15-312 Lecture on Dynamic Dispatch

Object-oriented programming is a computing model by which a multitude of *objects* interact with each other. An object consists of *fields* containing data and *methods* that permit manipulating the object's fields and interacting with other objects. Computation in an object-oriented system happens by invoking the methods of an object, which in turn may invoke the methods of other objects, and so on. This organization, by which an object contains its own internal state (its fields) and the functionalities relevant to this state (its methods), is called *encapsulation* and is one of the characteristics of object-oriented programming.

Objects that differ at most by the value of their fields (their state) but not by their methods belong to the same *class*. In a sense, a class is the "mold" of an object: it defines its methods and the type of its fields — a class is a hybrid of types (of the fields) and values (of the methods). Each object belonging to a class is an *instance* of this class: it gets the methods from the class and has its values associated to its fields. Classes are another characteristic of object-oriented programming.

A third characteristic is *inheritance*. Inheritance is an approach to defining new classes on the basis of existing classes. In the simplest case (single inheritance), the new class (the *subclass*) differs from the existing class (the *superclass*) by extending it with new fields and methods, or by redefining (*overriding*) some of its methods. A consequence of inheritance is that an object can belong to multiple classes: the class for which it was defined and also all of its superclasses. This object is viewed as an instance of the superclass by ignoring the added fields and methods, and by reverting overridden methods to those of the superclass.

In this handout, we will examine the class mechanism deprived of some orthogonal aspects of the above characteristics. In particular, we will consider a setup consisting of multiple classes, each with the same methods (but possibly very different fields). An object can therefore be an instance of several classes, but each defines the exact same methods. Notions that are not considered include the hierarchical aspect of inheritance (which is dealt with through subtyping — see PFPL Ch. 23 and 26) and the self-referential nature of objects (which is modeled by means of recursive types — see PFPL Ch.16).

Classes and Methods

The best way to get going is to forget about objects and instead consider a setup where we want to describe a high-level entity with some well-defined operations, but with multiple low-level representations for these entities. The common operations will be our methods. The multiple representations will be our classes (or more precisely the field portion of our classes).

Borrowing from PFPL Ch. 22, we will use points on the plane as our on-going example of a "high-level entity". The operations of interest (our methods) will be the distance of a point from the origin (Distance) and the quadrant where a point is located (Quadrant) — of course we could define many more interesting operations but this shall suffice for our purposes. The underlying representations (our classes) will be the Cartesian and polar coordinate systems (again this will be enough): the fields of the Cartesian class will be the x and y coordinates of the point, both of type real. By contrast, the fields of the Polar class will be a real for the radial distance from the origin and an angle for the angle relative to the positive half of x-axis (we assume that an angle is a floating point number with values between 0 and 2π, together with appropriate operations).

In this setup, a user may choose to work with either Cartesian or polar coordinates, and in each case be given access to the Distance and Quadrant methods. Our job is therefore to provide an implementation of each method for each class: that's a total of four functions. It is natural to arrange these four functions in tabular form, where the rows correspond to the possible representations (the classes) and the columns with the various methods. We can then display it as follows:

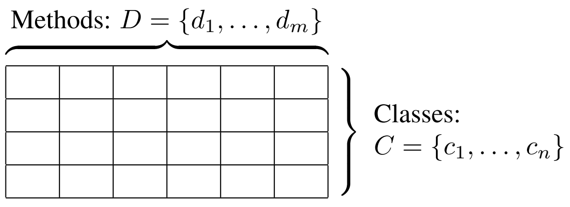
		Distance:	Quadrant:

Cartesian:	$\langle x, y\rangle$	$\sqrt{x^2+y^2}$	if $x \geq 0$ & $y \geq 0$ then I
			else if $x \geq 0$ & $y < 0$ then II
			else if $x < 0$ & $y < 0$ then III
			else if $x < 0$ & $y \geq 0$ then IV

Polar:	$\langle r, \theta\rangle$	r	if $0 \leq \theta < \pi/2$ then I
			else if $\pi/2 \leq \theta < \pi$ then II
			else if $\pi \leq \theta < 3\pi/2$ then III
			else if $3\pi/2 \leq \theta < 2\pi$ then IV

where a quadrant is given by the enumeration type quad $\triangleq$ {I, II, III, IV}.

