SHACC: A Functional Prototyper for a Component Calculus

André Martins[1], Luís S. Barbosa[1], and Nuno F. Rodrigues[2]

[1] DI - CCTC, University of Minho, 4710-057 Braga, Portugal
[2] DIGARC - IPCA 4750-810 Barcelos, Portugal

Abstract. Over the last decade component-based software development arose as a promising paradigm to deal with the ever increasing complexity in software design, evolution and reuse. SHACC is a prototyping tool for component-based systems in which components are modelled coinductively as generalized Mealy machines. The prototype is built as a HASKELL library endowed with a graphical user interface developed in *Swing*.

Keywords: software composition, Mealy machines, prototyping.

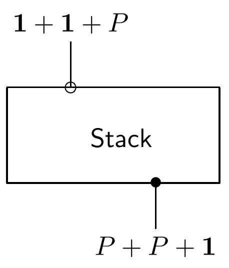
1 Introduction

SHACC is a HASKELL -based prototype for a calculus of state-based components framed as generalised Mealy machines detailed in [1,2]. A typical example of such a state-based component is the ubiquitous *stack*. Denoting by U its internal state, a stack of values of type P is handled through the usual

$$\mathsf{top} : U \longrightarrow P, \quad \mathsf{pop} : U \longrightarrow P \times U \quad \text{and} \quad \mathsf{push} : U \times P \longrightarrow U$$

operations. A 'black box' view, however, hides U from the stack environment and regards each operation as a pair of input/output ports. For example, the top operation becomes declared as $\mathsf{top} : \mathbf{1} \longrightarrow P$, where $\mathbf{1}$ stands for the nil (or unit) datatype. The intuition is that top is activated with the simple pushing of a 'button' (its argument being the stack private state space) whose effect is the production of a P value in the corresponding output port. Similarly typing push as $\mathsf{push} : P \longrightarrow \mathbf{1}$ means that an external argument is required on activation but no visible output is produced, but for a trivial indication of successful termination. Such 'port' signatures are grouped together in the diagram below. Combined input type $\mathbf{1} + \mathbf{1} + P$ models the choice among three functionalities (top, pop and push, in this order), of which only one takes input of type P.

$$\begin{cases} \mathsf{pop} : & \mathbf{1} \longrightarrow P \\ \mathsf{top} : & \mathbf{1} \longrightarrow P \\ \mathsf{push} : & P \longrightarrow \mathbf{1} \end{cases}$$



Component Stack encapsulates a number of services through a public *interface* providing limited access to its internal *state space*. Furthermore, it *persists* and *evolves* in time, in a way which can only be traced through observations at the interface level. One might capture these intuitions by providing an explicit semantic definition in terms of a function $[\![\mathsf{Stack}]\!] : U \times I \longrightarrow (U \times O + \mathbf{1})$, where I, O abbreviate $\mathbf{1} + \mathbf{1} + P$ and $P + P + \mathbf{1}$, respectively. The presence of $\mathbf{1}$ in its result type indicates that the overall behaviour of this component is *partial*: in a number of state configurations the execution of some operations may fail. Function $[\![\mathsf{Stack}]\!]$ describes how Stack reacts to input stimuli, produces output data (if any) and changes state. It can also be written in a curried form as

$$\overline{[\![\mathsf{Stack}]\!]} : U \longrightarrow (U \times O + \mathbf{1})^I$$

that is, as a *coalgebra* $U \longrightarrow \mathsf{T}\, U$ for functor $\mathsf{T}\, X = ((X \times O) + \mathbf{1})^I$.

The Stack example illustrates the basic elements of a semantic model for state-based components: *a)* the presence of an *internal state space* which evolves and persists in time, and *b)* the possibility of *interaction* with other components through well-defined interfaces and during the overall computation. This favours adoption of a *coalgebraic* modelling framework: components are inherently dynamic, possess an observable behaviour, but their internal configurations remain hidden and should be identified if not distinguishable by observation. The qualificative 'state-based' is used in the sense the word 'state' has in automata theory — the internal memory of the automaton which both constrains and is constrained by the execution of component operations. Such operations are encoded in a functor which constitutes the (syntax of the) component interface. Building on such a representation, reference [1] developed a calculus of component composition. The experimental tool SHACC presented here provides a HASKELL based prototyper for this calculus.

