to the second requirement: that Emerald support statically typed polymorphism. Let us first consider an example of implicit polymorphism:

```
const map ← typeobject m
    operation apply[a : t] → [r : t]
        forall t
end m
```

Here is an object that has type *map*:

```
const identity ← object id
    operation apply[e : t] → [r : t]
        forall t
        r ← e
    end apply
end id
```

The operation *apply* of *identity* simply returns its argument; it is the identity function. It is important that the syntactic type of the result be the same as that of the argument; this is achieved by use of the type parameter t, which is introduced by the clause **forall** t, and is bound to the syntactic type of the formal parameter a. The signature of *identity.apply* thus depends on the *type* of its argument e; we can now appreciate why signatures must be functions. Referring back to rule (9) in Section 3, we see that the argument to the signature function is $\langle \tau[\![e]\!], e \rangle$. The signature of *apply* in *map* is $\lambda\langle t, v\rangle.\langle t, t\rangle$; when applied to $\langle \tau[\![e]\!], e \rangle$ the result is $\langle \tau[\![e]\!], \tau[\![e]\!] \rangle$. In the invocation *identity.apply*$[e]$, the type of the argument e trivially conforms to $\mathsf{arg}\ \langle \tau[\![e]\!], \tau[\![e]\!] \rangle = \tau[\![e]\!]$, the conformity condition in the antecedent of rule (9) is always satisfied, and the invocation is type correct whenever the other conditions are satisfied. Moreover, the type of *identity.apply*$[e]$ is $\mathsf{res}\ \langle \tau[\![e]\!], \tau[\![e]\!] \rangle = \tau[\![e]\!]$ as required.