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.
Components
Component - I

Can be seen as a black box
Provided with an internal state
It has an input language
And an output language
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
Currying \( cp : U \rightarrow I \rightarrow B(U \times O) \)

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

Which is a coalgebra: \( cp : U \rightarrow \top U \)

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

- The voting pad
  \[\text{emit} : N \rightarrow 1 \rightarrow (N \times 1) + 1\]
  \[\text{emit } n \ast = \begin{cases} i_1(n - 1, \ast) & \text{if } n \not\equiv 0 \\ i_2 \ast & \text{else} \end{cases}\]

- The ballot box
  \[\text{reset} : N \rightarrow N \rightarrow B(N \times 1)\]
  \[\text{vote} : N \rightarrow 1 \rightarrow B(N \times 2)\]
Encapsulation

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

- emit: $1 \rightarrow 1$
- reset: $N \rightarrow 1$
- vote: $1 \rightarrow 2$
Haskell
Type classes

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

```haskell
class Show a where
    show :: a → String
```

An instance mechanism provides particular implementations for particular types

```haskell
instance Show Bool where
    show True = "T"
    show False = "F"
```
class Convert a b | a → b where
  convert :: a → 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

\[ ex = HCons \text{ "foo" } (HCons \text{ True } HNil) \]

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 \textit{inject} and \textit{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:

\[
vp = \lambda n \rightarrow (\text{emit}, \text{emitf } n) \ast . \text{HNil}
\]

where

\[
\text{emitf } st () = \begin{cases} 
\text{if } st \not\equiv 0 & \text{then } \text{Just } (st - 1, ()) \\
\text{else } \text{Nothing}
\end{cases}
\]
Machine activation \textit{DoCompIO}

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

\begin{verbatim}
system > comp init_state
Action : action1
result
Action : action2
result
...
\end{verbatim}
External choice

- 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 $\boxplus$)
- The new language is a concatenation of the old languages tagged with $LEFT$ or $RIGHT$ according with the component where it belongs
Hook

- 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 \(\text{Either } i (o) z\)

- While the result is of the type \(z\) the component continues
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 $f : i' \rightarrow i$ and another with type $g : o \rightarrow o'$.

It first uses $f$ to transform the input to the correct one passing it to the original component and rewraps the output using $g$. 

**Wrap**
Lift

- 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 then will pass the result to the second one.
An example
Visually
Formally

The formal definition is

\[ V_{S_n} = (((n VP; \neg \nabla_n \neg) \boxdot BB) [a_+, s_+]) \uparrow_1 \]

where \( VP \) is defined by the voting pad defined earlier and \( BB \) is defined the ballot box also already defined.
Voting pad: creating a system with three voting pads:

\[ vp_3 = vp \mp vp \mp vp \]

where

\[ vp = \lambda n \to (\text{emit}, \text{emitf } n) \ast HNil \]

where

\[ \text{emitf } n () = \begin{cases} \text{if } n \neq 0 & \text{then } \text{Just } (n - 1, ()) \\ \text{else } \text{Nothing} \end{cases} \]
**Ballot box:** the ballot box component

\[ bb = \lambda st \rightarrow (\text{reset}, \text{resetf } st) \ast \ast \ast \text{(vote, votef } st) \ast . \ast . \text{HNil} \]

where

\[ \text{resetf } n (\text{Left } rv) = \text{Just } (rv, \text{Left } ()) \]
\[ \text{votef } n (\text{Right } _) = \text{Just } (st - 1, \text{Right } (st - 1 \equiv 1)) \]
Implementing III

Voting system: just join the existent pieces

\[ vs = \downarrow (\text{wrap} (\text{vp}3; \text{cod} \boxplus \text{bb}) a_+ s_+) \text{ hp} \]

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
- \( \text{cod id} \) defined as \( \text{cpLift} \uplus \text{cod_label} \)
The input language of the component must be an instance of \textit{Read}.

\[
\text{vsAnimation}() = \text{evalStateT}$ doCompIO (\rightarrow \text{vs})
\]

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

\[ VS > \text{vsAnimation} (((((20, 33), 14), ()), 4) \]

\begin{align*}
\text{Action : } & \text{emit2} \\
& (\text{Emit2}, \text{False}) \\
\text{Action : } & \text{emit1} \\
& (\text{Emit1}, \text{False}) \\
\text{Action : } & \text{emit1} \\
& (\text{Emit1}, \text{True}) \\
\text{Action : } & \text{reset 3} \\
& (\text{Reset}, ())
\end{align*}
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?