Outline. The following section provides a brief overview of the calculus and an example. The prototyping tool is described in section 3.

2 A Components' Calculus

Given a collection of sets I, O, ..., acting as component interfaces, a component taking input in I and producing output in O is specified by a pointed coalgebra

$$\langle u_p \in U_p, \overline{a}_p : U_p \longrightarrow \mathsf{B}(U_p \times O)^I \rangle$$

where u_p is the initial state, B a strong monad capturing the component behaviour model (*e.g.*, partiality, as above, or non determinism or ...), and the coalgebra dynamics is given by $a_p : U_p \times I \longrightarrow \mathsf{B}\ (U_p \times O)$. This definition means that the computation of an action in a component will not simply produce an output and a continuation state, but a B-structure of such pairs. The monadic structure provides tools to handle such computations. Unit (η) and multiplication (μ), act, respectively, as a value embedding and a 'flatten' operation to reduce nested behavioural effects. Strength, either in its right (τ_r) or left (τ_l) version, caters for context information.

References [1,2] introduce a small set of component combinators and study their properties. Their implementation in SHACC is parametric on the component behaviour discipline encoded in a monad B.

Components with compatible interfaces (for example, $p : I \longrightarrow K$ and $q : K \longrightarrow O$) can be composed sequentially as

$$p\,;q \;=\; \langle\langle u_p, u_q\rangle \in U_p \times U_q, \overline{a}_{p;q}\rangle$$

where $a_{p;q} : U_p \times U_q \times I \longrightarrow \mathsf{B}(U_p \times U_q \times O)$ is detailed as follows [1]

$$\begin{aligned} a_{p;q} \;=\; & U_p \times U_q \times I \xrightarrow{\times r} U_p \times I \times U_q \xrightarrow{a_p \times \mathsf{id}} \\ & \mathsf{B}(U_p \times K) \times U_q \xrightarrow{\tau_r} \mathsf{B}(U_p \times K \times U_q) \xrightarrow{\mathsf{B}(a\cdot \times r)} \\ & \mathsf{B}(U_p \times (U_q \times K)) \xrightarrow{\mathsf{B}(\mathsf{id} \times a_q)} \mathsf{B}(U_p \times \mathsf{B}(U_q \times O)) \\ & \xrightarrow{\mathsf{B}\tau_l} \mathsf{BB}(U_p \times (U_q \times O)) \xrightarrow{\mathsf{BB}a^\circ} \\ & \mathsf{BB}(U_p \times U_q \times O) \xrightarrow{\mu} \mathsf{B}(U_p \times U_q \times O) \end{aligned}$$

HASKELL monadic technology provides all the ingredients for a direct implementation of this definition, suitably parametric on a strong monad `b`. Each component is represented by a monadic function from pairs of state-input values to `b`-computations of state-output pairs. The HASKELL definition of each combinator in the calculus follows closely the corresponding mathematical construction, as illustrated below for sequential composition. Computation proceeds through Kleisli composition. Note, finally, that in order to guarantee state persistence (and propagation of state values) the implementation of SHACC resorts to HASKELL state monad which is suitably combined with monad `b` capturing the underlying behavioral model.

```
seqCompostion :: Strong b =>
    ((u,i)-> b (u,k)) -> ((v,k)-> b (v,o))
     -> ((u,v), i) -> b ((u,v),o)

seqCompostion p q = mult . (fmap (fmap assocl)). (fmap lstr).
                    (fmap (id >< q)) . (fmap xl).
                    rstr . (p >< id) . xr
```

The identity of sequential composition is component $\mathsf{copy}_K \;=\; \langle * \in \mathbf{1}, \overline{a}_{\mathsf{copy}_K}\rangle$, where $a_{\mathsf{copy}_K} \;=\; \eta_{\mathbf{1}\times K}$. The monoidal structure is expressed as bisimulation equations:

$$\begin{aligned} &\mathsf{copy}_I\,; p \;\sim\; p \;\sim\; p\,; \mathsf{copy}_O \\ &(p\,;q)\,; r \;\sim\; p\,;(q\,;r) \end{aligned}$$

[1] The definition resorts to standard isomorphisms, such as associativity (a) and exchange ($\times r : A \times B \times C \rightarrow A \times C \times B$, $xl : A \times (B \times C) \rightarrow B \times (A \times C)$), as well as to natural transformations $\tau_r : T \times - \Longrightarrow T(\mathsf{id} \times -)$ and $\tau_l : - \times T \Longrightarrow T(- \times \mathsf{id})$ denoting right and left monad strength.

