

# A Functional Approach to the Observer Pattern

Álvaro García Pérez

Oxford, May 2009

# Outline

## Introduction

Motivation

The Observer Pattern

## Functional Implementation

Explicit State

State inside a Monad

Alternative Structures

## Case Studies

GUI Using Gtkhs

Reactive Values

## Related Work

Functional Reactive Programming

More Related Work

# Motivation

- ▶ Generic reusable functional solution to propagating changes.
- ▶ Decoupling subject management and updating strategies.
- ▶ Algebraic structure capturing this?
- ▶ Aims to find a new structure, no direct translations from OO.

# The Observer Pattern (I)

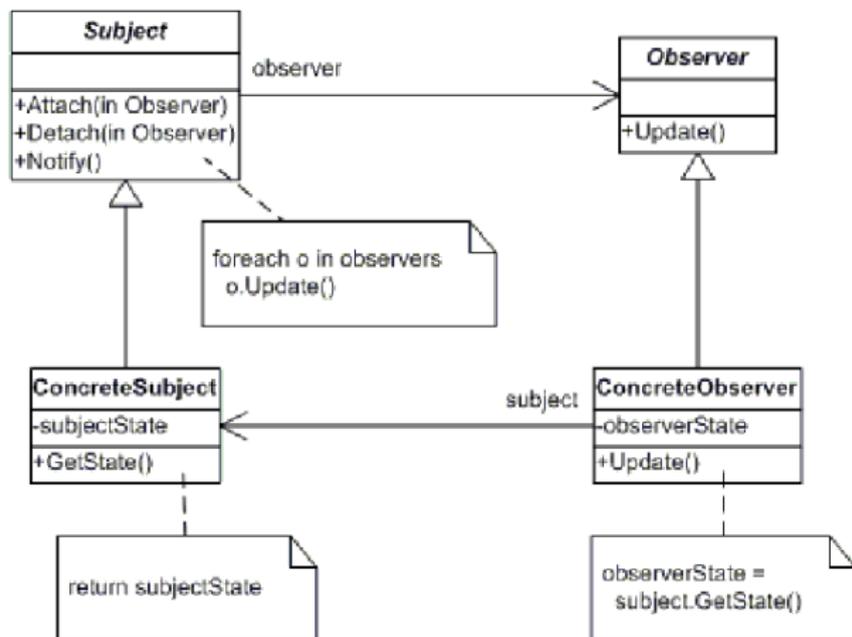


Figure: Observer pattern

# The Observer Pattern (II)

Possible interpretation in Haskell.

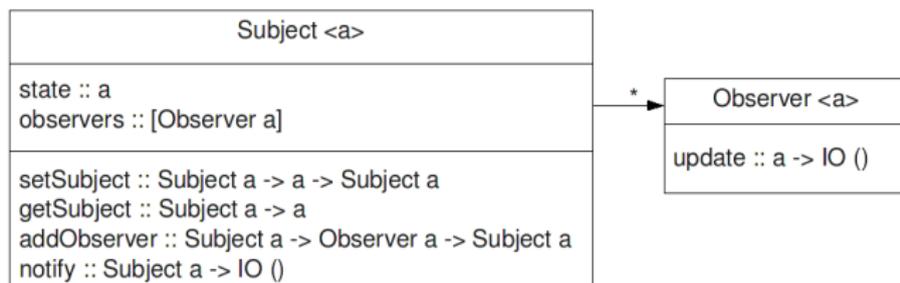


Figure: Observer pattern interpreted

- ▶ Parametrics instead of inheritance.
- ▶ Observers perform effects by IO monad.
- ▶ Passing self reference in subject methods.

## The Observer Pattern (III)

A returned type could be used instead of ().

**type** *Observer*  $m\ a\ b = a \rightarrow m\ b$

*notify* :: *Subject*  $m\ a \rightarrow m\ [b]$

But:

- ▶ Most uses just discard that value.
- ▶ Subject would need to know about that type (less genericity)

## Existing Analogies

$mapfs :: [t \rightarrow u] \rightarrow t \rightarrow [u]$

$mapfs [] x = []$

$mapfs (observer : observers) x = (observer x) : mapfs observers x$

Nedunuri and Cook's [7] functional analog of the Observer pattern.

- ▶ Just describes the notify/updating mechanism.
- ▶ Lacks encapsulation and decoupling details and a reusable framework.

