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:

  \[ \text{Stack } A \triangleq A^* \]

- Functionality

  \( \text{empty} : \text{Stack } A \rightarrow \mathbb{B} \)
  \[ \text{empty } s \triangleq s = [ ] \]

  \( \text{push} : A \rightarrow \text{Stack } A \rightarrow \text{Stack } A \)
  \[ \text{push } a s \triangleq a : s \]

  \( \text{Pop} : \text{Stack } A \rightarrow \text{Stack } A \)
  \[ \text{Pop } s \triangleq \text{tail } s \]

  \( \text{clear} : \text{Stack } A \rightarrow \text{Stack } A \)
  \[ \text{clear } s \triangleq [ ] \]
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

$$\delta : A \rightarrow (S \rightarrow (B \times S))$$

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.

\[
push : A \rightarrow (Stack A \rightarrow (1 \times Stack A))
\]

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.

\[
clear : 1 \rightarrow (Stack A \rightarrow (1 \times Stack A))
\]

(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.

\[
\text{Top} : 1 \rightarrow (\text{Stack } A \rightarrow (A \times \text{Stack } A))
\]

\[
\text{Top } s \triangle (\text{head } s, s)
\]

\[
\text{pre } \neg (\text{empty } s)
\]

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 **write** on it)
- **Top** and **empty** **read** (abbrev. **rd**) the state only
Comments

- Building 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 implicit specifications.
Second step: go implicit

empty : (s : Stack A) → (r : IB)
post  r = (s = [ ])

push : (a : A) → (s : Stack A) → (r : Stack A)
post  r = a : s

Pop : (s : Stack A) → (r : Stack A)
pre  ¬(empty s)
post  r = tail s

Top : (s : Stack A) → (r : A)
pre  ¬(empty s)
post  r = head s

clear : (s : Stack A) → (r : Stack A)
post  r = [ ]
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:

\[
\begin{align*}
\text{empty} & : \rightarrow (r : \mathbb{B}) \\
\text{rd } s & : \text{Stack } A \\
\text{push} : (a : A) & \rightarrow \\
\text{wr } s & : \text{Stack } A \\
\text{Pop} & : \rightarrow \\
\text{wr } s & : \text{Stack } A \\
\text{Top} & : \rightarrow (r : A) \\
\text{rd } s & : \text{Stack } A \\
\text{clear} & : \rightarrow \\
\text{wr } s & : \text{Stack } A
\end{align*}
\]
Notation: state readers

By writing

\[
OP : (b : B) \leftarrow (a : A)
\]

\[
rd \ s : St
\]

\[
\text{pre} \ \text{precond}(s, a)
\]

\[
\text{post} \ \text{postcond}(s', b, s, a)
\]

we mean an operation which does not modify the state:

\[
\text{pre-} OP \ : \ St \times A \to \mathbb{B}
\]

\[
\text{pre-} OP(s, a) \triangleq \text{precond}(s, a)
\]

\[
\text{post-} OP \ : \ St \times B \times St \times A \to \mathbb{B}
\]

\[
\text{post-} OP(s', b, s, a) \triangleq \text{postcond}(s', b, s, a) \land s' = s
\]
Notation: state writers

By writing

\[ \text{OP} : (b : B) \leftarrow (a : A) \]
\[ \text{wr } s : St \]
\[ \text{pre } \text{precond}(s, a) \]
\[ \text{post } \text{postcond}(s', b, s, a) \]

we mean

\[ \text{pre-OP} : St \times A \rightarrow \mathbb{B} \]
\[ \text{pre-OP}(s, a) \triangleq \text{precond}(s, a) \]

\[ \text{post-OP} : St \times B \times St \times A \rightarrow \mathbb{B} \]
\[ \text{post-OP}(s', b, s, a) \triangleq \text{postcond}(s', b, s, a) \]

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:

\[ \text{EMPTY} : \rightarrow (b : \mathbb{I}B) \]  
\[ \text{rd} s : \text{Stack A} \]  
\[ \text{post} \quad b = (\text{empty } s) \]

\[ \text{PUSH} : (a : A) \rightarrow \]  
\[ \text{wr} s : \text{Stack A} \]  
\[ \text{post} \quad s' = a : s \]

\[ \text{POP} : \rightarrow (r : A) \]  
\[ \text{wr} s : \text{Stack A} \]  
\[ \text{pre} \quad \neg (\text{empty } s) \]  
\[ \text{post} \quad s' = \text{tail } s \land r = \text{head } s \]

\[ \text{TOP} : \text{Stack A} \rightarrow A \]  
\[ \text{rd} s : \text{Stack A} \]  
\[ \text{pre} \quad \neg (\text{empty } s) \]  
\[ \text{post} \quad r = \text{head } s \]

\[ \text{CLEAR} : \rightarrow \]  
\[ \text{wr} s : \text{Stack A} \]  
\[ \text{post} \quad s' = [ ] \]
Combining functions to build writers

