# Objectification — from functional to state-based models

J.N. Oliveira

Dept. Informática,
Universidade do Minho
Braga, Portugal

DI/UM, 2007

# Functional modeling

Consider the following model of a *stack*:

- Datatype:

$$Stack\ A \triangleq A^{\star}$$

- Functionality

$$\begin{array}{l} empty : Stack\ A \to \mathbb{B} \\ empty\ s \triangleq s = [\,] \end{array}$$

$$\begin{array}{l} push : A \to Stack\ A \to Stack\ A \\ push\ a\ s \triangleq a : s \end{array}$$

$$\begin{array}{l} Pop : Stack\ A \to Stack\ A \\ Pop\ s \triangleq tail\ s \\ \mathbf{pre}\ \neg(empty\ s) \end{array}$$

$$\begin{array}{l} Top : Stack\ A \to A \\ Top\ s \triangleq head\ s \\ \mathbf{pre}\ \neg(empty\ s) \end{array}$$

$$\begin{array}{l} clear : Stack\ A \to Stack\ A \\ clear\ s \triangleq [\,] \end{array}$$

# Questions

- Is this the *only* way to specify a stack?
- Compare with, at programming level
    - functional program (eg. in Haskell)
    - imperative program (eg. in C)
    - object oriented program (eg. in Java)
- How do we *bridge the gap* between such an abstract model and other models closer to such programming languages?

# Objectification

- Process of inferring object class models from a purely functional models
- Based on Coad and Yourdon's principle:

  > *The potential class must have a set of identifiable operations that can change the value of its attributes in some way.* [1]

- One needs to identify what Coad and Yourdon mean by **attributes**
- More generally, one needs to identify the **state** space of an **automaton**

# About automata

Given a set $A$ (input alphabet), a set $B$ (output alphabet) and a set of states $S$, a Deterministic Mealy Machine (DMM) is specified by a transition function of type

$$\delta \quad : \quad A \to (S \to (B \times S))$$

Wherever $(b, s') = \delta\ a\ s$, we say that there is a transition

$$s' \xleftarrow{a|b} s$$

and refer to $s$ as the **before** state, and to $s'$ as the **after** state.

# First step: identify the DMM

Analysis of functionality of example given shows:

- All involve either an argument or result of type $Stack\ A$
- There is at least one function where $Stack\ A$ is the type of both an argument and the result (two in fact: $push$ and $Pop$.)
- Easy to see that eg.

$$push \quad : \quad A \rightarrow (Stack\ A \rightarrow (1 \times Stack\ A))$$

  is itself a **DMM** (note the 1 signaling the *empty* output) whose **state** is of type $Stack\ A$
- Other functionality can be converted into DMMs by adding 1s where needed, eg.

$$clear \quad : \quad 1 \rightarrow (Stack\ A \rightarrow (1 \times Stack\ A))$$

  (note the *empty* input this time)

# First step: identify the DMM

- Other functionality can be converted into DMMs by explicitly declaring that the state doesn't change, eg.

$$\begin{aligned} & Top : 1 \to (Stack\ A \to (A \times Stack\ A)) \\ & Top\ \_\ s \triangleq (head\ s, s) \\ & \mathbf{pre}\ \neg(empty\ s) \end{aligned}$$

Altogether

- we can build an **object** as a composite DMM which encompasses the whole functionality,
- whose **state** is of type $Stack\ A$ and where
- $push$, $Pop$ and $clear$ *modify* the state (they **wr**ite on it)
- $Top$ and $empty$ *read* (abbrev. **rd**) the state only

# Comments

- Builiding the DMM as above is the right (formal) way but involves a number of technical details [2] which it is wise to ignore for the time being
- Below we head for a *practical* method based on **pre**/**post**-conditions
- So we go for implict specifications

# Second step: go implicit

$empty : (s : Stack\ A) \to (r : \mathbb{B})$
**post** $r = (s = [\,])$

$push : (a : A) \to (s : Stack\ A) \to (r : Stack\ A)$
**post** $r = a : s$

$Pop : (s : Stack\ A) \to (r : Stack\ A)$
**pre** $\neg(empty\ s)$
**post** $r = tail\ s$

$Top : (s : Stack\ A) \to (r : A)$
**pre** $\neg(empty\ s)$
**post** $r = head\ s$