This table is called the *dispatch matrix*. In general, given a fixed set of classes $C = \{c_1, \ldots, c_n\}$ and a fixed set of methods $D = \{d_1, \ldots, d_m\}$, the dispatch matrix has the following form:


Methods: $D = \{d_1, \ldots, d_m\}$
Classes: $C = \{c_1, \ldots, c_n\}$

One row for each class and one column for each method. At the intersection of row c and column d, we have a function that implements method d for entities obeying the representation defined by class c — call its body e^c_d.

Because each row implements the methods for the same class, every function on this row takes as input the constituents of this class's representation: two real for the Cartesian class and one real and one angle for the Polar class. Dually, because every column provides the same functionality based on the different underlying representations, the methods along this column will all return values of the same type: a real for Distance and a quad for Quadrant. It is again convenient to organize the types of the functions in our dispatch matrix in tabular form:

$$\begin{array}{rr|c|c|}
 & & \underline{\text{Distance}}\text{:} & \underline{\text{Quadrant}}\text{:} \\
\cline{3-4}
\underline{\text{Cartesian}}\text{:} & \mathsf{real}\times\mathsf{real}\to & \mathsf{real}\times\mathsf{real}\to\mathsf{real} & \mathsf{real}\times\mathsf{real}\to\mathsf{quad} \\
\cline{3-4}
\underline{\text{Polar}}\text{:} & \mathsf{real}\times\mathsf{angle}\to & \mathsf{real}\times\mathsf{angle}\to\mathsf{real} & \mathsf{real}\times\mathsf{angle}\to\mathsf{quad} \\
\cline{3-4}
\multicolumn{1}{r}{} & \multicolumn{1}{r}{} & \multicolumn{1}{c}{\downarrow} & \multicolumn{1}{c}{\downarrow} \\
\multicolumn{1}{r}{} & \multicolumn{1}{r}{} & \multicolumn{1}{c}{\mathsf{real}} & \multicolumn{1}{c}{\mathsf{quad}}
\end{array}$$

This highlights the fact that all functions on the same row of the dispatch matrix have the same input type and all functions on the same column have the same output type. Abstractly, let τ^c be the common representation type of class c (a row in the dispatch matrix) and ρ_d be the return type of method d (a column in the matrix), then the cell where this row and this column intersect has type $\tau^c \to \rho_d$ and the function in this cell will have the form $\lambda x : \tau^c e^c_d$ where the body e^c_d has type ρ_d assuming that the input x has type τ^c. This is described schematically as follows for types and values:

$$\begin{array}{r c|c|c}
 & & \vdots & \\
\cline{2-4}
\tau^c \to & \cdots & \tau^c \to \rho_d & \cdots \\
\cline{2-4}
 & & \vdots & \\
\multicolumn{1}{r}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{\downarrow} & \multicolumn{1}{c}{} \\
\multicolumn{1}{r}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{\rho_d} & \multicolumn{1}{c}{}
\end{array}
\qquad\qquad
\begin{array}{r c|c|c}
 & & \vdots & \\
\cline{2-4}
x : \tau^c & \cdots & \lambda x : \tau^c .\, e^c_d & \cdots \\
\cline{2-4}
 & & \vdots & \\
\multicolumn{1}{r}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{: \rho_d} & \multicolumn{1}{c}{}
\end{array}$$

Types *Values*

As our example shows, the type τ^c associated with a class $c \in C$ is typically a product. Its values are records of values for each of the fields of C.

The Dispatch Matrix

So far, we have viewed the dispatch matrix as a convenient way to visualize the functions implementing the methods for each representation class, as well as their type. Next, we will be interested in its own status in a programming language.

