PFPL Supplement: Dynamic Dispatch as an Abstract Type*

Robert Harper

Fall, 2019

Dynamic dispatch may be seen as an abstract type of objects supporting two methods, creation of an object of a class, and sending a message to an object to obtain a result. The textbook describes an example with two classes of complex number, `cart` and `pol`, and two methods, `dist` and `quad`.

A dynamic dispatch scenario with classes C and methods D specified by instance types τ^c for each $c \in C$, and result types ρ_d for each $d \in D$, may be organized as implementation of the following existential type:

$$\tau_{\mathtt{dd}} \triangleq \exists(\, t_{\mathsf{obj}}.\langle \mathtt{new} \hookrightarrow \langle \tau^c \rightharpoonup t_{\mathsf{obj}} \rangle_{c \in C}, \mathtt{snd} \hookrightarrow \langle t_{\mathsf{obj}} \rightharpoonup \rho_d \rangle_{d \in D} \rangle\,). \tag{1}$$

Given a package, call it x, of this type, a client may open x to gain access to the object creation and message send operations as follows:

$$\mathtt{open}\, x\, \mathtt{as}\, t_{\mathsf{obj}}\, \mathtt{with}\, \langle \mathtt{new} \hookrightarrow new, \mathtt{snd} \hookrightarrow snd \rangle\, \mathtt{in}\, e_{client} \tag{2}$$

Within with expression e_{client} define

$$\mathtt{new}\langle c \rangle(\, e^c\,) \triangleq new \cdot c(\, e^c\,) \qquad\qquad \mathtt{snd}\langle d \rangle(\, e\,) \triangleq snd \cdot d(\, e\,)$$

wherein $e^c : \tau^c$ is instance data appropriate to class c and $e : t_{\mathsf{obj}}$. A new instance of class c is created by the `new` operation applied to an argument of type τ^c. It creates an abstract object of type t_{obj} to which one may send a message d to obtain a result of type ρ_d. Importantly, the types τ^c and ρ_d are independent of the abstract type t_{obj}; their values are meaningful outside of the object abstraction. Moreover, the statics of the `open` expression ensures that the type of the client also be meaningful apart from t_{obj} so as to ensure that abstraction is not violated.

For example, in the case of the abstract type of complex numbers, write

$$\begin{aligned} z &\triangleq \mathtt{new}\langle \mathtt{cart} \rangle(\, \langle \mathtt{x} \hookrightarrow x, \mathtt{y} \hookrightarrow y \rangle\,) \\ &= new \cdot \mathtt{cart}(\, \langle \mathtt{x} \hookrightarrow x, \mathtt{y} \hookrightarrow y \rangle\,) \end{aligned}$$

to create a complex number z with rectangular coordinates (x, y). Then write

$$\begin{aligned} u &\triangleq \mathtt{snd}\langle \mathtt{dist} \rangle(\, z\,) \\ &= snd \cdot \mathtt{dist}(\, z\,) \end{aligned}$$

to compute the squared distance of z from the origin, namely $x^2 + y^2$.

Two natural implementations of the type $\tau_{\mathtt{dd}}$ arise, one by taking an object to be a tuple of methods, one for each method $d \in D$, and one taking an object to be an instance datum labeled with some class $c \in C$. In the former case creating an object requires some work, but sending a message is simply a projection. In the latter case creating an object is simply an injection, but sending a message requires a case analysis. More precisely, two packages of type $\tau_{\mathtt{dd}}$ are given by

$$\mathtt{pack}\, \tau^{\mathrm{I}}_{\mathtt{obj}}\, \mathtt{with}\, \langle \mathtt{new} \hookrightarrow e^{\mathrm{I}}_{\mathtt{new}}, \mathtt{snd} \hookrightarrow e^{\mathrm{I}}_{\mathtt{snd}} \rangle\, \mathtt{as}\, \tau_{\mathtt{dd}}$$

and

$$\mathtt{pack}\, \tau^{\mathrm{II}}_{\mathtt{obj}}\, \mathtt{with}\, \langle \mathtt{new} \hookrightarrow e^{\mathrm{II}}_{\mathtt{new}}, \mathtt{snd} \hookrightarrow e^{\mathrm{II}}_{\mathtt{snd}} \rangle\, \mathtt{as}\, \tau_{\mathtt{dd}}$$

whose components are defined as follows:

$$\begin{aligned}
\tau^{\mathrm{I}}_{\mathtt{obj}} &\triangleq \langle d \hookrightarrow \rho_d \rangle_{d \in D} \\
e^{\mathrm{I}}_{\mathtt{new}} &\triangleq \langle \lambda\, (\, x^c : \tau^c\,)\, \langle d \hookrightarrow e_{\mathtt{DM}} \cdot c \cdot d(\, x^c\,) \rangle_{d \in D} \rangle_{c \in C} \\
e^{\mathrm{I}}_{\mathtt{snd}} &\triangleq \langle \lambda\, (\, x : t_{\mathtt{obj}}\,)\, x \cdot d \rangle_{d \in D}
\end{aligned}$$

and

$$\begin{aligned}
\tau^{\mathrm{II}}_{\mathtt{obj}} &\triangleq [c \hookrightarrow \tau^c]_{c \in C} \\
e^{\mathrm{II}}_{\mathtt{new}} &\triangleq \langle \lambda\, (\, x^c : \tau^c\,)\, c \cdot x^c \rangle_{c \in C} \\
e^{\mathrm{II}}_{\mathtt{snd}} &\triangleq \langle \lambda\, (\, x : t_{\mathtt{obj}}\,)\, \mathtt{case}\, x\, \{ c \cdot x^c \hookrightarrow e_{\mathtt{DM}} \cdot c \cdot d(\, x^c\,) \mid c \in \rho_d \} \rangle_{d \in D}.
\end{aligned}$$

For example, in the case of the complex numbers an object is either a tuple of type

$$\tau^{\mathrm{I}}_{\mathtt{obj}} \triangleq \langle \mathtt{dist} \hookrightarrow \rho_{\mathtt{dist}}, \mathtt{quad} \hookrightarrow \rho_{\mathtt{quad}} \rangle,$$

or an injection of type

$$\tau^{\mathrm{II}}_{\mathtt{obj}} \triangleq [\mathtt{cart} \hookrightarrow \tau^{\mathtt{cart}}, \mathtt{pol} \hookrightarrow \tau^{\mathtt{pol}}].$$

The creation and message sending operations are defined according to the general case given above.

These two representations are "equivalent" in the sense that no client of the dynamic dispatch abstraction can distinguish between them; the client's behavior is the same whichever form is used. The key to proving this is to observe that the typing rule for `open` ensures that (a) the client is polymorphic in the abstract type $t_{\mathtt{obj}}$ of objects, and (b) the client computes a value of an extrinsically meaningful type, one that does not involve $t_{\mathtt{obj}}$. Taken together, this means that the abstract type is interpreted by a binary simulation relation that relates the two implementations. As long as the `new` and `snd` preserve this relation, it is ensured that the client behavior is the same, regardless of which implementation is chosen.

Thus, the first step is to define a simulation relation between the two implementation types, $\tau^{\mathrm{I}}_{\mathtt{obj}}$ and $\tau^{\mathrm{II}}_{\mathtt{obj}}$, that expresses when two values of disparate types are "equivalent" insofar as the `new` and `snd` operations are concerned. So, in what sense is a tuple of methods equivalent to an injected instance value? The question can only be answered by reference to the implementations of the associated operations. Examining the implementation (I), the methods in the tuple are given by the code in the dispatch matrix, specialized to the instance data used to create the tuple. Examining (II), the injection of the instance data is used to select methods from the dispatch matrix appropriate

to that instance. This suggests defining the binary relation R between the two implementation types as follows:

$$R(e^{\mathrm{I}}, e^{\mathrm{II}}) \quad \textit{iff} \quad e^{\mathrm{I}} \longmapsto^* \langle d \hookrightarrow e^{\mathrm{I}}_d \mid d \in D \rangle,\ e^{\mathrm{II}} \longmapsto^* c \cdot e^{\mathrm{II}}_c,\ \textit{and for all } d \in D,\ e^{\mathrm{I}}_d =_{\rho_d} e_{\mathtt{DM}} \cdot c \cdot d(\, e^{\mathrm{II}}_c\,).$$

Because the dynamics is deterministic, and from its definition, the relation R respects evaluation in that $R(e^{\mathrm{I}}, e^{\mathrm{II}})$ iff $e^{\mathrm{I}} \longmapsto^* e$, $e^{\mathrm{II}} \longmapsto^* e'$ and $R(e, e')$.

