A Type Level Approach to Component Prototyping

Jácome Miguel Cunha

jacome@di.uminho.pt

Universidade do Minho

CIC 2006

Outline

- Motivation
- Type-Level Programing
- PUReCAMILA
- Components and Coalgebras
- Constructing a Folder
- Conclusions and Future Work

Motivation

- The theoretical component model involves n-ary products and sums
- These are not commonly found as programming language constructs
- The type-system of Haskell allows to encode them
- This should bring the implementation closer to the theory

Type-Level Programming

The base rules:

- Typel-level *predicate*: **class** `P x`
- Type-level *relation*: **class** `R x y`
- Type-level *function*: **class** `F x y z | x y -> z`
  where `f :: x -> y -> z`
  (with value-level function `f`)

Classes work on the type level and its functions on the value level.

Example

Consider the following example:

data `Zero; zero = undefined :: Zero`
data `Succ n; succ = undefined :: n -> Succ n`

This data types are only labels.

class `Nat n`
instance `Nat Zero`
instance `Nat n => Nat (Succ n)`

With this class and the respective instances, we have a naturals representation.

Another Example

```
class Add a b c | a b -> c
    where add :: a -> b -> c

instance Add Zero b b
   where add a b = b

instance (Add a b c) =>
          Add (Succ a) b (Succ c)
   where add a b = succ (add (pred a) b)

pred :: Succ n -> n
pred = undefined
```

PURECAMILA

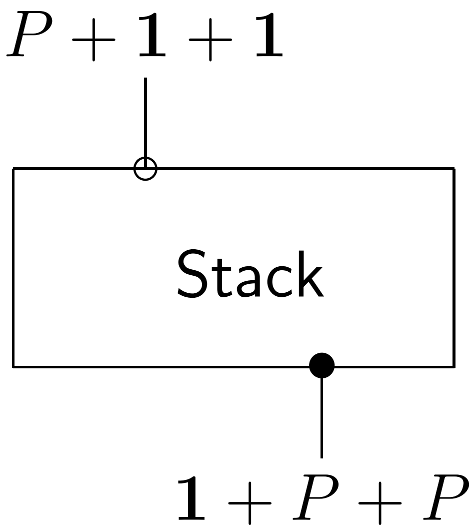
Some features of PURECAMILA

- Improvement of CAMILA, a prototyping system
- Implemented in HASKELL
- It has pre and post conditions, invariants and OO classes

Components

Let's look at this "stack":

$$\begin{array}{l} push : U \times P \longrightarrow U \\ pop : U \longrightarrow P \times U \\ top : U \longrightarrow P \end{array} \quad \xrightarrow{encapsulate} \quad \begin{array}{l} push : P \longrightarrow 1 \\ pop : 1 \longrightarrow P \\ top : 1 \longrightarrow P \end{array}$$



A Coalgebra?

Doing two renamings

- $I = P + \mathbf{1} + \mathbf{1}$
- $O = \mathbf{1} + P + P$

The stack can be represented by

$$Stack : U \times I \longrightarrow (U \times O + 1) \equiv Stack : U \longrightarrow (U \times O + 1)^I$$

Which is a coalgebra $U \longrightarrow \mathsf{T}\ U$ for the functor

$$\mathsf{T}\ X = ((X \times O) + 1)^I$$

The Input Type

The component input interface:

```
type Input = (PUSH, Int) :++: (POP, ()) :++:
             (TOP, ()) :++: HVoid
```

The function names are type-level labels, and the :++: and HVoid combinators build type-labeled n-ary sums.

```
class Sum l s x | l s -> x
  where select :: l -> s -> Maybe x
        inject :: l -> x -> s
```

The Output Type

The component output interface:

```
type Output s m = m (s, (PUSH, ()) :++: (POP, Int)
                       :++: (TOP, Int) :++: HVoid)
```

The output is parameterized in the state (`s`) and in the monad (`m`).

These two types (Input and Output) are easily manipulated with the `injection` and `selection` functions:

```
in = inject pop () :: Input
out = select pop in :: Maybe ()
```

The Stack Type

The stack type

```
type Stack s m = s -> (PUSH, Int -> m (s, ()))
                      (POP, () -> m (s, Int)) :*:
                      (TOP, () -> m (s, Int)) :*: HNil
```

The stack type is also parameterized in the state and in the monad.
The `:*:` represents the arbitrary-length tuple construction.

The Stack Component

The components must be constructed based on this stack model:

```
stack = \s -> (push, pushf s) .*. (pop, popf s) .*.
              (top, topf s) .*. HNil
  where
    pushf xs x       = return (x:xs, ())
    popf [] ()       = mzero
    popf (x:xs) ()  = return (xs, x)
    topf l ()        = return (l, head l)
```

The PassMessage

The hard work is done here:

```
class PassMessage s p s' | s p -> s'
    where passMessage :: s -> p -> s'

instance => PassMessage
        (HEither (l,e) is)
        (st -> (HCons (l', e -> m (st, r)) fs), st)
        (m (st, HEither (l', r) os))
```

It receives the input, the component itself paired with the state and returns a monadic pair with the new state and the output.

The Application Operator

The `@.` operator signature

```
(@.) :: ( CamilaMonad m, Sum l o1 o, Sum l it i,
          PassMessage it (cp, st) (m (st, o1)) )
 => cp -> it -> o1 -> st -> l -> i -> m (st, (l, o))
```

The PassaMessage is used here:

```
(@.) cp (_::int) (_::o) st l i = do
      let input = inject l i :: int
      (st', output) <- passMessage input (cp, st)
      let (Just out') = select l output
      return (st', (l, out'))
```

The Choice Operator – ⊞

Choice: allows to choose between two components

```
(|+|) :: (s1->l1) -> (s2->l2) -> ((s1, s2) -> lf)

c1 |+| c2 = \(s1, s2) -> toLeftLst c1 (s1, s2)
               `hAppend` toRightLst c2 (s1, s2)
```

where

- `hAppend` is the n-ary product concatenation
- `toLeftLst` is a function which transforms a simple component into a component that receives a pair of states and "`LEFT` labels" (`toRightLst` is it's dual)

The Hook Operator – ↰

This operator uses the component output to feed it back:

class `Hook ls s lf i o m | ls s lf m -> i o`
where
```
hook :: ls -> cp -> s -> lf -> i -> m (s, (lf, o))
```

In the next slide I'll show how to use it.

A Folder from two Stacks

```
folder =
hook ((tl, RIGHT top .*. LEFT push .*. HNil)
  .*. (tr, LEFT pop .*. RIGHT push .*. HNil).*.HNil)
      (stack |+| stack)
```

The user needs to specify the rules to the new operations.

Conclusions

With this approach

- We create a coalgebraic component implementation
- A suitable component algebra was/will be implemented
- It is now possible to construct new software components from old ones

Future Work

To be useful, there's much more to do:

- Finish the operators implementation (wrap, parallel, etc.)
- Animate components
- Add concurrency
- Add sockets
- ...