# Implementation with Explicit State

- ▶ State is modelled inside the *Subject*.
- ▶ *Subject* is passed around as the self reference.
- ▶ Using monads to encapsulate effects.
- ▶ The result of any method is now returned into the monad.

## Implementation with Explicit State (cont'd)

**data** *Subject m a* = *S a [Observer m a]*

**type** *Observer m a* = *a → m ()*

*setSubject* :: *Monad m ⇒ Subject m a → a → m (Subject m a)*

*getSubject* :: *Monad m ⇒ Subject m a → m a*

*addObserver* :: *Monad m ⇒ Subject m a → Observer m a →*  
*m (Subject m a)*

*notify* :: *Monad m ⇒ Subject m a → m ()*

*notify* (*S × []*) = *return ()*

*notify* (*S × (o : os)*) = **do** *o x*  
*notify \$ S × os*

## State inside a Monad

- ▶ The subject state and the observers could be passed implicitly inside a monad.
- ▶ Transforming the effectful monad by this one.
- ▶ Using Haskell's *StateT*.

## State inside a Monad (cont'd)

```
type SubjectT a m b = StateT (a, [Observer a m]) m b  
newtype Observer a m = O{getO :: a → SubjectT a m ()}
```

```
setSubject    :: Monad m ⇒ a → SubjectT a m ()  
getSubject    :: Monad m ⇒ SubjectT a m a  
addObserver   :: Monad m ⇒ Observer a m → SubjectT a m ()  
notify        :: Monad m ⇒ SubjectT a m ()  
notify = StateT (λ(a, os) → case os of  
  [] → return ((), (a, os))  
  _  → notifyAux os (a, os))  
notifyAux [] (a, os) = return ((), (a, os))  
notifyAux (o' : os') (a, os) =  
  do init ← notifyAux os' (a, os)  
      runStateT ((getO o') a) $ snd init
```

# Arrows and Applicative Effects

If we forget about the *getSubject* method, which is a common use of the pattern, we can give alternative implementations taking arrows [5] or applicative functors [4] as the effectful world.  
(Future Work)

# GUI Using Gtkhs

We have two windows:

- ▶ One acts as the subject.
- ▶ The other acts as the observer, updating its window title according to the changes on the subject window.

## Gtkhs

- ▶ Graphical objects as data types (*window*, *button*).
- ▶ Hierarchies using type classes.
- ▶ Signals represent events in the outer world (*onClicked*).
- ▶ Handlers can be attached to signals over graphical objects (*onClicked button...*).

## GUI Using Gtkhs (cont'd)



Figure: Before clicking on the subject window button



Figure: After clicking on the subject window button

## Without the Observer Pattern

```
main = do
```

```
initGUI
```

```
windowS ← windowNew
```

```
buttonS ← buttonNew
```

```
onClicked buttonS (set windowS [windowTitle := "New title"])
```

```
windowO ← windowNew
```

```
onClicked buttonS (do title ← get windowS windowTitle  
                        set windowO [windowTitle := title])
```

```
mainGUI
```

# Without the Observer Pattern (cont'd)

## Advantages:

- ▶ Shorter code.

## Drawbacks:

- ▶ Every new updating strategy must have a hard-reference to the subject.
- ▶ Impossible to decouple the code managing the subjects from the code managing observations.

## With the Observer Pattern (Explicit State) (I)

```
main = do
```

```
initGUI
```

```
windowS ← windowNew
```

```
buttonS ← buttonNew
```

```
s ← newSubject windowS
```

```
windowO ← windowNew
```

```
o ← newObserver ( $\lambda w \rightarrow$  do title ← get w windowTitle  
                                set windowO [windowTitle := title])
```

```
s ← addObserver s o
```

```
onClicked buttonS (do set windowS [windowTitle := "New title"]  
                       setSubject s windowS  
                       return ())
```

```
mainGUI
```

## With the Observer Pattern (Explicit State) (II)

### Advantages:

- ▶ Easy to decouple the observing strategy from the subject management.

### Drawbacks:

- ▶ Requires additional machinery.
- ▶ Due to referential transparency the topology of the subject and observers cannot be changed at run time.

## With the Observer Pattern (Explicit State) (III)

Interleaving the effects on the subject with the target effectful world (*IO*) is difficult:

- ▶ We'd need to have mutable subjects (the *Subject* data type, not the inner state) to add new observers inside the handler of some signal.
- ▶ *IORef* gets rid of this problem, but is an impure and Haskell specific solution.

