0.1 Flavors of polymorphism

Polymorphism is not a feature exclusive to object-oriented languages. For example the ML language is a prime example of a non object-oriented language supporting a polymorphic type system (see Milner *et al.*, 1990). Also, most languages, including Fortran and Pascal, support implicit conversion between integers and floats, and backwards from floats to integers, and (in Pascal) from integer subranges to integers. Polymorphism (including such conversions) is a means to relieve the programmer from the rigidity imposed by typing. Put differently, it's a way in which to increase the expressivity of the type system.

0-1

Typing – ***protection against errors***

- *static* – type checking at compile time
- *strong* – all expressions are type consistent

Untyped – ***flexibility***

- bitstrings, sets, λ-calculus

Exceptions to monomorphic typing:

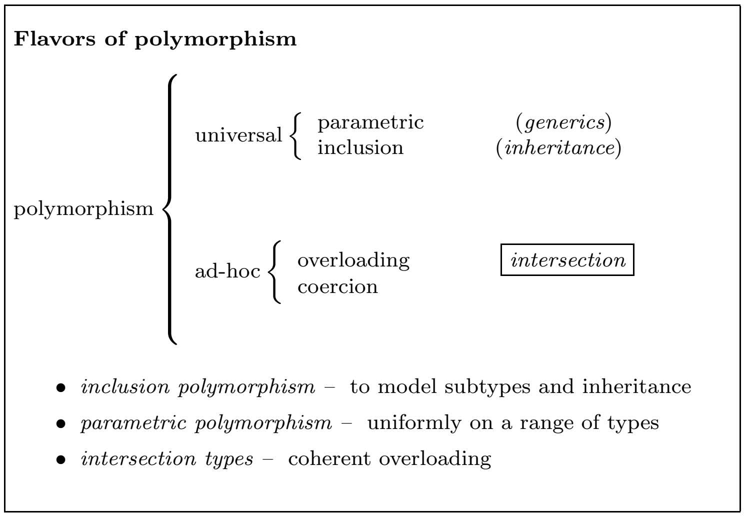
- *overloading, coercion, subranging, value-sharing (nil)*

Slide 0-1: 9-typing

Typing, as we have argued before, is important as a means to protect against errors. We must distinguish between *static typing* (which means that type checking takes place at compile time) and *strong typing* (which means that each expression must be type consistent). In other words, strong typing allows illegal operations to be recognized and rejected. Object-oriented languages (such as Eiffel, and to a certain extent C++) provide strong typing which is a mixture of static typing and runtime checks to effect the dynamic binding of method invocations. See slide **??**.

Typed languages impose rather severe constraints on the programmer. It may require considerable effort to arrive at a consistently typed system and to deal with the additional notational complexity of defining the appropriate types. In practice, many programmers and mathematicians seem to have a preference for working in an untyped formalism, like bitstrings, (untyped) sets or (untyped) lambda calculus. We may further note that languages such as Lisp, Prolog and Smalltalk are popular precisely because of the flexibility due to the absence of static type checking.

For reliable software development, working in an untyped setting is often considered as not satisfactory. However, to make typing practical, we need to relieve the typing regime by supporting well-understood exceptions to monomorphic typing, such as overloaded functions, coercion between data types and value sharing between types (as provided by a generic nil value). More importantly,



0-2

Flavors of polymorphism

$$\text{polymorphism}\begin{cases}\text{universal}\begin{cases}\text{parametric} & (\textit{generics})\\ \text{inclusion} & (\textit{inheritance})\end{cases}\\ \\ \text{ad-hoc}\begin{cases}\text{overloading} & \boxed{\textit{intersection}}\\ \text{coercion} & \end{cases}\end{cases}$$

- *inclusion polymorphism* – to model subtypes and inheritance
- *parametric polymorphism* – uniformly on a range of types
- *intersection types* – coherent overloading