In a way, the dispatch matrix of our example is just four functions: it is therefore convenient to view it as a tuple with four elements:

$$\begin{array}{ll} \langle & \lambda\langle x,y\rangle\ \sqrt{x^2+y^2}, \\ & \lambda\langle x,y\rangle\ \textsf{if}\ x \geq 0\ \&\ y \geq 0\ \textsf{then} \ldots, \\ & \lambda\langle r,\theta\rangle\ r. \\ & \lambda\langle r,\theta\rangle\ \textsf{if}\ 0 \leq \theta < \pi/2\ \textsf{then} \ldots \qquad \rangle \end{array}$$

which naturally has the following product type:

$$\begin{array}{ll} & (\textsf{real} \times \textsf{real} \to \textsf{real}) \\ \times & (\textsf{real} \times \textsf{real} \to \textsf{quad}) \\ \times & (\textsf{real} \times \textsf{angle} \to \textsf{real}) \\ \times & (\textsf{real} \times \textsf{angle} \to \textsf{quad}) \end{array}$$

Observe that there is nothing tabular about this type: it is just the product of four function types.

Abstractly, we can define a generic dispatch matrix as a tuple e^C_D consisting of all the implementations $\lambda x : \tau^c.\, e^c_d$ for all classes $c \in C$ and methods $d \in D$:

$$e^C_D \quad \triangleq \quad \langle\langle \lambda x : \tau^c.\, e^c_d \rangle_{d \in D}\rangle_{c \in C}$$

The type τ^C_D of this dispatch matrix has therefore the form

$$\tau^C_D \quad \triangleq \quad \prod_{c \in C} \prod_{d \in D} (\tau^c \to \rho_d)$$

These expressions view the matrix as tuple of tuples (the columns). The tabular form is back, although tenuously.

It is easy to show that

$$e^C_D \cdot d \cdot c \quad \mapsto^* \quad \lambda y : \tau^c.\, e^c_d$$

Organizations

Our next goal will be to design an abstraction of the dispatch matrix that supports a natural notion of *object* as well as two basic operations on them: creating an object and invoking a method on it. The dispatch matrix as we presented it so far does not lend itself to such an interpretation: it is just a tuple of functions.

Given a dispatch matrix e^C_D of type τ^C_D, we are therefore interested in defining the following three entities:

- A type $\textsf{obj}$ that captures our understanding of what an object is.
- An operation $\textsf{new}[c](e)$ that, given the an expression e that evaluates to the fields of class c, returns an object. Viewed as a function of e, this operation has type $\textsf{new}[c](\_) : \tau^c \to \textsf{obj}$.

- An operation $e \Leftarrow d$ that returns the result of invoking method d on the object e. Viewed as function of e, this operation has type $\_ \Leftarrow d : \mathsf{obj} \to \rho_d$.

Naturally, if the dispatch matrix contains the function $\lambda y : \tau^c.\, e^c_d$ at the intersection of row c and column d, we want $(\mathsf{new}[c](e)) \Leftarrow d$ to behave exactly as $[e/y]e^c_d$. We will actually be able to achieve a stronger result, namely

$$(\mathsf{new}[c](e)) \Leftarrow d \quad \mapsto^* \quad e^c_d\; e$$

We will obtain two natural organizations along these lines by prioritizing either the rows or the columns of the dispatch matrix. But first, a more general concept.

Type Isomorphisms

Intuitively, two types τ_1 and τ_2 are *isomorphic*, written $\tau_1 \cong \tau_2$, if, whenever a value of one type is used, we could extract the exact same information by using an appropriate value of the other type instead. This implies in particular that there exist two expressions[1]

- $e_{12} : \tau_1 \to \tau_2$, and
- $e_{21} : \tau_2 \to \tau_1$

such that

- $e_{21}\ (e_{12}\ v_1) \mapsto^* v_1$ for every value $v_1 : \tau_1$, and
- $e_{12}\ (e_{21}\ v_2) \mapsto^* v_2$ for every value $v_2 : \tau_2$.

These functions are called the *witnesses* of the type isomorphism. They are bijections and each other's inverse. The function e_{12} is a transformation that permits using an expression of type τ_1 wherever an expression of type τ_2 is expected, and dually for e_{21}. Moreover, given any value $v_1 : \tau_1$, the value $v_2 : \tau_2$ of the expression $e_{12}\ v_1$ contains the exact same information as v_1, and it could be recovered by applying e_{21} to it. The function e_{21} has a similar property.