Parallel composition, denoted by $p \boxtimes q$, corresponds to a synchronous product: both components are executed simultaneously when triggered by a pair of legal input values. Note, however, that the behavioral effect, captured by monad B, propagates. For example, if B can express component failure and one of the arguments fails, product fails as well. Two other tensors capture other forms of component aggregation: *external choice* $\boxplus$ and *concurrent* $\boxast$ composition. When interacting with $p \boxplus q : I + J \rightarrow O + R$, the environment chooses either to input a value of type I or one of type J, which triggers the corresponding component (p or q, respectively), producing the relevant output. In its turn, concurrent composition combines choice and parallel, in the sense that p and q can be executed independently or jointly, depending on the input supplied.

A *wrapping* mechanism $p[f, g]$ which encodes the pre- and post-composition of a component with a function is defined as a combinator which generalises the renaming connective found in process algebras. Moreover, any function $f : A \longrightarrow B$ can be lifted to a component whose interfaces are given by their domain and codomain types. Formally, a function $f : A \longrightarrow B$ gives rise to component $\ulcorner f \urcorner = \langle * \in \mathbf{1}, \overline{a}_{\ulcorner f \urcorner} \rangle$ *i.e.*, a coalgebra over $\mathbf{1}$ whose action is given by currying $a_{\ulcorner f \urcorner} = \mathsf{B}(\mathbf{1} \times B) \cdot (\mathsf{id} \times f)$.

Finally, generalized interaction is catered through a sort of "feedback" mechanism on a subset of the inputs. This is defined by a combinator, called *hook*, which connects some input to some output wires and, consequently, forces part of the output of a component to be fed back as input. Formally, the *hook* combinator $- \Lsh_Z$ maps each component $p : I + Z \longrightarrow O + Z$ to $p \Lsh_Z : I + Z \longrightarrow O + Z$.

3 The SHACC Tool

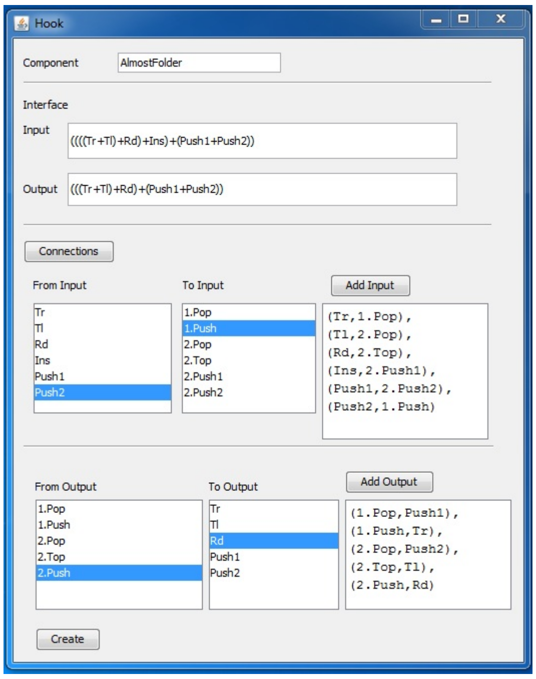
The SHACC was developed as a *proof-of-concept* prototype for the component calculus proposed in [2,1]. It allows the (interactive) definition of state-based components through the set of combinators available in the calculus: Figure 1 illustrates the application of the *hook* combinator to link a user-specified number of ports with opposite polarity.