$clear : (s : Stack\ A) \to (r : Stack\ A)$
**post** $r = [\,]$

# Third step: factor out state

A way to indicate that $Stack\ A$ is the DMM's state is to drop this from the signatures while marking each operation as a state **reader** or state **writer**:

$$empty :\rightarrow (r : \mathbb{B})$$
$$\textbf{rd}\ s : Stack\ A$$

$$Top :\rightarrow (r : A)$$
$$\textbf{rd}\ s : Stack\ A$$

$$push : (a : A) \rightarrow$$
$$\textbf{wr}\ s : Stack\ A$$

$$clear :\rightarrow$$
$$\textbf{wr}\ s : Stack\ A$$

$$Pop :\rightarrow$$
$$\textbf{wr}\ s : Stack\ A$$

# Notation: state readers

By writing

$$
\begin{aligned}
& OP : (b : B) \leftarrow (a : A) \\
& \mathbf{rd}\ s : St \\
& \mathbf{pre}\ \mathit{precond}(s, a) \\
& \mathbf{post}\ \ \mathit{postcond}(s', b, s, a)
\end{aligned}
$$

we mean an operation which does not modify the state:

$$
\begin{array}{rcl}
\text{pre-}OP & : & St \times A \to \mathbb{B} \\
\text{pre-}OP(s, a) & \triangleq & \mathit{precond}(s, a)
\end{array}
$$

$$
\begin{array}{rcl}
\text{post-}OP & : & St \times B \times St \times A \to \mathbb{B} \\
\text{post-}OP(s', b, s, a) & \triangleq & \mathit{postcond}(s', b, s, a) \ \wedge \ s' = s
\end{array}
$$

## Notation: state writers

By writing

$$\begin{array}{l} OP : (b : B) \leftarrow (a : A) \\ \mathbf{wr}\ s : St \\ \mathbf{pre}\ \ precond(s, a) \\ \mathbf{post}\ \ postcond(s', b, s, a) \end{array}$$

we mean

$$\begin{array}{rcl} \text{pre-}OP & : & St \times A \rightarrow \mathbb{B} \\ \text{pre-}OP(s, a) & \triangleq & precond(s, a) \end{array}$$

$$\begin{array}{rcl} \text{post-}OP & : & St \times B \times St \times A \rightarrow \mathbb{B} \\ \text{post-}OP(s', b, s, a) & \triangleq & postcond(s', b, s, a) \end{array}$$

that is, condition $s' = s$ is dropped.

# Fourth step: merge and rename

Readers and writers can be combined so as to build a DMM whose transitions involve operations which both yield a result **and** modify the state:

$EMPTY :\rightarrow (b : \mathbb{B})$
**rd** $s : Stack\ A$
**post** $b = (empty\ s)$

$PUSH : (a : A) \rightarrow$
**wr** $s : Stack\ A$
**post** $s' = a : s$

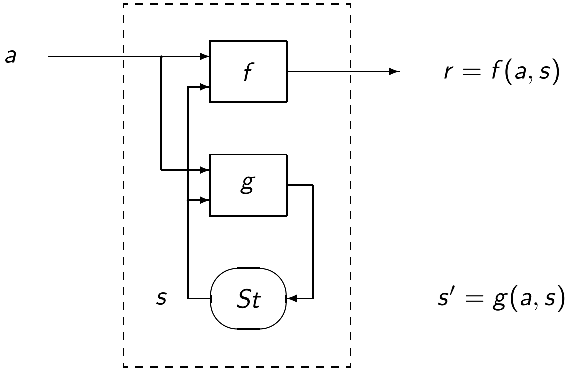
$POP :\rightarrow (r : A)$
**wr** $s : Stack\ A$
**pre** $\neg(empty\ s)$
**post** $s' = tail\ s \wedge r = head\ s$

$TOP : Stack\ A \rightarrow A$
**rd** $s : Stack\ A$
**pre** $\neg(empty\ s)$
**post** $r = head\ s$

$CLEAR :\rightarrow$
**wr** $s : Stack\ A$
**post** $s' = [\,]$

# Combining functions to build writers

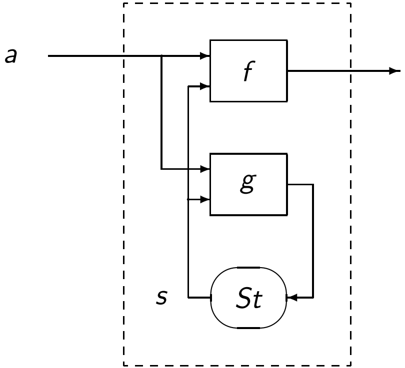
The "output first" pattern:




**post** $r = f(a, s) \wedge s' = g(a, s)$

# Combining functions to build writers

The "update first" pattern:




$r = f(a, s')$

$s' = g(a, s)$

$$\textbf{post} \quad s' = g(a, s) \wedge r = f(a, s')$$

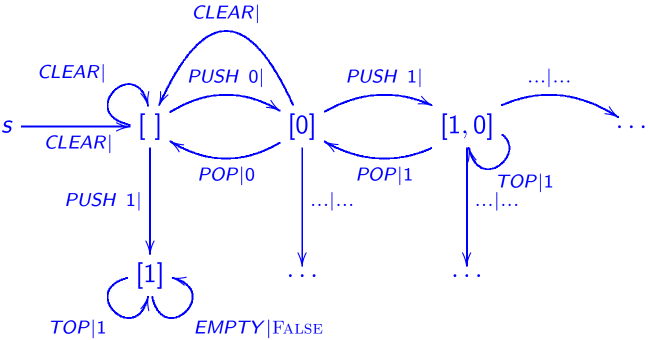
# Example of *update first* writer

A cash-point operation:

$$\begin{aligned}&\mathit{DEBIT} : (m : \mathit{Amount}) \rightarrow (r : \mathit{Receipt})\\ &\textbf{wr}\ s : \mathit{Account}\\ &\textbf{pre}\ m \leq \mathit{balance}\ s\\ &\textbf{post}\ \ s' = \mathit{debit}\ m\ s\ \wedge\ r = \mathit{balance}\ s'\end{aligned}$$

