1 *Introduction*

It is often stated that object-oriented programming languages are a major improvement over older procedural style languages. If so, why are their static type systems so poor? Some of the static type systems of object-oriented languages are too restrictive, resulting in the need for a plethora of type casts, either checked (as in Java [AGH99]) or unchecked (as in C++ [ES90]). Others allow programs with type errors to be executed. In some of these languages the type errors may be caught at run time (as in the language Beta [KMMPN87]), while in others (like current implementations of Eiffel [Mey92]) the errors may result in run-time crashes.

In this text we will explore the foundations of object-oriented programming languages. Our purpose in examining the formal underpinnings of object-oriented languages is to answer questions like the one in the previous paragraph. This study will help the reader gain deeper insight into the fundamental concepts of these languages. It will help explain why certain features are designed the way they are, as well as provide a tool to help design more expressive, yet statically type-safe, object-oriented languages.

While the first object-oriented language, Simula 67 [BDMN73], was designed and implemented in the mid-60's, and the Smalltalk [GR83] language was first introduced in the early '70's, it wasn't until the advent of C++ in the mid-'80's that a large number of programmers and organizations began adopting object-oriented languages. Even then, many users of C++ simply used it as a "better C" with support for abstraction. However, programmers increasingly adopted pure object-oriented languages like Smalltalk, Eiffel, and, most recently, Java, while an increasing number of C++ programmers write programs in an object-oriented style.

Why has the object-oriented style become so popular? Certainly no small part has been played by the tendency of programmers to jump on the latest "fad" language. However there is real substance behind the reasons for the increasing use of object-oriented languages. There seem to be clear advantages for the object-oriented style in organizing and reusing software components. For example, subtyping and inheritance (notions we will define more carefully later) seem to make it much easier to adapt and reuse existing software components.

However, in many ways the quality of object-oriented programming languages falls short of existing procedural and functional languages. In this text we will focus on two ways in which they fall short – the shortcomings of type systems and the deficiencies in expressiveness of existing object-oriented programming languages.

Based on our years of experience in programming (and teaching programming) in traditional procedural languages such as FORTRAN [Bac81], Pascal [Wir71], C [KR78], Modula-2 [Wir85], and Ada [US 80], as well as functional languages like LISP [MAE$^+$65], Scheme [SS75], ML [MTH90], Miranda [Tur86], and Haskell [HJW92], we are convinced that a strong type system, especially a statically type-safe system, is a very important tool in implementing reliable programs. Thus it would be highly advantageous to provide static type systems for object-oriented languages that are of the same quality as those available for traditional procedural and functional languages, yet make it easy for the programmer to express his or her algorithmic ideas in an object-oriented style.

Unfortunately, commercially available object-oriented languages fall far short of that goal. The static type systems of object-oriented languages tend to be either insecure or more inflexible than one might desire. In some cases the rigidity of the type system leads programmers to rely on type casts (sometimes checked at run time, sometimes not) in order to obtain the expressiveness desired. In other cases, the type systems are too flexible, requiring the run-time system to generate link-time or run-time checks to ensure the integrity of the computation.

1.1 Type systems in programming languages

Type systems in programming languages assign types to all values in a computation. Static type systems also assign type expressions to all expressions of the language. Operations are provided with type information that determines to which types of values they may be applied. For example, a concatenation operator may be restricted to be applied to pairs of strings. An "integer" addition operator may be restricted to be applied only to pairs of integers. A "real" addition operator (which may be represented by the same symbol as the "integer" addition operator) may be restricted to be applied only to pairs of reals. (We treat an overloaded operator symbol or name as referring to multiple operations rather than a single operation with multiple typings.)

Programming languages include primitive data types like integers, reals, booleans, etc., and operations that apply to values of those types. These languages also provide type constructors that allow programmers to build up composite or structured data types (*e.g.*, records or structs, arrays, sets, etc.), as well as providing operations that may construct or be applied to values of these types. In most languages, these more complex types can be named, though their structure is visible and accessible to programmers. While more operations on these types may be designed by the programmer by writing new functions or procedures, these new operations are built from the primitive operations provided by the language. However, any programmer using these structured types may take advantage of the built-in operations to access components of the data structure, by-passing the new operations provided by the type designer. Thus these new type definitions do not appear like predefined types – their structure is visible to all.

ABSTRACT DATA TYPE

The introduction of the notion of *abstract data type* (ADT) [GTW78, Gut77] in the early 1970's, and its introduction in a number of programming languages (*e.g.*, Clu [L$^+$81], Modula-2, and Ada) provided programmers with a mechanism that made it possible to introduce a collection of data type and value definitions, and operations on those definitions, that behaved more like a primitive data type.