The definition of a new, base component is directly made in HASKELL . A specific strong monad B is chosen to model the envisaged behavioral effect. The code below corresponds to a Stack component, where B is instantiated to HASKELL `Maybe` monad to capture partiality.

```
stack  (xs, ("Push", Just a))  = Just ( a:xs, ("Push", a))
stack  (xs, ("Pop", Nothing))
       |(xs == [])  = Nothing
       | otherwise  = Just ( tail xs, ("Pop", head xs))
stack  (xs, ("Top", Nothing))
       |(xs == [])  = Nothing
       | otherwise  = Just (  xs, ("Top", head xs))
```

In a subsequent step the component's interface is created from a suitable annotation in the source code. For this example:

```
@Input: (( 1:Pop + 1:Top) + P:Push)
@Output:(( P:Pop + P:Top) + 1:Push)
```



Component: AlmostFolder

Interface

Input: ((((Tr+Tl)+Rd)+Ins)+(Push1+Push2))

Output: (((Tr+Tl)+Rd)+(Push1+Push2))

From Input: Tr, Tl, Rd, Ins, Push1, Push2

To Input: 1.Pop, 1.Push, 2.Pop, 2.Top, 2.Push1, 2.Push2

```
(Tr,1.Pop),
(Tl,2.Pop),
(Rd,2.Top),
(Ins,2.Push1),
(Push1,2.Push2),
(Push2,1.Push)
```

From Output: 1.Pop, 1.Push, 2.Pop, 2.Top, 2.Push

To Output: Tr, Tl, Rd, Push1, Push2

```
(1.Pop,Push1),
(1.Push,Tr),
(2.Pop,Push2),
(2.Top,Tl),
(2.Push,Rd)
```

Fig. 1. Linking ports through the *hook* combinator

where `Pop`, `Top` and `Push` are introduced as labels for the component's available services.

Figure 1 refers to an example from the SHACC library, in which a virtual version of a paper folder is built through the combination of two stacks modelling, respectively, the folder left and right piles.

The Folder component provides ports corresponding to the operations *read*, *insert* a new page, *turn a page right* and *turn a page left*. Its construction requires first an adaptation to be performed on each instance of the Stack component. This is needed, for example, to hide the *top* operation on the left stack whereas renaming the *top* on the right as the Folder *read* operation. In a second stage, both stacks are put together through the $\boxplus$ combinator and, finally, suitable feedback loops are established, through the *hook* operator, to connect ports. This ensures, for example, that the left turn of a page is achieved through a *pop* performed on the right stack connected to a *push* on the left one. Formally, this amounts to the following expression in the component calculus (see [4] for a detailed discussion)

$$\mathsf{Folder} \;=\; ((\mathsf{LeftS} \boxplus \mathsf{RightS})[wi, wp]) \;{}^{\Lsh}{}_{P+P}$$

where $\mathsf{RightS} = \mathsf{Stack}[id + \triangledown, id]$ and $\mathsf{LeftS} = \mathsf{Stack}[i_2 + Id, (id + !_{p+1}) \cdot a_+]$.

A crucial ingredient in defining Folder is to suitably wrapp the two underlying Stack components so that the intended output-input ports are effectively connected. Formally this is achieved through the *wrapping* combinator, as in the specification of LeftS and

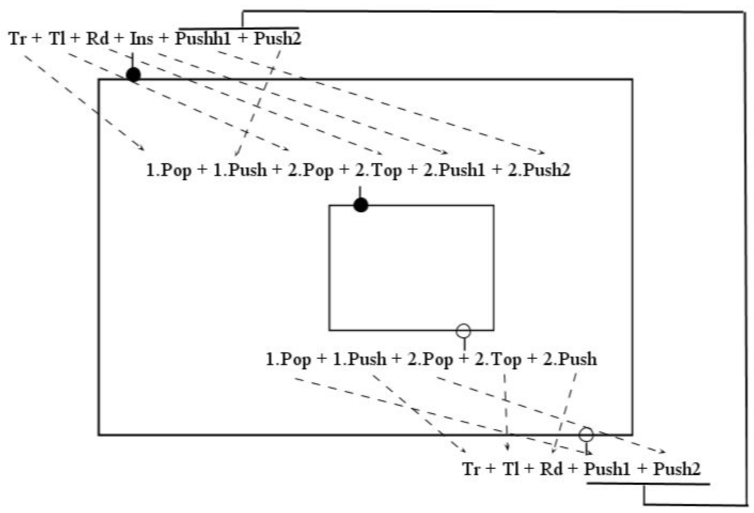
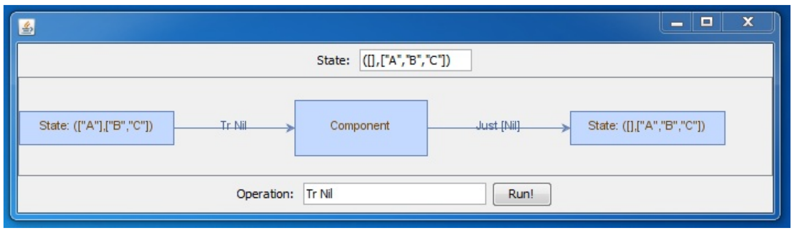

