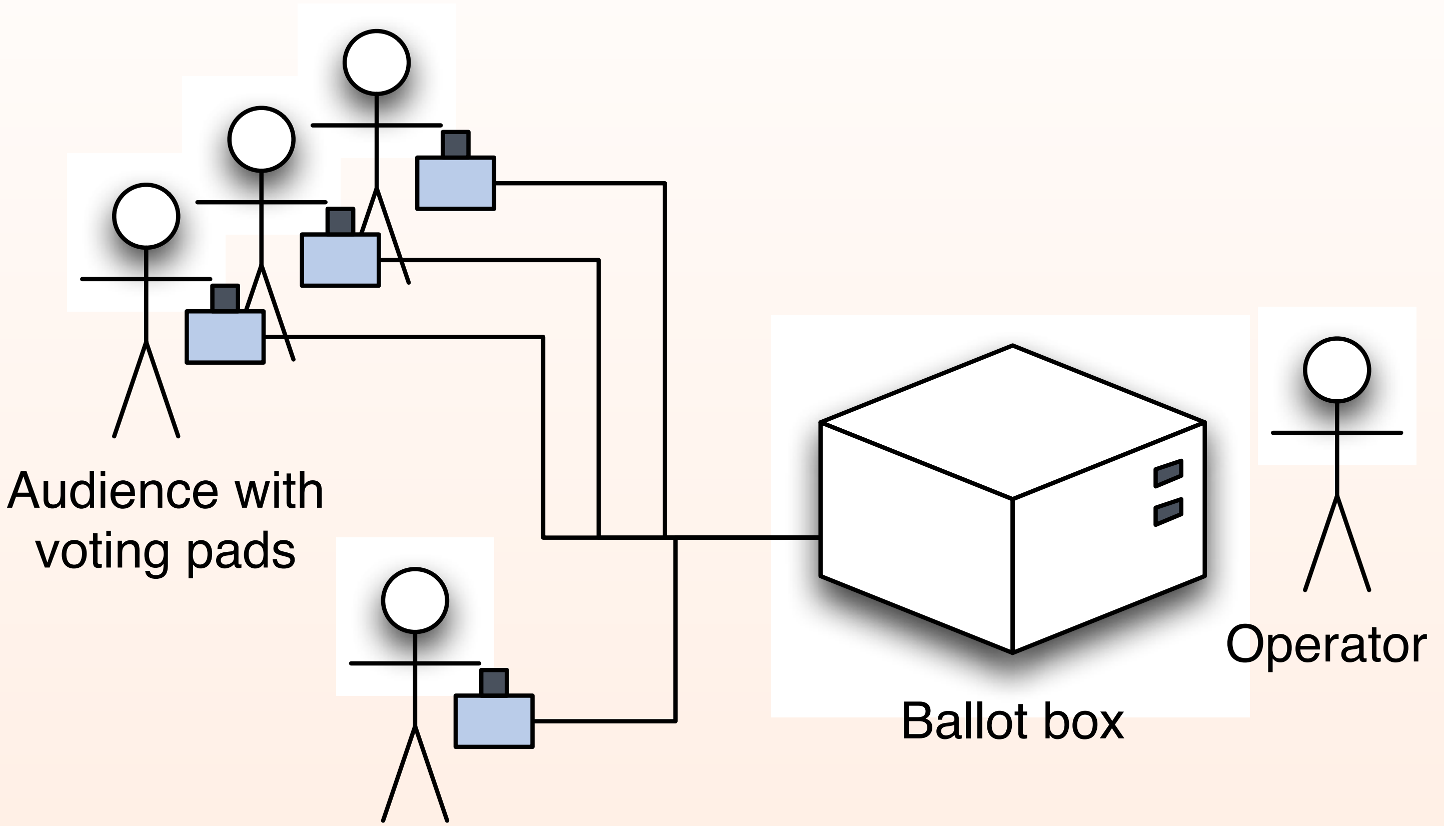
Motivation

Algebraic theories offer abstraction over specifics of component states and interfaces