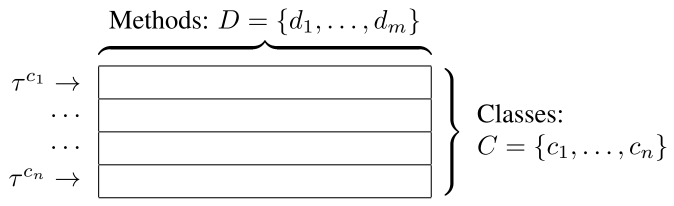
As an example, $\tau \times \tau' \cong \tau' \times \tau$ are isomorphic for any types τ and τ'.

Class-Based Organization

The *class-based* organization of the dispatch matrix exploits the observation that every function on the same row takes the same input. This common input can then be factored out by means of a type isomorphism, leaving a structure that is prominently organized

[1]The general definition of type isomorphism is discussed in depth in Ch. 47 and 48 of PFPL.

along the rows of the dispatch matrix. This yields the following picture:


Methods: $D = \{d_1, \ldots, d_m\}$
$\tau^{c_1} \rightarrow$
$\cdots$
$\cdots$
$\tau^{c_n} \rightarrow$
Classes: $C = \{c_1, \ldots, c_n\}$

Since each row corresponds to a class $c \in C$, the class-based organization associates with each value $e : \tau^c$ all the methods on that row. Because the user can invoke any of these methods on e, it is natural to collect them into a tuple. The following type isomorphism justifies this intuition.

$$\prod_{c \in C} \prod_{d \in D} (\tau^c \rightarrow \rho_d) \quad \cong \quad \prod_{c \in C} \tau^c \rightarrow \left(\prod_{d \in D} \rho_d \right)$$

Its witnesses are displayed in Figure 1.

Given a dispatch matrix expression e^C_D, we denote with $e^{\boxed{C}}_D$ the corresponding dispatch matrix organized by classes. It is the value obtained by applying the left witness $e^{C \rightarrow \boxed{C}}$ in Figure 1 to e^C_D:

$$\begin{array}{ll}
 & \overbrace{(\lambda m : \tau^C_D.\, \langle \lambda x^c : \tau^c.\, \langle (m \cdot c \cdot d)\ x^c \rangle_{d \in D} \rangle_{c \in C})}^{e^{C \rightarrow \boxed{C}}}\ e^C_D \\
\mapsto & \langle \lambda x^c : \tau^c.\, \langle (e^C_D \cdot c \cdot d)\ x^c \rangle_{d \in D} \rangle_{c \in C} \\
= & \langle \lambda x^c : \tau^c.\, \langle ((\overbrace{\langle \langle \lambda x : \tau^c.\, e^c_d \rangle_{d \in D} \rangle_{c \in C}}^{e^C_D}) \cdot c \cdot d)\ x^c \rangle_{d \in D} \rangle_{c \in C} \\
\mapsto^* & \langle \lambda x^c : \tau^c.\, \langle (\lambda x : \tau^c.\, e^c_d)\ x^c \rangle_{d \in D} \rangle_{c \in C} \\
\mapsto & \langle \lambda x^c : \tau^c.\, \langle e^c_d \rangle_{d \in D} \rangle_{c \in C} \\
\triangleq & e^{\boxed{C}}_D
\end{array}$$

where the third line stems from our earlier definition of e^C_D as $\langle \langle \lambda x : \tau^c.\, e^c_d \rangle_{d \in D} \rangle_{c \in C}$.

In the class-based organization, an object of class c is taken to be simply the collection of all the behaviors induced by the methods on the corresponding row. Therefore, we can define

$$\mathsf{obj} \quad \triangleq \quad \prod_{d \in D} \rho_d$$

Then, given an expression $e^c : \tau^c$ that computes to the fields of class c (and therefore that can be supplied to every method on that row), the object creation operation is defined as

$$\mathsf{new}[c](e^c) \quad \triangleq \quad (e^{\boxed{C}}_D \cdot c)\ e^c$$

Here, $e^{\boxed{C}}_D \cdot c$ picks the row of $e^{\boxed{C}}_D$ corresponding to class c, which is a function that expects an argument of type τ^c. Since e^c is such an argument, applying this function

