Paradigm Shift

Using Object-Oriented Techniques in Functional Programming

Struggle To Think Functional

Programmers understands the basic of the paradigm like:

- First-Class Function
- Immutability
- Higher-Order Function
- Purity
- Map, Recursion, ...

Struggle To Think Functional

Many problems can be viewed through object-oriented pretty nicely.

Still, many of them struggle to transform it to functional using the basics mentioned before.

It isn’t enough.

How I can express object hierarchy!?

Object-Oriented Principles

- Data and Behavior
- Polymorphism/Dispatch
- Encapsulation
- Inheritance

Object-Oriented Principles

- Data and Behavior
- Polymorphism/Dispatch
- Encapsulation
- Inheritance

Data & Behavior

Data & Behavior

Object-Oriented says an object has set of properties and methods to represent its behavior.

In functional programming, data and behavior can be represented as functions.

No difference between them.

Data & Behavior

$$\begin{tikzcd}
 & \mathbb{R} \\
Rectangle \arrow[ur, bend left, "getWidth"] \arrow[dr, bend right, "getHeight"'] & \\
 & \mathbb{R}
\end{tikzcd}$$

Data & Behavior

Data is just primitive information/properties of a type.

Behavior is build on top of the data.

Data & Behavior

$$\begin{tikzcd}
 & & \mathbb{R} \\
Rectangle \arrow[urr, dotted, bend left, "getWidth"] \arrow[drr, dotted, bend right, "getHeight"'] \arrow[dd, "getRectanglePerimeter"'] & & \\
 & & \mathbb{R} \\
\mathbb{R} & &
\end{tikzcd}$$

Data & Behavior

Data construction is also just a function mapping a product type to the data type.

The function is an isomorphism. It has an inverse and can be called "deconstruction".

As construction and deconstruction are isomorphic, the data type and its corresponding product type can be theoretically used interchangeably.

Data & Behavior

$$\begin{tikzcd}
 & & \mathbb{R} \\
Rectangle \arrow[urr, bend left, "getWidth"] \arrow[drr, bend right, "getHeight"'] \arrow[r, dashed, shift right, "deconstruct"'] & \mathbb{R} \times \mathbb{R} \arrow[l, dashed, shift right, "construct"'] \arrow[ur, bend right, "first"'] \arrow[dr, bend left, "second"] & \\
 & & \mathbb{R}
\end{tikzcd}$$

Data & Behavior

Many functional languages deconstruct product types to lists of variables which hold values of their properties.

It can help eliminating a lot of boilerplate code to explicit deconstruct the data types.

As all data types are constructed from product types and isomorphic to them, so that the data types can theoretically be deconstructed to lists of variables as well.

Polymorphism & Dispatch

Polymorphism

Object-Oriented polymorphism is one type of many polymorphisms that is called “subtype-polymorphism”.

In functional programming, it can be seen as a function to map a subtype to a supertype.

Polymorphism

$$\begin{tikzcd}
X_1 \arrow[dr, dashed, bend left, "\varphi_1"'] \arrow[drr, bend left, "f_1"] & & \\
& X \arrow[r, "f"] & Y \\
X_2 \arrow[ur, dashed, bend right, "\varphi_2"] \arrow[urr, bend right, "f_2"'] & &
\end{tikzcd}$$

Polymorphism

$$\begin{tikzcd}
Square \arrow[dr, dashed, "\varphi_1"'] \arrow[drr, bend left, "getSquarePerimeter"] & & \\
 & Quadrilateral \arrow[r, "getPerimeter"] & \mathbb{R} \\
Rectangle \arrow[ur, dashed, "\varphi_2"] \arrow[urr, bend right, "getRectanglePerimeter"'] & &
\end{tikzcd}$$

Polymorphism

Traditional OO languages require hierarchy to be defined upfront.

Extending interfacing may be not trivial.

And sometimes, causes mismatch in the abstraction.

Polymorphism

$$\begin{tikzcd}
Square \arrow[dr, dotted] \arrow[drrr, dotted, bend left] \arrow[ddr, dashed, "\phi_1"'] \arrow[ddrrr, "drawHalfSquare"] & & & \\
& Quadrilateral \arrow[rr, dotted] & & \mathbb{R} \\
& SymmetricalShape \arrow[rr, "drawHalfImage"] & & Image \\
Rectangle \arrow[uur, dotted] \arrow[uurrr, dotted] \arrow[ur, dashed, "\phi_2"] \arrow[urrr, bend right, "drawHalfRectangle"'] & & &
\end{tikzcd}$$

Polymorphism

Functional programming love extensibility.

We don’t like to define interfacing upfront.

Polymorphism (Coercion)

If the concrete functions have similar implementation, they can usually be factored through a general data type.