The “output first” pattern:

\[ a \rightarrow f(a, s) \rightarrow r = f(a, s) \]

\[ s \rightarrow g(a, s) \rightarrow s' = g(a, s) \]

\[ \text{post } r = f(a, s) \land s' = g(a, s) \]
The “update first” pattern:

\[
\begin{align*}
    a & \quad \rightarrow \quad f \quad \rightarrow \quad r = f(a, s') \\
    s & \quad \rightarrow \quad g \quad \rightarrow \quad s' = g(a, s) \\
    \text{post} & \quad s' = g(a, s) \land r = f(a, s')
\end{align*}
\]
Example of *update first* writer

A cash-point operation:

$$DEBIT : (m : Amount) \rightarrow (r : Receipt)$$

$$\text{wr } s : Account$$

$$\text{pre } m \leq balance s$$

$$\text{post } s' = \text{debit } m \ s \land r = balance \ s'$$
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 $A = \{0, 1\}$, $B = A \cup \{B\}$, the state transition diagram will include...
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.

\[
\begin{align*}
\text{Pick} & : \rightarrow (x : \text{Marble}) \\
\text{wr } b & : \text{Bag} \\
\text{pre } b & \neq \{\} \\
\text{post } x & \in b \land b' = b - \{x\}
\end{align*}
\]

So, in general, Mealy machines can be nondeterministic.
Proof obligation

For every

\[ OP : (b : B) \leftarrow (a : A) \]

\[ \text{wr/rd} \ s : St \]

\[ \text{pre} \ ... \]

\[ \text{post} \ ... \]

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

Satisfiability

\[
\forall s, a : \\
\left( s \in St \land a \in A : \right) \\
\text{pre-OP}(s, a) \Rightarrow \exists s' , b : s' \in St \land b \in B : \text{post-OP}(s', b, s, a)
\]
DMMs can be built and animated using the state monad:

- Recall

\[ \delta : A \rightarrow (S \rightarrow (B \times S)) \]

- Every function of type \( ST \ S \ B \) will be referred to as a state transformer

- For a fixed state space \( S \), \( F \ \overset{\text{def}}{=} \ ST \ S \) 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:

\[
\text{update} : (S \rightarrow S) \rightarrow ST \ S \ 1
\]

\[
\text{update } f \triangleq \langle !, f \rangle
\]

Query the state:

\[
\text{query} : (S \rightarrow B) \rightarrow ST \ S \ B
\]

\[
\text{query } f \triangleq \langle f, id \rangle
\]

Return a result:

\[
\text{return} : B \rightarrow ST \ S \ B
\]

\[
\text{return } b \triangleq \langle b, id \rangle
\]
Combining state transformers

Sequential composition:

\[
\text{seq} : \text{ST} \ S \ A \rightarrow \text{ST} \ S \ B \rightarrow \text{ST} \ S \ B \\
\text{seq } f \ g \triangleq \text{do } \{f ; g\}
\]

"Update first" transformer:

\[
\text{updfst} : (A \rightarrow S \rightarrow S) \rightarrow (A \rightarrow S \rightarrow B) \rightarrow A \rightarrow \text{ST} \ S \ B \\
\text{updfst } g \ f \ a \triangleq \text{do } \{\text{update}(g \ a); \text{query}(f \ a)\}
\]

"Query first" transformer:

\[
\text{qryfst} : (A \rightarrow S \rightarrow S) \rightarrow (A \rightarrow S \rightarrow B) \rightarrow A \rightarrow \text{ST} \ S \ B \\
\text{qryfst } g \ f \ a \triangleq \text{do } \{b \leftarrow \text{query}(f \ a); \text{update}(g \ a); \text{return } b\}
\]
Animating state transformers

Running:

\[ \text{run} : ST \ S \ A \rightarrow S \rightarrow (A \times S) \]

\[ \text{run } g \ s \triangleq g \ s \]

Example: given \( POP \triangleq \text{qryfst head tail} \), by running \( POP \) over state \([1, 2, 3]\) one obtains

\[ \text{run } POP \ [1, 2, 3] = (1, [2, 3]) \]

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.*  

A. Cruz, L. Barbosa, and J. Oliveira.  
From algebras to objects: Generation and composition.  