0.1 Abstraction and types

The concern for abstraction may be regarded as the driving force behind the development of programming languages (of which there are astoundingly many). In the following we will discuss the role of abstraction in programming, and especially the importance of types. We then briefly look at what mathematical means we have available to describe types from a foundational perspective and what we may (and may not) expect from types in object-oriented programming.

0.1.1 Abstraction in programming languages

In [Shaw84], an overview is given of how increasingly powerful abstraction mechanisms have shaped the programming languages we use today. See slide 0-1.

0-1

Abstraction – ***programming methodology***

- control abstractions – *structured programming*
- data abstraction – *information hiding*

The kind of abstraction provided by ADTs can be supported by any language with a procedure call mechanism (given that appropriate *protocols* are developed and observed by the programmer). [DT88]

Slide 0-1: Abstraction and programming languages

Roughly, we may distinguish between two categories of abstractions: abstractions that aid in specifying *control* (including subroutines, procedures, *if-then-else* constructs, *while*-constructs, in short the constructs promoted by the school of *structured programming* in their battle against the *goto*); and abstractions that allow us to hide the actual representation of the data employed in a program (introduced to support the *information hiding* approach, originally advocated in [Parnas72a]).

Although there is clearly a pragmatic interest involved in developing and employing such abstractions, the concern with abstraction (and consequently types) is ultimately motivated by a concern with programming methodology and, as observed in [DT88], the need for reliable and maintainable software. However, the introduction of language features is also often motivated by programmers' desires for ease of coding and naturalness of expression.

In the same vein, although types were originally considered as a convenient means to assist the compiler in producing efficient code, types have rapidly been recognized as a way in which to capture the meaning of a program in an implementation independent way. In particular, the notion of abstract data types (which has, so to speak, grown out of data abstraction) has become a powerful device (and guideline) to structure large software systems.

In practice, as the quotation from [DT88] in slide 0-1 indicates, we may employ the tools developed for structured programming to realize abstract data types in a program, but with the obvious disadvantage that we must rely on conventions with regard to the reliability of these realizations. Support for abstract data types (support in the sense as discussed in section **??**) is offered (to some extent) by languages such as Modula-2 and Ada by means of a syntactic module or package construct, and (to a larger extent) by object-oriented languages in the form of object classes. However, both realizations are of a rather *ad hoc* and pragmatic nature, relying in the latter case on the metaphor of encapsulation and message passing. The challenge to computer science in this area is to develop a notion of types capturing the power of abstract data types in a form that is adequate both from a pragmatic point of view (in the sense of allowing efficient language support) and from a theoretical perspective (laying the foundation for a truly declarative object-oriented approach to programming).

0.1.2 A foundational perspective – types as constraints

Object-oriented programming may be regarded as a *declarative* method of programming, in the sense that it provides a computation model (expressed by the metaphor of encapsulation and message passing) that is independent of a particular implementation model. In particular, the inheritance subtype relation may be regarded as a pure description of the relations between the entities represented by the classes. Moreover, an object-oriented approach favors the development of an object model that bears close resemblance to the entities and their relations living in the application domain. However, the object-oriented programming model is rarely introduced with the mathematical precision characteristic of descriptions of the other declarative styles, for example the functional and logic programming model. Criticizing, [DT88] remark that *OOP is generally expressed in philosophical terms, resulting in a proliferation of opinions concerning what OOP really is.*

From a type-theoretical perspective, our interest is to identify abstract data types as elements of some *semantic* (read mathematical) domain and to characterize their properties in an unambiguous fashion. See slide 0-2.

0-2

Abstract data types – ***foundational perspective***

- unambiguous values in some *semantic* domain

Mathematical models – ***types as constraints***

- algebra – *set oriented*
- second order lambda calculus – *polymorphic types*
- constructive mathematics – *formulas as types*

Slide 0-2: Mathematical models for types

There seems to be almost no limit to the variety and sophistication of the mathematical models proposed to characterize abstract data types and inheritance. We may make a distinction between first order approaches (based on ordinary set theory) and higher order approaches (involving typed lambda calculus and constructive logic).

The algebraic approach is a quite well-established method for the formal specification of abstract data types. A type (or sort) in an algebra corresponds to a set of elements upon which the operations of the algebra are defined. In the next section, we will look at how equations may be used to characterize the behavioral aspects of an abstract data type modeled by an algebra.

Second order lambda calculus has been used to model information hiding and the polymorphism supported by inheritance and templates. In the next chapter we will study this approach in more detail.

In both approaches, the meaning of a type is (ultimately) a set of elements satisfying certain restrictions. However, in a more abstract fashion, we may regard a type as specifying a constraint. The better we specify the constraint, the more tightly the corresponding set of elements will be defined (and hence the smaller the set). A natural consequence of the idea of *types as constraints* is to characterize types by means of logical formulas. This is the approach taken by type theories based on constructive logic, in which the notion of *formulas as types* plays an important role. Although we will not study type theories based on constructive logic explicitly, our point of view is essentially to regard types as constraints, ranging from purely syntactical constraints (as expressed in a signature) to semantic constraints (as may be expressed in contracts).

From the perspective of types as constraints, a typing system may contribute to a language framework guiding a system designer's conceptualization and supporting the verification (based on the formal properties of the types employed) of the consistency of the descriptive information provided by the program. Such an approach is to be preferred (both from a pragmatic and theoretical point of view) to an *ad hoc* approach employing special annotations and support mechanisms, since these may become quite complicated and easily lead to unexpected interactions.