$$\begin{tikzcd}[row sep=huge]
\tau_D^C \triangleq \prod_{c\in C}\prod_{d\in D}(\tau^c \to \rho_d) \arrow[d, shift right, "\begin{array}{l} e^{C\to\boxed{C}} \triangleq \\ \lambda m : \tau_D^C. \\ \quad \langle \lambda x^c : \tau^c. \\ \qquad \langle (m \cdot c \cdot d)\ x^c\rangle_{d\in D} \\ \quad \rangle_{c\in C} \end{array}"'] \\
\tau_D^{\boxed{C}} \triangleq \prod_{c\in C}\tau^c \to \underbrace{\left(\prod_{d\in D}\rho_d\right)}_{\mathsf{obj}} \arrow[u, shift right, "\begin{array}{l} e^{\boxed{C}\to C} \triangleq \\ \lambda m' : \tau_D^{\boxed{C}}. \\ \quad \langle\ \langle \lambda y^c : \tau^c. ((m' \cdot c)\ y^c) \cdot d\rangle_{d\in D} \\ \quad \rangle_{c\in C} \end{array}"']
\end{tikzcd}$$

Figure 1: Transformation into the Class-Based Organization

to it yields a result for each method in D. This result is an expression, say $e^{\boxed{O}}$, of type obj.

Then, to invoke a method d on such an object, it suffices to select its d-th component. Therefore, the invocation of d on object $e^{\boxed{O}}$ is defined by simply projecting $e^{\boxed{O}}$ along d:

$$e^{\boxed{O}} \Leftarrow d \quad\triangleq\quad e^{\boxed{O}} \cdot d$$

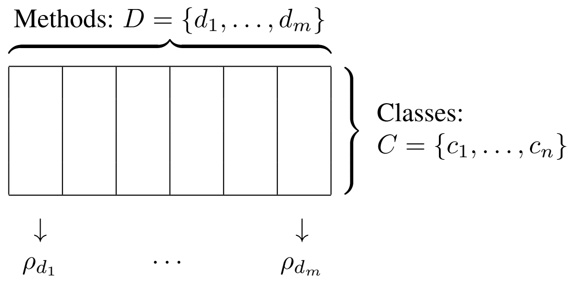
The definitions of both $\mathsf{new}[c](\_)$ and $\_ \Leftarrow d$ can be read off the right witness $e^{\boxed{C}\to C}$ in Figure 1 after replacing the variable m' with $e_D^{\boxed{C}}$:

$$\langle\langle \lambda y^c : \tau^c.\ \overbrace{\underbrace{((e_d^{\boxed{C}} \cdot c)\ y^c)}_{\mathsf{new}[c](y^c)} \cdot d}^{e^{\boxed{O}} \Leftarrow d}\ \rangle_{d\in D}\rangle_{c\in C}$$

where $e^{\boxed{O}}$ is the result of evaluating $\mathsf{new}[c](e^c)$ for whatever expression e^c is supplied as y^c.

Method-Based Organization

The *method-based* organization takes a stance that is dual to that of the class-based organization. It exploits that fact that the functions along the columns of the dispatch matrix have the same type and leverages another type isomorphism to factor out this common output type, which leads to a structure that is prominently organized along the columns of the dispatch matrix. The pictorial interpretation now takes the following form:


Methods: $D = \{d_1, \ldots, d_m\}$
Classes: $C = \{c_1, \ldots, c_n\}$
ρ_{d_1}
$\cdots$
ρ_{d_m}

Given that each column in the dispatch matrix corresponds to a method $d \in D$, invoking d on some entity amounts to selecting the right function along this column on the basis of the representation that was chosen for this entity — its class. Therefore, we shall apply one of several possible functions, depending on the class. Such a choice calls for a sum labeled with the classes in C. The following type isomorphism fully develops this intuition:

$$\prod_{c\in C}\prod_{d\in D}(\tau^c \to \rho_d) \quad\cong\quad \prod_{d\in D}\left(\sum_{c\in C}\tau^c\right) \to \rho_d$$

The witnesses of this type isomorphism are displayed in Figure 2.

