0.1 Types versus classes

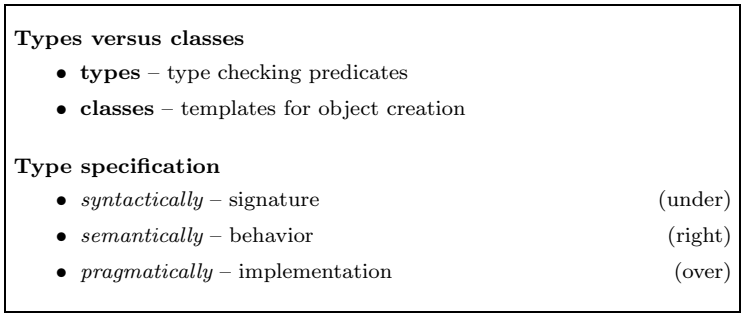
Types are primarily an aid in arriving at a consistent system description. Most (typed) object-oriented programming languages offer support for types by employing classes as a device to define the functionality of objects. Classes, however, have originated from a far more pragmatic concern, namely as a construct to enable the definition and creation of objects. Concluding this chapter, we will reflect on the distinction between types and classes, and discuss the role types and classes play in reusing software through derivation by inheritance. This discussion is meant to prepare the ground for a more formal treatment to be given in the next chapter. It closely follows the exposition given in [WZ88].

Types must primarily be understood as predicates to guide the process of type checking, whereas classes have come into being originally as templates for object creation.

It is interesting to note how (and how easily) this distinction may be obscured. In practice, when compiling a program in Java or C++, the compiler will notify the user of an error when a member function is called that is not listed in the public interface of the objects class. As another example, the runtime system of Smalltalk will raise an exception, notifying the user of a dynamic type error, when a method is invoked that is not defined in the object's class or any of its superclasses. Both kinds of errors have the flavor of a typing error, yet they rely on different notions of typing and are based on a radically different interpretation of classes as types.

To put types into perspective, we must ask ourselves what means we have to indicate the type of an expression, including expressions that somehow reference a class description.

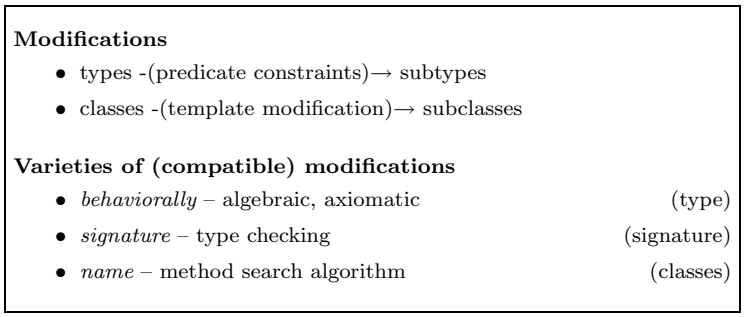
In [WZ88], three attitudes towards typing are distinguished: (1) typing may be regarded as an administrative aid to check for simple typos and other administrative errors, (2) typing may be regarded as the ultimate solution to defining the behavior of a system, or (3) typing may (pragmatically) be regarded as a consequence of defining the behavior of an object. See slide **??**. Before continuing, the reader is invited to sort the various programming languages discussed into the three slots mentioned.



Slide 0-1: Types and classes

Typing as an administrative aid is typically a task for which we rely on a compiler to check for possible errors. Evidently, the notion of typing that a compiler employs is of a rather syntactic nature. Provided we have specified a signature correctly, we may trust a compiler with the routine of checking for errors. As languages that supports signature type checking we may (obviously) mention Java and C++.

Evidently, we cannot trust the compiler to detect conceptual errors, that is incomplete or ill-conceived definitions of the functionality of an object or collections of objects. Yet, ultimately we want to be able to specify the behavior of an object in a formal way and to check mechanically for the adequacy of this definition. This ideal of *semantic types* underlies the design of Eiffel, not so much the Eiffel type system as supported by the Eiffel compiler, but the integration of assertions in the Eiffel language and the notion of *contracts* as a design principle. Pragmatically, we need to rely on runtime (consistency) checks to detect erroneous behavior, since there are (theoretically rather severe) limits on the extent to which we may verify behavioral properties in advance. (Nevertheless, see section **??** for some attempts in this direction.)



Slide 0-2: Type modifications

Finally, we can take a far more pragmatic view towards typing, by regarding the actual specification of a class as an implicit characterization of the type of the instances of the class. Actually, this is the way (not surprisingly, I would say) types are dealt with in Smalltalk. Each object in Smalltalk is typed, by virtue of being an instance of a class. Yet, a typing error may only be detected dynamically, as the result of not responding to a message.