We can just “coerce” the domain of the concrete functions to that general type.

Polymorphism (Coercion)

$$\begin{tikzcd}
Square \arrow[dr, bend left, "\varphi_1"'] \arrow[drr, dashed, bend left, "getSquarePerimeter"] & & \\
& Quadrilateral \arrow[r, "getPerimeter"] & \mathbb{R} \\
Rectangle \arrow[ur, bend right, "\varphi_2"] \arrow[urr, dashed, bend right, "getRectanglePerimeter"'] & &
\end{tikzcd}$$

Polymorphism (Coercion)

Sometimes, the general type construction is not trivial.

If the general function can be further factored through another function which maps the type to another simple structure,

We can bypass the tedious construction by construct the simple one instead!!!

Polymorphism (Coercion)

$$\begin{tikzcd}
Square \arrow[dr, bend left, "\varphi_1"'] \arrow[drr, bend left, "\phi_1 = \phi \circ \varphi_1"] \arrow[drrr, dashed, bend left, "getSquarePerimeter"] & & & \\
& Quadrilateral \arrow[r, "\phi"] \arrow[rr, dashed, bend right, "getPerimeter"'] & \mathbb{R} \times \mathbb{R} \times \mathbb{R} \times \mathbb{R} \arrow[r, "sumLengthOfAllEdges"] & \mathbb{R} \\
Rectangle \arrow[ur, bend right, "\varphi_2"] \arrow[urr, bend right, "\phi_2 = \phi \circ \varphi_2"'] \arrow[urrr, dashed, bend right, "getRectanglePerimeter"'] & & &
\end{tikzcd}$$

Polymorphism (Coercion)

When combined with the fact that functional programming prefers simple structures and they are isomorphic to product types, many generalization can be solved using coercion.

That is why product types matter in functional programming as it allows many generalization to be easily implemented/extended in any level of codes.

Polymorphism (Dispatch)

Not all generalization can be easily factored through coercion to a product type.

In this case, we prefer to solve it by dispatch.

Object-Oriented implement it with subtyping.

Polymorphism (Dispatch)

$$\begin{tikzcd}
MacBook \arrow[dr, dashed, bend left, "\varphi_1"'] \arrow[drr, bend left, "getMacBookBasePrice"] & & \\
& Merchandise \arrow[r, dashed, "getBasePrice"] & \mathbb{R} \\
Kindle \arrow[ur, dashed, bend right, "\varphi_2"] \arrow[urr, bend right, "getKindleBasePrice"'] & &
\end{tikzcd}$$

Polymorphism (Dispatch)

$$\begin{tikzcd}
MacBook \arrow[dr, dashed, "\varphi_1"'] \arrow[drr, bend left, "getMacBookUplift"] & & \\
& Merchandise \arrow[r, dashed, "getUplift"] & \mathbb{R} \\
Kindle \arrow[ur, dashed, "\varphi_2"] \arrow[urr, bend right, "getKindleUplift"'] & &
\end{tikzcd}$$

Polymorphism (Dispatch)

To factor “getBasePrice” and “getUplift” through another type may not be easy as the concrete implementations may be too different and difficult to unify.

Polymorphism (Dispatch)

In OO, dispatch process is done at runtime to ensure that, for this instance, calling “getBasePrice” or “getUplift” will invoke the right concrete method.

In functional programming, we love dispatch that can be run in compile-time.