General purpose PLs do not offer this level of abstraction

We bridge the gap between abstract component models and their type-safe implementation

Example



Components

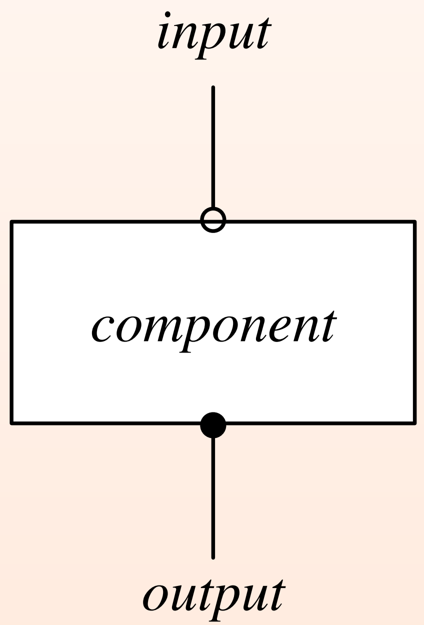
Component - I

Can be seen as a black box

Provided with an internal state

It has an input language

And an output language



Component - II

The general form: $cp : U \rightarrow I \rightarrow B(U \times O)$

- U represents the internal state
- I the input language
- O the output language
- B a behavioural monad, for example, one representing partiality, that is, allowing the component to fail

Components as coalgebras

Currying $cp : U \to I \to B(U \times O)$

We get $cp : U \to B(U \times O)^I$

Which is a coalgebra: $cp : U \to \top U$

Where $\top X = B(X \times O)^I$

Example

- The voting pad

  emit : $N \longrightarrow 1 \longrightarrow (N \times 1) + 1$

  emit $n * =$

  $\mathbf{if}\ n \not\equiv 0\ \mathbf{then}\ i_1\ (n-1, *)\ \mathbf{else}\ i_2 *$

- The ballot box

  reset : $N \longrightarrow N \longrightarrow B(N \times 1)$

  vote : $N \longrightarrow 1 \longrightarrow B(N \times 2)$

Encapsulation

Each operation can be seen as a pair of input/output ports. For example:

- emit: $1 \longrightarrow 1$
- reset: $N \longrightarrow 1$
- vote: $1 \longrightarrow 2$

Haskell

Type classes

Class: gather functions with the same signature, over a certain type

$$\textbf{class}\ Show\ a\ \textbf{where}$$
$$\quad show :: a \rightarrow String$$

An instance mechanism provides particular implementations for particular types

$$\textbf{instance}\ Show\ Bool\ \textbf{where}$$
$$\quad show\ True = \texttt{"T"}$$
$$\quad show\ False = \texttt{"F"}$$

Type-Level

$$\textbf{class}\ Convert\ a\ b \mid a \rightarrow b\ \textbf{where}$$
$$\quad convert :: a \rightarrow b$$

This class means that it is possible to uniquely transform the type a into the type b

It also gives us a function to transform values

The computation is performed at compile time, not in run time

HList

It is a strongly typed implementation of n-ary tuples which can contain things of different types

$$ex = HCons\ \texttt{"foo"}\ (HCons\ True\ HNil)$$

A set of operators like append two lists or zip two lists are provided

The Library

N-ary sums

Dual model of the presented n-ary tuples

Example: $HEither\ A\ (HEither\ B\ HVoid)$

Encode labelled sums (language)

$$HEither\ (Reset, N)\ (HEither\ (Vote, 1)\ HVoid)$$

With $inject$ and $select$ we can add and get things from these sums

Components

A component is encapsulated using the $Encapsulate$ class

This class infers the input and output language of the component and its state and behavioural monad

$$
\begin{aligned}
&vp = \lambda n \rightarrow (emit, emitf\ \ n) \mathbin{.*.} HNil \\
&\quad \textbf{where} \\
&\qquad emitf\ \ st\ () = \textbf{if}\ st \not\equiv 0 \\
&\qquad\qquad\qquad\qquad\ \textbf{then}\ Just\ (st - 1, ()) \\
&\qquad\qquad\qquad\qquad\ \textbf{else}\ Nothing
\end{aligned}
$$