Slide 0-2: Flavors of polymorphism

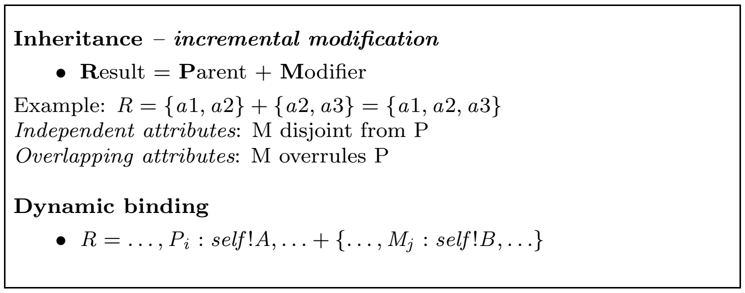
however, we must provide for controlled forms of polymorphism. In [CW85], a distinction is made between *ad hoc* polymorphism (which characterizes the mechanisms mentioned as common exceptions to monomorphic typing) and *universal* polymorphism (which allows for theoretically well-founded means of polymorphism). Universal polymorphism may take the form of *inclusion polymorphism* (which is a consequence of derivation by inheritance) or *parametric polymorphism* (which supports generic types, as the template mechanism offered by C++). See slide **??**. The term *inclusion polymorphism* may be understood by regarding inheritance as a means to define the properties of a (sub)type incrementally, and thus (by adding information) delimiting a subset of the elements corresponding to the supertype. When overloading is done in a systematic fashion we may speak of *intersection* types, which allows for polymorphism based on a finite enumeration of types. See section **??**.

Inheritance as incremental modification

The notion of inheritance as incremental modification was originally introduced in [WZ88]. Abstractly, we may characterize derivation by inheritance in a formula as $R = P + M$, where R is the result obtained by modifying the parent P by (modifier) M. See slide **??**.

For example, we may define the record consisting of attributes $a_1 \ldots a_n$ by adding $\{a_2, a_3\}$ to the parent $\{a_1, a_2\}$. Clearly, we must make a distinction between *independent* attributes (that occur in either P or M) and *overlapping* attributes (that occur in both P and M and are taken to be overruled by the definition given in M).

An important property of objects, not taken into account in our interpretation



0-3

Inheritance – ***incremental modification***

- **R**esult = **P**arent + **M**odifier

Example: $R = \{a1, a2\} + \{a2, a3\} = \{a1, a2, a3\}$
Independent attributes: M disjoint from P
Overlapping attributes: M overrules P

Dynamic binding

- $R = \ldots, P_i : \mathit{self}!A, \ldots + \{\ldots, M_j : \mathit{self}!B, \ldots\}$

Slide 0-3: Inheritance as incremental modification

of *object as records* given before, is that objects (as supported by object-oriented languages) may be referring to themselves. For example, both in the parent and the modifier methods may be defined that refer to a variable *this* or *self* (denoting the object itself). It is important to note that the variable *self* is dynamically bound to the object and not (statically) to the textual module in which the variable *self* occurs. [WZ88] make a distinction between attributes that are redefined in M, *virtual* attributes (that need to be defined in M) and *recursive* attributes (that are defined in P). Each of these attributes may represent methods which (implicitly) reference *self*. (In many object-oriented languages, the variable *self* or *this* is implicitly assumed whenever a method defined within the scope of the object is invoked.) Self-reference (implicit or explicit) underlies dynamic binding and hence is where the power of inheritance comes from. Without self-reference method calls would reduce to statically bound function invocation.

Generic abstract data types

Our goal is to arrive at a type theory with sufficient power to define generic (polymorphic) abstract data types. In the following section, we will develop a number of type calculi (following Pierce, 1993) that enable us to define polymorphic types by employing *type abstraction*.

Type abstraction may be used to define generic types, data hiding and (inheritance) subtypes. The idea is that we may characterize generic types by quantifying over a type variable. For example, we may define the identity function *id* generically as $\forall\, T.id(x : T) = x$, stating that for arbitrary type T and element x of type T, the result of applying *id* to x is x. Evidently this holds for any T.

In a similar way, we may employ type parameters to define generic abstract data types. Further, we may improve on our notion of *objects as records* by defining a packaging construct that allows for data hiding by requiring merely that there exists a particular type implementing the hidden component.

Also, we may characterize the (inheritance) subtyping relation in terms of bounded quantification, that is quantification over a restricted collection of types (restricted by imposing constraints with respect to the syntactic structure of the type instantiating the type parameter).