This relation is preserved by the `new` and `snd` operations. More precisely, interpreting the type t_{obj} by the relation R,

1. If $e^{\mathrm{I}}_c =_{\tau^c} e^{\mathrm{II}}_c$, then $new^{\mathrm{I}} \cdot c(\, e^{\mathrm{I}}_c\,) =_{t_{\mathsf{obj}}} new^{\mathrm{II}} \cdot c(\, e^{\mathrm{II}}\,)$, *i.e.*, $R(new^{\mathrm{I}} \cdot c(\, e^{\mathrm{I}}_c\,), new^{\mathrm{II}} \cdot c(\, e^{\mathrm{II}}\,))$.
2. If $e^{\mathrm{I}} =_{t_{\mathsf{obj}}} e^{\mathrm{II}}$, *i.e.*, $R(e^{\mathrm{I}}, e^{\mathrm{II}})$, then $snd^{\mathrm{I}} \cdot d(\, e^{\mathrm{I}}\,) =_{\rho_d} snd^{\mathrm{II}} \cdot d(\, e^{\mathrm{II}}\,)$

The variables new^{I} and new^{II}, and similarly annotated versions of snd, indicate the implementations in question.

Let us consider the verifications required.

1. By definition
$$new^{\mathrm{I}} \cdot c(\, e^{\mathrm{I}}_c\,) \longmapsto^* e^{\mathrm{I}}_{\mathtt{new}}(\, e^{\mathrm{I}}_c\,) \longmapsto^* \langle d \hookrightarrow e_{\mathtt{DM}} \cdot c \cdot d(\, e^{\mathrm{I}}_c\,)\rangle_{d \in D},$$
and, similarly,
$$new^{\mathrm{II}} \cdot c(\, e^{\mathrm{II}}_c\,) \longmapsto^* c \cdot e^{\mathrm{II}}_c.$$
For these to be related by R, it suffices to show for all $d \in D$,
$$e_{\mathtt{DM}} \cdot c \cdot d(\, e^{\mathrm{I}}_c\,) =_{\rho_d} e_{\mathtt{DM}} \cdot c \cdot d(\, e^{\mathrm{II}}_c\,).$$
Now, by the parametricity theorem, the (c, d) entry of the dispatch matrix is related to itself by equality at type $\tau^c \to \rho_d$. Because it is assumed that $e^{\mathrm{I}}_c =_{\tau^c} e^{\mathrm{II}}_c$, the desired equation follows directly from the definition of equality at a function type.

2. By definition of the implementation
$$snd^{\mathrm{I}} \cdot d(\, e^{\mathrm{I}}\,) \longmapsto^* e^{\mathrm{I}}_{\mathtt{snd}} \cdot d(\, e^{\mathrm{I}}\,) \longmapsto^* e^{\mathrm{I}}_{\mathtt{snd}} \cdot d(\, e^{\mathrm{I}}\,)$$
and
$$snd^{\mathrm{II}} \cdot d(\, e^{\mathrm{II}}\,) \longmapsto^* e^{\mathrm{II}}_{\mathtt{snd}} \cdot d(\, e^{\mathrm{II}}\,) \longmapsto^* \mathtt{case}\, e^{\mathrm{II}}\, \{c \cdot x^c \hookrightarrow e_{\mathtt{DM}} \cdot c \cdot d(\, x^c\,) \mid c \in \rho_d\}$$
Because it is assumed that $R(e^{\mathrm{I}}, e^{\mathrm{II}})$, it follows that
$$e^{\mathrm{I}}_{\mathtt{snd}} \cdot d(\, e^{\mathrm{I}}\,) \longmapsto^* \langle d \hookrightarrow e^{\mathrm{I}}_d \mid d \in D\rangle \cdot d \longmapsto e^{\mathrm{I}}_d$$
and
$$\mathtt{case}\, e^{\mathrm{II}}\, \{c \cdot x^c \hookrightarrow e_{\mathtt{DM}} \cdot c \cdot d(\, x^c\,) \mid c \in \rho_d\} \longmapsto^* \mathtt{case}\, c \cdot e^{\mathrm{II}}_c\, \{c \cdot x^c \hookrightarrow e_{\mathtt{DM}} \cdot c \cdot d(\, x^c\,) \mid c \in \rho_d\} \longmapsto e_{\mathtt{DM}} \cdot c \cdot d(\, e^{\mathrm{II}}_c\,)$$
and
$$e^{\mathrm{I}}_d =_{\rho_d} e_{\mathtt{DM}} \cdot c \cdot d(\, e^{\mathrm{II}}_c\,),$$
which completes the proof.

References

Robert Harper. *Practical Foundations for Programming Languages.* Cambridge University Press, Cambridge, England, Second edition, 2016.