Machine activation $DoCompIO$

- It turns a component into a interactive state machine
- The user starts the machine with an initial state and will be asked for actions

$$system > comp\ init\_state$$
$$Action : action1$$
$$result$$
$$Action : action2$$
$$result$$
$$\ldots$$

External choice $\boxplus$

- It allows the composition of two components
- Just one at each time can be activated, never both at the same time (the opposite of the parallel composition $\boxtimes$)
- The new language is a concatenation of the old languages tagged with $LEFT$ or $RIGHT$ according with the component where it belongs

Hook $\Lsh$

- It allows to feed back the component with (part of) its own output
- $(new\_act, (old\_act1, old\_act2)) \times \ldots$
- The language of input (output) must be of the form $Either\ i\ (o)\ z$
- While the result is of the type z the component continues

Wrap $[\,]$

- It wraps a component with input type i and output type o into a component with input type i' and output o'
- Two functions must be given: one of signature $f :: i' \longrightarrow i$ and another with type $g :: o \longrightarrow o'$
- It first uses f to transform the input to the correct one passing it to the original component and rewraps the output using g

Lift $\ulcorner\ \urcorner$

- It creates a component from a function
- A label must be supplied to name the action (which is the function)
- The state is the trivial one
- It allows the integration of existing functions into a component-based design

Pipeline ;

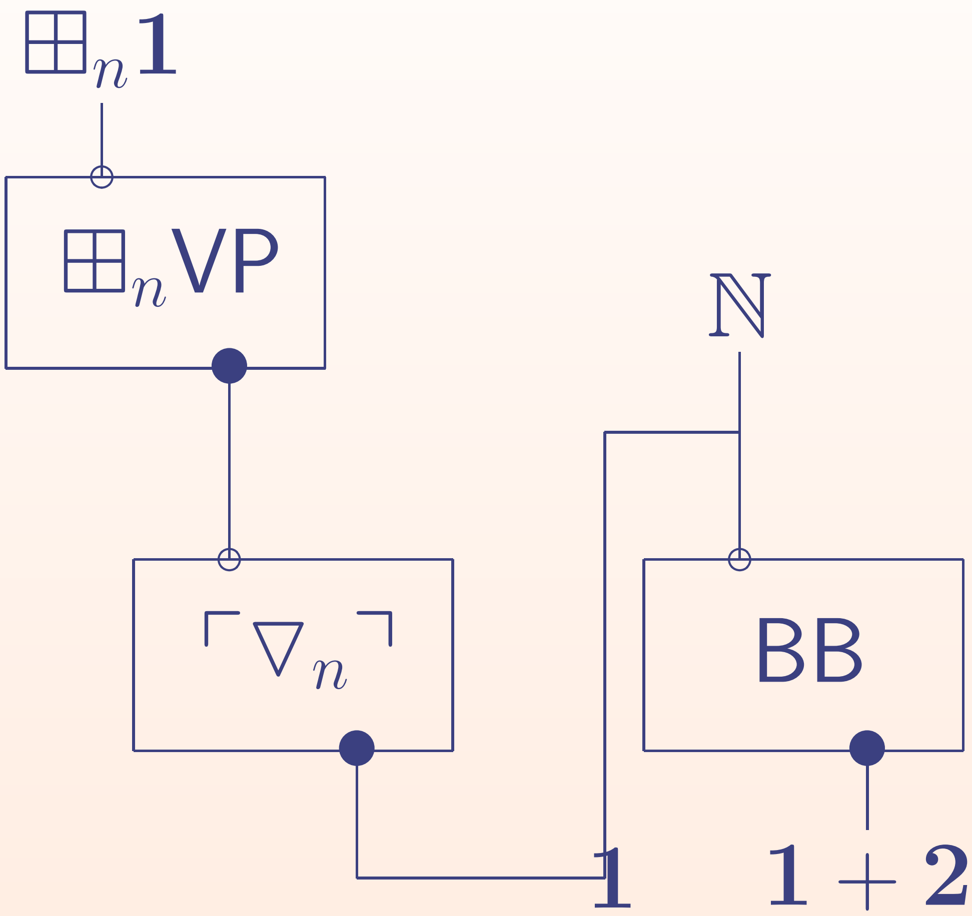
- It implements the sequential composition which is very important in this paradigm