Given a dispatch matrix e^C_D, we denote with $e^C_{\boxed{D}}$ the corresponding dispatch matrix organized by methods. It is the value of applying the left $e_{D\to\boxed{D}}$ witness in Figure 2 to e^C_D:

$$\begin{array}{cl}
& \overbrace{(\lambda m : \tau^C_D.\, \langle \lambda x_D : \sum_{c\in C}\tau^c.\, \mathsf{case}\ x_D\{c \cdot x^c_d \Rightarrow (m \cdot c \cdot d)\ x^c_d\}\rangle_{d\in D})}^{e_{D\to\boxed{D}}}\ e^C_D \\
\mapsto & \langle \lambda x_D : \sum_{c\in C}\tau^c.\, \mathsf{case}\ x_D\{c \cdot x^c_d \Rightarrow (e^C_D \cdot c \cdot d)\ x^c_d\}\rangle_{d\in D} \\
= & \langle \lambda x_D : \sum_{c\in C}\tau^c.\, \mathsf{case}\ x_D\{c \cdot x^c_d \Rightarrow ((\overbrace{\langle\langle \lambda x : \tau^c.\, e^c_d\rangle_{d\in D}\rangle_{c\in C}}^{e^C_D}) \cdot c \cdot d)\ x^c_d\}\rangle_{d\in D} \\
\mapsto & \langle \lambda x_D : \sum_{c\in C}\tau^c.\, \mathsf{case}\ x_D\{c \cdot x^c_d \Rightarrow (\lambda x : \tau^c.\, e^c_d)\ x^c_d\}\rangle_{d\in D} \\
\mapsto & \langle \lambda x_D : \sum_{c\in C}\tau^c.\, \mathsf{case}\ x_D\{c \cdot x^c_d \Rightarrow e^c_d\}\rangle_{d\in D}
\end{array}$$

In the method-based organization, an *object* for method d is a choice of all the possible representations that d can be invoked on. This is the sum of the representation types corresponding to all the classes:

$$\mathsf{obj} \quad\triangleq\quad \sum_{c\in C}\tau^c$$

Given an expression $e^c : \tau^c$ that computes to the fields of class c, we create such an object by simply injecting it into the appropriate component of the above sum type:

$$\mathsf{new}[c](e) \quad\triangleq\quad c \cdot e$$

$$\begin{tikzcd}
\tau^C_D \triangleq \prod_{c\in C}\prod_{d\in D}(\tau^c \to \rho_d) \arrow[d, shift right=2, "\begin{array}{l} e_{D\to\boxed{D}} \triangleq \\ \lambda m : \tau^C_D. \\ \quad \langle \lambda x_D : \sum_{c\in C}\tau^c. \\ \qquad \mathsf{case}\ x_D \\ \qquad\quad \{c\cdot x^c_d \Rightarrow (m\cdot c\cdot d)\ x^c_d\} \\ \quad \rangle_{d\in D} \end{array}"'] \\
\tau^C_{\boxed{D}} \triangleq \prod_{d\in D}\underbrace{\left(\sum_{c\in C}\tau^c\right)}_{\mathsf{obj}} \to \rho_d \arrow[u, shift right=2, "\begin{array}{l} e_{\boxed{D}\to D} \triangleq \\ \lambda m' : \tau^C_{\boxed{D}}. \\ \quad \langle\ \langle \lambda y^c : \tau^c.\,(m'\cdot d)\ (c\cdot y^c)\rangle_{c\in C} \\ \quad \rangle_{d\in D} \end{array}"']
\end{tikzcd}$$

Figure 2: Transformation into the Method-Based Organization

Invoking a method d on such an object $e_{\boxed{o}}$ is achieved by simply selecting the d-th column of $e^C_{\boxed{D}}$, i.e., $e^C_{\boxed{D}} \cdot d$, and calling it on this object:

$$e_{\boxed{o}} \Leftarrow d \quad \triangleq \quad (e^C_{\boxed{D}} \cdot d)\ e_{\boxed{o}}$$

Note again that the definitions of $\mathsf{new}[c](\_)$ and $\_ \Leftarrow d$ can be read off the right witness in Figure 2 after replacing the variable m' with $e^C_{\boxed{D}}$:

$$\langle\langle \lambda y^c : \tau^c.\ \overbrace{(m'\cdot d)\ \underbrace{(c\cdot y^c)}_{\mathsf{new}[c](y^c)}}^{e_{\boxed{o}}\Leftarrow d}\ \rangle_{c\in C}\rangle_{d\in D}$$

where $e_{\boxed{o}}$ is the result of evaluating $\mathsf{new}[c](e^c)$ for the expression e^c supplied as y^c.