ADT's included both a specification and an implementation, which were usually provided separately. The ADT specification provided a name for the type and provided specifications, both type and behavioral, for a collection of operations on the type. The type specification for an operation includes the types of the parameters, if any, and the return type. We will refer to such a type specification as the *signature* of the operation. These specifications were usually packaged together, and provided sufficient information for a programmer to write programs that used the type. The ADT implementation provided a representation for the values of the type, typically as a structured data type, and the implementations of the operations, written as procedures and functions that were allowed to access the representation of the data type.

SIGNATURE

Programmers using ADT's were not allowed access to the implementation of a data type, thus making it easier to replace one implementation of

INFORMATION HIDING

an ADT by another. This *information hiding* was an important feature of the use of ADT's. Early language mechanisms that provided support for ADT's included Clu's clusters, Modula-2's modules, and Ada's packages. ML's signatures and structures later provided similar mechanisms.

Object-oriented languages introduced the notions of classes and objects. *Objects* contain both state (values) and methods (operations). The main operation provided for objects is sending a *message* to an object. *Classes* provide both specification and implementation information on objects. Not only are the names and specifications of methods included in classes, but also representation information for the state and methods. Most object-oriented languages provide mechanisms for allowing the programmer to restrict access to the representation of the state or methods of objects from clients or subclasses in order to support information hiding.

Some object-oriented languages also allow programmers to provide only specification information on objects. For example, several languages allow the programmer to provide pure *abstract* (C++, Java) or *deferred* (Eiffel) classes. The programmer simply provides method names and signatures, omitting all mention of the representation of state and implementations of methods. Java's interfaces, while they may have initially been included to provide support for some aspects of multiple inheritance, provide a clean representation for this separation of interface and implementation. Several classes with entirely different representations may implement the same interface. A procedure or function whose parameter type is given by an interface can take as actual parameters objects generated from any class that implements the interface. This promotes a notion of reusability that is essentially independent of the notions of inheritance and subtyping.

Languages like Ada, Clu, and ML allow the user to define parameterized types (*e.g.*, Stack(T), Tree(T), etc.). These can be seen as functions that take types as parameters and return new types. These languages also typically allow the programmer to define polymorphic functions (functions that take types as parameters, but return values rather than types). There appears to be a strong correlation between the increased expressiveness of programming languages and the increasing richness of their type systems.

1.2 Type checking and strongly typed languages

TYPE SYSTEM

Type systems for programming languages are typically designed to provide several important functions. These include:

- Safety: Type checking of programs should prevent (either at compile or run time) the execution of certain illegal operations. In Chapter 13 we go into more detail on which illegal operations type systems are responsible for preventing. For now, we simply provide the examples of attempting to add a string to an integer as a type error, and dividing an integer by zero as a non-type error.

  The first is a type error because that operation should never be applied to two operands, one of which is a string and the other of which is an integer. The second is not a type error because division is an operation that is normally applied to pairs of integers. However, when the operation is applied to certain combinations of values from those types, an error results. Thus, information on the types of the operands is not sufficient to determine whether the operation will be erroneous.

- Optimization: Type checking can provide useful information to a compiler or interpreter. This information can be used to allocate storage for values, select appropriate code to execute (*e.g.*, for overloaded operations), and support various optimizations.

- Documentation: Type annotation (or, to a lesser extent, inference) provides documentation on constructs that make it easier for programmers to determine how the constructs can or should be used. Of course, the programmer should provide more than just type information as documentation, but our experience is that omission of type information significantly impacts the comprehensibility of code.

- Abstraction: The ability to name types, and, even more importantly, the ability to hide the implementation of types, allows (even forces) the programmer to think at a higher level of abstraction in programming. This hiding of details allows more straightforward modeling of the problem domain, while making it possible to change the implementation of a type and its operations without impacting the correctness of programs using the implementation. Of course, an important reason for changing an implementation is to improve some aspect of the behavior of the program, but correctness of the program should be dependent only on the specification of the provided operations.

STRONGLY TYPED LANGUAGE

Every value generated in a program is associated with a type, either explicitly or implicitly. In a *strongly typed* language, the language implementation is required to provide a type checker that ensures that no type errors will occur at run time. For example, it must check the types of operands in order to ensure that nonsensical operations, like dividing the integer 5 by the string "hello", are not performed. Strongly typed languages may either be dynamically or statically type checked. Dynamic type checking normally occurs during program execution, while static type checking occurs prior to program execution, typically at compile time.[1] Other type-related checks may take place at program link time.