Formal models There is a wide variety of formal models available in the literature. These include algebraic models (to characterize the meaning of abstract data types), models based on the lambda-calculus and its extensions (which are primarily used for a type theoretical analysis of object-oriented language constructs), algebraic process calculi (which may be used to characterize the behavior of concurrent objects), operational and denotational semantic models (to capture structural and behavioral properties of programs), and various specification languages based on first or higher-order logics (which may be used to specify the desired behavior of collections of objects).

We will limit ourselves to studying algebraic models capturing the properties of abstract data types and objects (section **??**), type calculi based on typed extensions of the lambda calculus capturing the various flavors of polymorphism and subtyping (sections **??**–**??**), and an operational semantic model characterizing the behavior of objects sending messages (section **??**).

Both the algebraic and type theoretical models are primarily intended to clarify the means we have to express the desired behavior of objects and the restrictions that must be adhered to when defining objects and their relations. The operational characterization of object behavior, on the other hand, is intended to give a more precise characterization of the notion of state and state changes underlying the verification of object behavior by means of assertion logics.

Despite the numerous models introduced there are still numerous approaches not covered here. One approach worth mentioning is the work based on the *pi-calculus*. The *pi-calculus* is an extension of algebraic process calculi that allow for communication via named channels. Moreover, the *pi-calculus* allows for a notion of migration and the creation and renaming of channels. A semantics of object-based languages based on the *pi-calculus* is given in [Walker90]. However, this semantics does not cover inheritance or subtyping. A higher-order object-oriented programming language based on the *pi-calculus* is presented in [PRT93].

Another approach of interest, also based on process calculi, is the object calculus (OC) described in [Nier93]. OC allows for modeling the operational semantics of concurrent objects. It merges the notions of agents, as used in process calculi, with the notion of functions, as present in the lambda calculus.

For alternative models the reader may look in the `comp.theory` newsgroup to which information concerning formal calculi for OOP is posted by Tom Mens of the Free University, Brussels.

0.1.3 Objectives of typed OOP

Before losing ourselves in the details of mathematical models of types, we must reflect on what we may expect from a type system and what not (at least not currently).

From a theoretical perspective our ideal is, in the words of [DT88], to arrive at a simple type theory that provides a consistent and flexible framework for *system descriptions* (in order to provide the programmer with sufficient descriptive power and to aid the construction of useful and understandable software, while allowing the efficient utilization of the underlying hardware).

Objectives of typed OOP – ***system description*** *0-3*

- packaging in a coherent manner
- flexible style of associating operations with objects
- inheritance of description components – *reuse, understanding*
- separation of specification and implementation
- explicit typing to guide binding decisions

Slide 0-3: Object orientation and types

The question now is, what support does a typing system provide in this respect. In slide 0-3, a list is given of aspects in which a typing system may be of help.

One important benefit of regarding ADTs as real types is that realizations of ADTs become so-called *first class citizens*, which means that they may be treated as any other value in the language, for instance being passed as a parameter. In contrast, syntactic solutions (such as the module of Modula-2 and the package of Ada) do not allow this.

Pragmatically, the objective of a type system is (and has been) the prevention of errors. However, if the type system lacks expressivity, adequate control for errors may result in becoming over-restrictive.

In general, the more expressive the type system the better the support that the compiler may offer. In this respect, associating constructors with types may help in relieving the programmer from dealing with simple but necessary tasks such as the initialization of complex structures. Objects, in contrast to modules or packages, allow for the automatic (compiler supported) initializations of instances of (abstract) data types, providing the programmer with relief from an error-prone routine.

Another area in which a type system may make the life of a programmer easier concerns the association of operations with objects. A polymorphic type system is needed to understand the automatic dispatching for virtual functions and the opportunity of overloading functions, which are useful mechanisms to control the complexity of a program, provided they are well understood.

Reuse and understanding are promoted by allowing inheritance and refinement of description components. (As remarked earlier, inheritance and refinement may be regarded as the essential contribution of object-oriented programming to the practice of software development.) It goes without saying that such reuse needs a firm semantical basis in order to achieve the goal of reliable and maintainable software.

Another important issue for which a powerful type system can provide support is the separation of specification and implementation. Naturally, we expect our type system to support type-safe separate compilation. But in addition, we may think of allowing multiple implementations of a single (abstract type) specification. Explicit typing may then be of help in choosing the right binding when the program is actually executed. For instance in a parallel environment, behavior may be realized in a number of ways that differ in the degree to which they affect locality of access and how they affect, for example, load balancing. With an eye to the future, these are problems that may be solved with a good type system (and accompanying compiler).

One of the desiderata for a type system for OOP, laid down in [DT88], is the separation of a *behavioral hierarchy* (specifying the behavior of a type in an abstract sense) and an *implementation hierarchy* (specifying the actual realization of that behavior). Separation is needed to accommodate the need for multiple realizations and to resolve the tension between subtyping and inheritance (a tension we have already noted in sections **??** and **??**).

Remark In these chapters we cannot hope to do more than get acquainted with the material needed to understand the problems involved in developing a type system for object-oriented programming. For an alternative approach, see [Palsberg94].