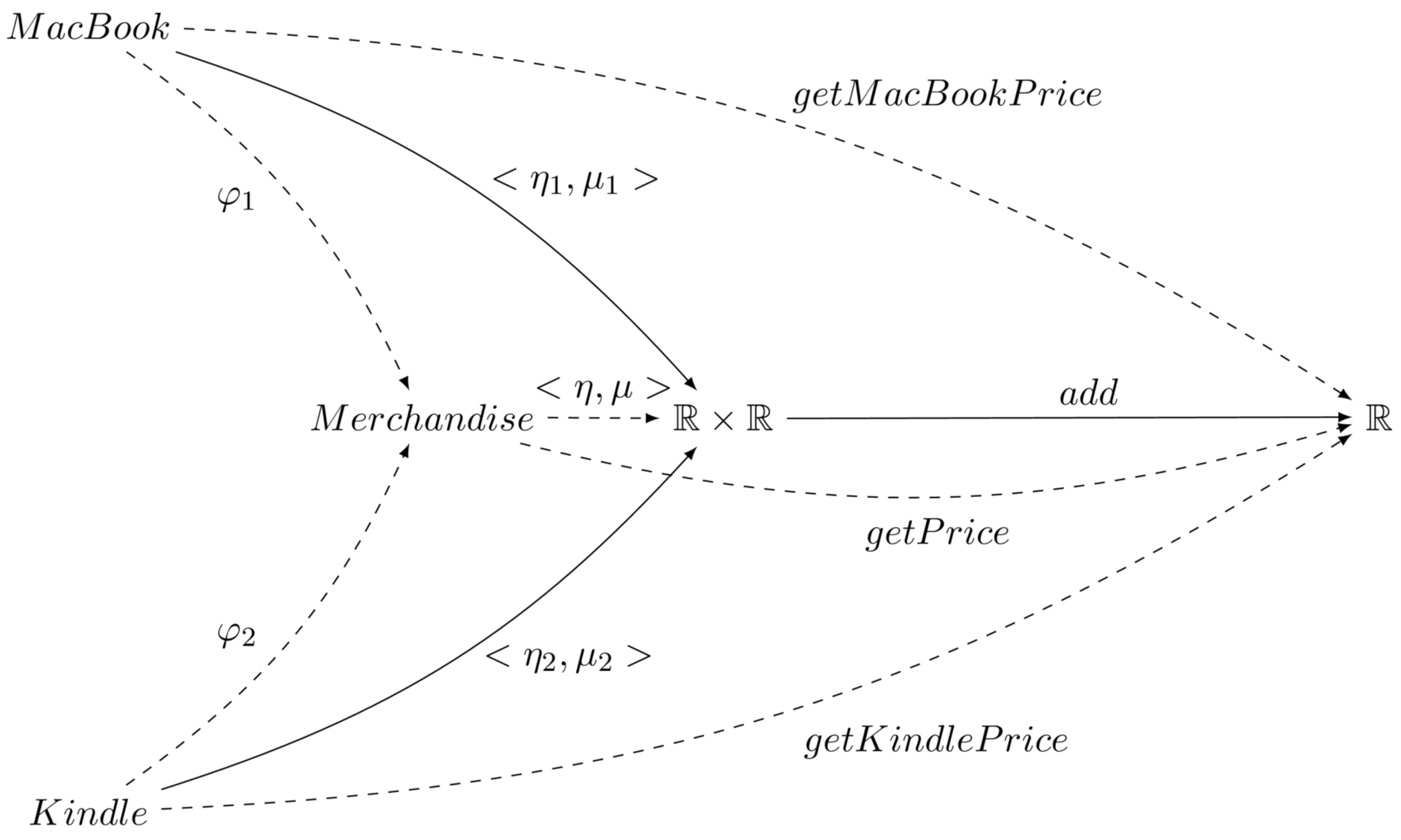
Polymorphism (Coercion + Dispatch)

Calculate a full price by combining base price and uplift together.

Polymorphism (Coercion + Dispatch)

$$\begin{tikzcd}
MacBook \arrow[dr, dashed, "\varphi_1"'] \arrow[drr, bend left, "getMacBookPrice"] & & \\
& Merchandise \arrow[r, "getPrice"] & \mathbb{R} \\
Kindle \arrow[ur, dashed, "\varphi_2"] \arrow[urr, bend right, "getKindlePrice"'] & &
\end{tikzcd}$$

Polymorphism (Coercion + Dispatch)



$$\begin{tikzcd}
MacBook \arrow[dr, dashed, bend left, "\varphi_1"'] \arrow[drr, bend left, "<\eta_1,\mu_1>"] \arrow[drrr, dashed, bend left, "getMacBookPrice"] & & & \\
& Merchandise \arrow[r, dashed, "<\eta,\mu>"] \arrow[rr, dashed, bend right, "getPrice"'] & \mathbb{R}\times\mathbb{R} \arrow[r, "add"] & \mathbb{R} \\
Kindle \arrow[ur, dashed, bend right, "\varphi_2"] \arrow[urr, bend right, "<\eta_2,\mu_2>"'] \arrow[urrr, dashed, bend right, "getKindlePrice"'] & & &
\end{tikzcd}$$

where $\eta = getBasePrice$ and $\mu = getUplift$

Polymorphism (Coercion + Dispatch)

Or work with parametric polymorphism (generic) to dispatch at compile time or at least check if the dispatching is possible.

$$\begin{tikzcd}
MacBook \arrow[dr, dashed, "\varphi_1"] \arrow[drr, bend left, "< \eta_1, \mu_1 >"] & & \\
\forall x . x_{has\ <\eta,\mu>} \arrow[u, "x = Macbook"'] \arrow[d, "x = Kindle"'] & Merchandise \arrow[r, dashed, "< \eta, \mu >"] & \mathbb{R} \times \mathbb{R} \\
Kindle \arrow[ur, dashed, "\varphi_2"] \arrow[urr, bend right, "< \eta_2, \mu_2 >"'] & &
\end{tikzcd}$$

where $\eta = getBasePrice$ and $\mu = getUplift$