# DMM semantics

- The behaviour of the *Stack* DMM is defined as the set of all state transitions which can take place as dictated by **pre**/**post**-condition pairs.
- Example: for $A = \{0, 1\}$, $B = A \cup \mathbb{B}$, the state transition diagram will include




etc

# Behavioural safety and nondeterminism

Note that

- state transition diagram rules out all transitions whose before-states violate pre-conditions
- in general, there may exist operations such as eg.

$$
\begin{array}{l}
Pick :\longrightarrow (x : Marble) \\
\textbf{wr } b : Bag \\
\textbf{pre } b \neq \{\} \\
\textbf{post } \; x \in b \; \wedge \; b' = b - \{x\}
\end{array}
$$

So, in general, Mealy machines can be nondeterministic.

# Proof obligation

For every

$$
\begin{array}{l}
OP : (b : B) \leftarrow (a : A) \\
\textbf{wr}/\textbf{rd}\ s : St \\
\textbf{pre}\ \ldots \\
\textbf{post}\ \ \ldots
\end{array}
$$

where $St$, $A$ and $B$ are subject to invariants, one is obliged to discharge the following proof:

## Satisfiability

$$
\left\langle
\begin{array}{c}
\forall\ s, a\ \ : \\
s \in St \wedge a \in A : \\
\text{pre-}OP(s, a) \Rightarrow \langle \exists\ s', b\ \ :\ \ s' \in St \wedge b \in B :\ \ \text{post-}OP(s', b, s, a) \rangle
\end{array}
\right\rangle
$$

# Back to functions

DMMs can be built and animated using the state monad:

- Recall

$$\delta \quad : \quad A \to \underbrace{(S \to (B \times S))}_{ST\ S\ B}$$

- Every function of type $ST\ S\ B$ will be referred to as a **state transformer**
- For a fixed state space $S$, $F \stackrel{\text{def}}{=} ST\ S$ can be turned into a **monad**
- *Split* combinator $\langle f, g \rangle a \triangleq (f\ a, g\ a)$ useful in building state transformers

# Building state transformers

Update state:

$$
\begin{aligned}
&\mathit{update} : (S \rightarrow S) \rightarrow ST\ S\ 1 \\
&\mathit{update}\ f \triangleq \langle !, f \rangle
\end{aligned}
$$

Query the state:

$$
\begin{aligned}
&\mathit{query} : (S \rightarrow B) \rightarrow ST\ S\ B \\
&\mathit{query}\ f \triangleq \langle f, \mathit{id} \rangle
\end{aligned}
$$

Return a result:

$$
\begin{aligned}
&\mathit{return} : B \rightarrow ST\ S\ B \\
&\mathit{return}\ b \triangleq \langle \underline{b}, \mathit{id} \rangle
\end{aligned}
$$

# Combining state transformers

Sequential composition:

$$seq : ST\ S\ A \to ST\ S\ B \to ST\ S\ B$$
$$seq\ f\ g \triangleq do\ \{f; g\}$$

"Update first" transformer:

$$updfst : (A \to S \to S) \to (A \to S \to B) \to A \to ST\ S\ B$$
$$updfst\ g\ f\ a \triangleq do\ \{update(g\ a); query(f\ a)\}$$

"Query first" transformer:

$$qryfst : (A \to S \to S) \to (A \to S \to B) \to A \to ST\ S\ B$$
$$qryfst\ g\ f\ a \triangleq do\ \{b \leftarrow query(f\ a); update(g\ a); return\ b\}$$

# Animating state transformers

Running:

$$run : ST\ S\ A \rightarrow S \rightarrow (A \times S)$$
$$run\ g\ s \triangleq\ g\ s$$

Example: given $POP \triangleq\ qryfst\ head\ tail$, by running $POP$ over state $[1, 2, 3]$ one obtains

$$run\ POP\ [1, 2, 3] \quad = \quad (1, [2, 3])$$

This **reactive** behaviour can only be animated for DMMs. Nondeterminism requires explicit use of test suites guided by post-conditions.

P. Coad and E. Yourdon.
*Object-Oriented Analysis*.
Prentice Hall, 2nd edition, 1991.

A. Cruz, L. Barbosa, and J. Oliveira.
From algebras to objects: Generation and composition.
*Journal of Universal Computer Science*, 11(10):1580–1612, 2005.