Tr + Tl + Rd + Ins + Pushh1 + Push2
1.Pop + 1.Push + 2.Pop + 2.Top + 2.Push1 + 2.Push2
1.Pop + 1.Push + 2.Pop + 2.Top + 2.Push
Tr + Tl + Rd + Push1 + Push2

Fig. 2. Linking output to input ports

RightS. The effect is depicted in Figure 2. In SHACC, however, the user has the option of manually selecting the ports to be linked, as illustrated in Figure 1.

SHACC allows both the (interactive) definition of this sort of component expressions and their execution in a simulation mode. Actually, once components are defined either from scratch (*i.e.*, by providing the corresponding HASKELL code directly) or by composition of other components, SHACC offers an environment for testing by simulation. The *Run* window in the tool offers two simulation modes: a *free* mode in which, if the component's behaviour model allows, execution may lead to 'disaster' (*e.g.*, by violation of port pre-conditions on a *partial* component), and a *safe* mode in which the effect of a port operation is foreseen and eventually precluded. Component testing, on the other hand, can be made in a purely interactive way, running event by event, or by executing a whole sequence of events specified through a regular expression and supplied to the tool. Figure 3 illustrates the tool execution mode.

The box labelled *State* in Figure 3 shows the initial value of the component's state. Box *Operation*, on the other hand, accepts the component service to be called. On


State: ([],["A","B","C"])

State: (["A"],["B","C"]) — Tr Nil → Component — Just [Nil] → State: ([],["A","B","C"])

Operation: Tr Nil

Fig. 3. Component prototyping in SHACC

executing a service from the component's interface SHACC displays three boxes representing the component state before, during and after service completion.

The SHACC tool is composed of a HASKELL combinator library and a graphical user interface developed in Swing. The choice of HASKELL was motivated by its expressiveness and extensibility, which provides an ideal means to support domain specific languages. Most important was the direct encoding in HASKELL 's 'monadic technology' of the entire component representation and manipulation. In particular, one resorted to HASKELL 's readily available *state monad* implementation, for storing the internal state of components being executed. This, together with HASKELL 's specific *do notation* for monadic type values manipulation, greatly reduced the effort of implementing the prototyper and its different execution modes.

Another important implementation detail, again resorting to monadic technology, is error detection and handling in a way which conforms to the underlying behavioral model. according to the execution mode). As already explained, in the Folder example above this resorts to the native `Maybe` data type, which forms an instance of the monad class, thus allowing for error propagation and detection at each specific point of component execution.

Finally, integration with *Swing*, to provide a user-friendly interface, proved effective.

Availability. SHACC is available from `shacc.wetpaint.com`. For the underlying calculus see references [2,1,4]. A refinement theory for this sort of component models is documented in [5,3].

References

1. Barbosa, L.S.: Towards a calculus of state-based software components. Journal of Universal Computer Science 9(8), 891–909 (2003)
2. Barbosa, L.S., Oliveira, J.N.: State-based components made generic. In: Peter Gumm, H. (ed.) CMCS 2003. Elect. Notes in Theor. Comp. Sci., vol. 82.1. Elsevier, Amsterdam (2003)
3. Barbosa, L.S., Oliveira, J.N.: Transposing partial components: An exercise on coalgebraic refinement. Theor. Comp. Sci. 365(1-2), 2–22 (2006)
4. Barbosa, L.S., Sun, M., Aichernig, B.K., Rodrigues, N.: On the semantics of componentware: A coalgebraic perspective. In: He, J., Liu, Z. (eds.) Mathematical Frameworks for Component Software, pp. 69–117. World Scientific, Singapore (2006)
5. Meng, S., Barbosa, L.S.: Components as coalgebras: The refinement dimension. Theor. Comp. Sci. 351, 276–294 (2005)