The solution with implicit state actually composes two effectful worlds. What happens if we use it?

## With the Observer Pattern (Implicit State)

```
windowS ← liftIO windowNew  
buttonS ← liftIO buttonNew  
newSubject windowS
```

```
windowO ← liftIO windowNew  
o ← return $ O (\w → do  
  title ← liftIO $ get w windowTitle  
  liftIO $ set windowO [windowTitle := title])
```

```
s ← addObserver o  
liftIO $ onClicked buttonS (do  
  set windowS [windowTitle := "New title"]  
  setSubject s windowS)
```

Cannot give type *SubjectT Window IO ()* to the handler of *onClicked!!*

## With the Observer Pattern (Implicit State) (cont'd)

- ▶ Common graphical libraries are not generic enough to support handlers in *MonadIO*.
- ▶ *IO* is too big, maybe it should be redesigned.

# Reactive Values

Consider sequencing and assignment:

```
b := 3;  
c := 4;  
a := b + c;  
b := 5; # a must be updated automatically
```

- ▶ The values must be updated automatically any time some other depending value is changed.
- ▶ This feature is included in many new scripting languages, like JavaFx.

## Reactive Values (cont'd)

We need to extend the previous solution in order to implement this case study:

- ▶ Working with multiple subjects.
- ▶ Letting to manipulate the subjects inside the observer handlers.
- ▶ Adding an abstract syntax for the possible expressions involving reactive values.

(Future work)

# Functional Reactive Programming (I)

Functional Reactive Programming (FRP) seem to use some mechanisms similar to the observer. They represent time in an explicit way [3, 2].

- ▶ *Time* is a pointed CPO.
- ▶ *Behaviour*  $a$  is semantically interpreted as a function of type  $Time \rightarrow a$ .
- ▶ *Event*  $a$  is semantically interpreted as  $[(Time, a)]$ , ordered by increasing *Time*.

FRP model continuous behaviours (which are later rendered to some discrete time) and instantaneous events (which are detected by finite pooling algorithms).

FRP is supported by massive concurrency, which is achieved by the particular implementations.

## Functional Reactive Programming (II)

Although FRP lets you to implement the same case studies than the pattern, their aims and mechanisms are not comparable.

- ▶ The pattern uses a model of discrete time, which is incremented with every change altering the subject whereas most common implementations of FRP use continuous time.
- ▶ The observers use a pooling strategy for every instant of the time defined above whereas FRP needs to improve this algorithm because of concurrency.
- ▶ The pattern enforces a non concurrent solution whereas FRP uses massive concurrency.

# Functional Reactive Programming (III)

## Advantages:

- ▶ FRP uses an hybrid push/pull programming model with topological order for the updates, which enforces the synchrony hypothesis and avoids glitches.

## Drawbacks:

- ▶ Its push/pull hybrid approach is highly concurrent and could be no worth in small systems.

## More Related Work

- ▶ Dependency structure between spreadsheet cells [1].
- ▶ Interactive applications using arrows [6].



Walter A.C.A.J. de Hoon, Luc M.W.J. Rutten, and Marko C.J.D van Eekelen.

Implementing a functional spreadsheet in clean.  
*Journal of Functional Programming*, 5:383–414, 1995.



Conal Elliott.

Simply efficient functional reactivity.  
Technical Report 2008-01, LambdaPix, April 2008.



Conal Elliott and Paul Hudak.

Functional reactive animation.  
In *International Conference on Functional Programming*, 1997.



Conor McBride and Ross Paterson.

Applicative programming with effects.  
*J. Funct. Program.*, 18(1):1–13, 2008.



Ross Paterson.

Arrows and computation.  
In Jeremy Gibbons and Oege de Moor, editors, *The Fun of Programming*, pages 201–222. Palgrave, 2003.



Marko van Eekelen Peter Achten and Rinus Plasmeijer.

Towards A Unified Semantic Model For Interactive Applicatons  
Using Arrows And Generic Editors.

In *Trends in Functional Programming*, pages 279–292,  
Nottingham, UK, 2006.



William Cook Srinivas Nedunuri.

Transforming declarative models using patterns in mda.

OOPSLA/GPCE: Best Practices for Model-Driven Software  
Development Workshop (MDSD'04), October 2004.