DYNAMICALLY TYPED LANGUAGE

In a *dynamically typed* language like LISP or Scheme, many operations are type checked just before they are performed. Thus the code for a plus operation may have to check the type of its operands just before the addition is performed. If both operands are integers, then an integer addition is performed. If both operands are floating point numbers or one is floating point and the other is an integer, then a floating point addition is performed. However, if one operand is a string and the other is a floating point number, then execution is terminated with an error message. In some languages an exception may be raised, which may either be handled by the program before resuming normal execution or, if there is no handler or no handler can successfully execute, the program terminates.

STATICALLY TYPED LANGUAGE

In a *statically typed* language, every expression of the language is assigned a type at compile time. If the type system can ensure that the value of each expression has a type compatible with the statically assigned type of the expression, then type checking of most operations can be performed at compile time, rather than delayed to run time.

Dynamically typed programming languages can be more expressive and flexible than statically typed languages, because the type checking is postponed until run time. In general, the problem of determining statically for an arbitrary program whether a type error will occur at run time is undecidable,[2], yet it is generally accepted that a static type system should be decidable. As a result, sound static type checkers will rule out some programs as potentially unsafe that would actually execute without a type error.

While the exclusion of safe programs would seem to be a major problem with static type checking, there are many advantages to having a statically type-checked language. These include:

- providing earlier, and usually more accurate, information on programmer errors,

1. For convenience, we will refer to static checks as occurring at compile time, even though similar checks take place before execution in interpreted as well as compiled languages.
2. We leave it as an exercise for the more sophisticated reader to show this problem can be reduced to the halting problem. Hint: Have a type error result only if a program that is input as data halts.

- eliminating the need for run-time type checks that can slow program execution and increase program size,
- providing documentation on the interfaces of components (*e.g.*, procedures, functions, and packages or modules), and
- providing extra information that can be used in compiler optimizations.

As a result most modern languages have static type systems.

Procedural languages like Pascal [Wir71], Clu [L^+81], Modula-2 [Wir85], and Ada 83 [US 80], and functional languages like ML [HMM86] and Haskell [HJW92] have reasonably safe static typing systems. While some of these languages have a few minor holes in the type system (*e.g.*, variant records in Pascal), ML, Haskell, CLU, and Ada provide fairly secure type systems.

Programmers used to dynamically type-checked languages may worry that the use of a static type system will disallow or restrict the use of programs that can be dynamically determined to be type safe. For example, the static type system of standard Pascal is so inflexible that it will not allow the programmer to write a single sort procedure that will work for integer arrays of different sizes, let alone for arrays of other types like reals or characters. The language C has a similarly restrictive type system, but provides specific mechanisms (type casts) to allow the programmer to bypass the static type system when it gets in the way of the programmer.

However, modern programming languages allow more flexible use of arrays as parameters and often include support for more advanced features, such as parametric polymorphism, that have increased the expressiveness of statically typed languages. Examples of statically type-safe, yet flexible, procedural and functional programming languages include Clu, Modula-2, Ada, ML, and Haskell.

Unfortunately the situation for static type checking in object-oriented languages is not as good. The following is a list of some properties of type-checking systems of some of the more popular object-oriented languages (or the object-oriented portions of hybrid languages).

- Some provide only dynamic type checks.
  Smalltalk
- Some are mainly statically type-safe (if no casts), but inflexible. These languages often require explicit mechanisms to escape from the type system (*e.g.*, unsafe type casts) to overcome deficiencies of the type system.
  C++, Object Pascal

- Some have very flexible static type systems, but the type systems as implemented are not sound.
  Eiffel

- Some are flexible, but need run-time type checks to overcome weaknesses in static typing.
  Beta, Java, Ada95

At the boundary between static and dynamic type systems are several constructs. Here there may be differences of opinion on what features are considered to be part of static type systems and which are part of dynamic systems.

For example, we consider constructs like `typecase` statements, which make explicit tests on the run-time type of a value, to be statically type-safe as long as the execution of such statements cannot give rise to run-time type errors or system-generated exceptions. An example of the use of such a construct in the language Theta [DGLM94] is given below. Assume the identifier `x` is declared with static type `S`, and assume that `T` and `U` are subtypes of `S`.

```
typecase x
   when T(t): ... t ...
   when U(u): ... u ...
   others: ... x ...
end;
```

In this statement, if `x`'s run-time type is a subtype of `T`, the value of `x` will be denoted by `t` (which is an identifier with static type `T`), and the code following the first `when` clause will be executed. Similarly, the code in the second `when` clause will be executed (with u denoting the value of `x`) if the run-time type is a subtype of `U`, but not of `T`. Finally, if the run-time type of `x` fails to be a subtype of any of the types listed in the `when` clauses, then the code in the `others` clause will be executed. This is type safe because each of the branches is required to type check correctly.