A distinction between perspectives on types (respectively syntactic, behavioral and pragmatic) may seem rather academic at first sight. However, the differences are, so to speak, amplified when studied in the context of type modifications, as for example effected by inheritance.

[WZ88] make a distinction between three notions of *compatible modifications*, corresponding to the three perspectives on types, respectively *signature compatible modifications* (which require the preservation of the static signature), *behaviorally compatible modification* (which rely on a mathematical notion of definability for

a type) and *name compatible modifications* (that rely on an operationally defined method search algorithm). See slide **??**.

Signature compatible modifications The assumption underlying the notion of *types as signatures* is that behavior is approximated by a (static) signature. Now the question is: to what extent can we define semantics preserving extensions to a given class or object?

0-3

Signature compatible modifications

- *behavior is approximated by signature*

Semantics preserving extensions

- horizontal – `Person = Citizen + age : 0..120`
- vertical – `Retiree = Person + age : 65..120`

Principle of substitutability

- an instance of a subtype can always be used in any context in which an instance of a supertype can be used

subsets are not subtypes

Retiree $\not<_{subtype}$ Person

Read-only substitutability

- subset subtypes, isomorphically embedded subtypes

Slide 0-3: The principle of substitutability

When we conceive of an object as a record consisting of (data and method) fields, we may think of two possible kinds of modifications. We may think of a *horizontal* modification when adding a new field, and similarly we may think of a modification as being *vertical* when redefining or constraining a particular field. For example, when we define *Citizen* as an entity with a name, we may define (at the risk of being somewhat awkward) a *Person* as a *Citizen* with an age and a *Retiree* as a *Person* with an age that is restricted to the range 65..120.

The principle by which we may judge these extensions valid (or not) may be characterized as the *principle of substitutability*, which may be phrased as: *an instance of a subtype can always be used in any context in which an instance of a supertype can be used*.

Unfortunately, for the extension given here we have an easy counterexample, showing that syntactic signature compatibility is not sufficient. Clearly, a *Person* is a supertype of *Retiree* (we will demonstrate this more precisely in section **??**). Assume that we have a function

set_age : Person * Integer -¿ Void

that is defined as `set_age(p,n)  p.age = n;` . Now consider the following fragment of code:

```
Person* p = r;                                   r refers to some Retiree
p-¿set_age(40);
```

where we employ object reference notation when calling *set_age*. Since we have assigned r (which is referring to a *Retiree*) to p, we know that p now points to a *Retiree*, and since a *Retiree* is a person we may apply the function *set_age*. However, *set_age* sets the *age* of the *Retiree* to *40*, which gives (by common standards) a semantic error. The lesson that we may draw from this is that being a subset is no guarantee for being a subtype as defined by the principle of substitutability. However, we may characterize the relation between a *Retiree* and a *Person* as being of a weaker kind, namely *read-only substitutability*, expressing that the (value of) the subtype may be used safely everywhere an instance of the supertype is expected, as long as it is not modified. Read-only substitutability holds for a type that stands in a subset relation to another type or is embeddable (as a subset) into that type. See slide **??**.

Behaviorally compatible modifications If the subset relation is not a sufficient condition for being in a subtype relation, what is? To establish whether the (stronger) substitutability relation holds we must take the possible functions associated with the types into consideration as well. First, let us consider what relations may exist between types. Recall that semantically a type corresponds to a set together with a collection of operations that are defined for the set and that the subtype relation corresponds to the subset relation in the sense that (taking a type as a constraint) the definition of a subtype involves adding a constraint and, consequently, a narrowing of the set of elements corresponding to the supertype.

Complete compatibility is what we achieve when the principle of substitutability holds. Theoretically, complete compatibility may be assured when the behavior of the subtype fully complies with the behavior of the supertype. Behavioral compatibility, however, is a quite demanding notion. We will deal with it more extensively in chapter **??**, when discussing *behavioral refinement*. Unfortunately, in practice we must often rely on the theoretically much weaker notion of *name compatibility*.

Name compatible modifications Name compatible modifications approximate behaviorally compatible modifications in the sense that substitutability is guaranteed, albeit not in a semantically verifiable way.

Operationally, substitutability can be enforced by requiring that each subclass (that we may characterize as a pragmatic subtype) provides at least the operations of its superclasses (while giving a sensible result on all argument types allowed by its superclasses). Actually, name compatibility is an immediate consequence of the overriding semantics of derivation by inheritance, as reflected in the search algorithm underlying method lookup. See slide **??**. Although name compatible modifications are by far the most flexible, from a theoretical point of view they are the least satisfying since they do not allow for any theory formation concerning the (desired) behavior of (the components of) the system under development.

Name compatible modifications *0-4*

- operational semantics – no extra compile/run-time checks

```
procedure search(name, module)
if name = action then do action
elsif inherited = nil
                then undefined
else search(name, inherited)
```

Slide 0-4: The inheritance search algorithm