- The output language of the first component must be the same as the input language of the second one

- It will execute the first component and the will pass the result to the second one

An example

Visually


$\boxplus_n \mathbf{1}$
$\boxplus_n$VP
$\mathbb{N}$
$\ulcorner \nabla_n \urcorner$
BB
$\mathbf{1}$
$\mathbf{1} + \mathbf{2}$

Formally

The formal definition is

$$VS_n = (((\boxplus_n VP; \ulcorner \nabla_n \urcorner) \boxplus BB) [a_+, s_+]) \Lsh_1$$

where VP is defined by the voting pad defined earlier and BB is defined the ballot box also already defined

Implementing I

Voting pad: creating a system with three voting pads:

$$vp3 = vp \boxplus vp \boxplus vp$$

where

$$
\begin{aligned}
& vp = \lambda n \rightarrow (emit, emitf\ n) \mathbin{.*.} HNil \\
& \quad \mathbf{where} \\
& \qquad emitf\ n\ () = \mathbf{if}\ n \neq 0 \\
& \qquad\qquad\qquad\qquad \mathbf{then}\ Just\ (n-1, ()) \\
& \qquad\qquad\qquad\qquad \mathbf{else}\ Nothing
\end{aligned}
$$

Implementing II

Ballot box: the ballot box component

$$
\begin{aligned}
bb = \lambda st \rightarrow\ & (reset, resetf\ \ st) \mathbin{.*.} \\
& (vote, votef\ \ st) \mathbin{.*.} HNil
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{where} \\
&\quad resetf\ \ n\ (Left\ rv) = Just\ (rv, Left\ ()) \\
&\quad votef\ \ n\ (Right\ \_) = \\
&\quad\quad Just\ (st - 1, Right\ (st - 1 \equiv 1))
\end{aligned}
$$

Implementing III

Voting system: just join the existent pieces

$$vs =^{\Lsh} (wrap\ (vp3 ; cod\ \boxplus\ bb)\ a_+\ s_+)\ hp$$

where

- a_+ is the implementation of the morphism which witnesses the sum associative law
- s_+ is the implementation of commutativity isomorphism for sum
- hp is the pattern needed to the hook
- cod id defined as $cpLift \bigtriangledown cod\_label$

Animating the vs

The input language of the component must be an instance of $Read$

$$
\begin{array}{l}
vsAnimation\ () = \\
\quad evalStateT\ \$\ doCompIO\ (\twoheadrightarrow\ vs)
\end{array}
$$

This function allows to run the component interactively as illustrated in the next slide

Running it

$$VS > vsAnimation\ ((((20, 33), 14), ()), 4)$$

$$Action : emit2$$

$$(Emit2, False)$$

$$Action : emit1$$

$$(Emit1, False)$$

$$Action : emit1$$

$$(Emit1, True)$$

$$Action : reset\ 3$$

$$(Reset, ())$$

$$\ldots$$

Conclusions and future work I

- We encoded a formal model for state-based components
- After modelling the components' model an algebraic suite of components was encoded
- The combinators can be neatly and effectively implemented in Haskell exploring techniques at the type level

Conclusions and future work II

- This provides a smooth way to directly incorporate components in Haskell
- It would be interesting to study how this library could take advantages using extensions as concurrency, mobility and distribution

Thank you!

Questions?