No run-time type errors can occur, because if `x` has a type that is not a subtype of the types specified in the `when` clauses, the code in the `others` clause will be executed, and it must be type safe for `x` having static type `S`.

Eiffel's "reverse assignment" involves an assignment from an expression with static type `T` to a variable whose static type `S` is a subtype of `T`. We consider this to be in the same category as `typecase`.

Suppose `x` is declared to have type `S`, where `S` is a subtype of `T`, the static type of `exp`. Then the statement

```
x ?= exp;
```

will type check. If the run-time type of `exp` is a subtype of `S`, the value of `exp` will be stored in the location corresponding to `x`. However, if the run-time type of `exp` fails to be a subtype of `S`, the value `void` is assigned to `x`. Thus in neither case does a run-time type error or system-generated exception occur.

This reverse assignment can be understood as a very restricted form of `typecase`. We can code the reverse assignment above using `typecase` as follows:

```
typecase exp
   when S(s): x := s;
   others:    x := void;
end;
```

On the other hand, we treat Java's type cast as not being statically type safe because the failure of a cast raises a run-time exception.[3]

As we shall see later, type restrictions on the redefinition of methods in many object-oriented languages give rise to situations where programmers often feel the need to by-pass the static type system. Some of these type restrictions follow from the need to preserve type safety when redefined methods are used in combination with inherited methods. Other restrictions are due to the desire to have subclasses always generate subtypes. While the introduction of bounded parametric polymorphism has helped loosen some of the rigidities of these languages, programmers of statically typed object-oriented languages are more likely to feel that static type safety gets in their way than programmers in statically typed procedural or functional languages.

As a result, in choosing from existing statically typed object-oriented languages, programmers are faced with unfortunate choices for overcoming the deficiencies of the type systems. They may attempt to program around these deficiencies, use constructs that require dynamic type checking, or use languages that allow run-time type errors to occur.

We make the case in this book that it is possible to define safe statically typed object-oriented languages that are sufficiently expressive to obviate the need for either run-time type checks or ways of escaping the type system. While borderline features like `typecase` statements or run-time checked

3. If Java could somehow guarantee that an `instanceof` check occurred before every type cast, like `typecase` statements in some languages, we would consider this to be a statically type-safe operation.

reverse assignments may occasionally be necessary to handle difficult problems with heterogeneous data structures, we prefer to have type systems that allow us to program as naturally as possible, while catching all type errors.

As we shall see in the course of this text, many type problems and rigidities arise in statically typed object-oriented languages because of the conflation of type with class, and with the mismatch of the inheritance hierarchy with subtyping. Whatever the cause, there appears to be much room for improvement in moving toward a combination of better security and greater expressiveness in the type systems.

1.3 Focus on statically typed class-based languages

In this text we explore the foundations of object-oriented languages by paying careful attention to the design of type systems and semantics for object-oriented languages. We will focus particularly on static type systems for class-based object-oriented languages.

There are great advantages to using statically typed languages; for example in helping programmers find and fix errors more efficiently. On the other hand, the restrictions on expressiveness can lead programmers to use languages that are not statically type safe or to find ways of by-passing the type system when it gets in the way. One of the goals of research in this area has been to ameliorate these inherent conflicts by designing language constructs that are both statically type safe and provide increased expressiveness.

Our focus on class-based rather than object-based languages comes from both practical and conceptual considerations. Class-based languages rely on classes that form templates for the generation of new objects. Object-based languages allow programmers to define objects directly, and usually provide mechanisms, for example prototypes, delegation, and cloning operations, for the creation of new objects from old. Like all distinctions in computer science, there is blurring at the edges between this categorization of languages, but the distinctions provided by this categorization are useful. (See Section 7.1.1 for a more detailed description of object-based languages.)

Virtually all popular object-oriented languages (*e.g.*, Simula 67, Smalltalk, Object Pascal, Eiffel, Objective C, C++, Ada95, and Java) are class-based. On the other hand, object-based languages (*e.g.*, Self, Cecil, and Emerald) tend to be research languages or are used by relatively small communities. Of course this popularity is not an indication that class-based languages are necessarily better, but it does suggest that there may be more interest in achieving a better understanding of class-based languages.

There are also conceptual reasons for preferring to analyze class-based languages. In class-based languages, classes and objects separate important concerns. Classes form extensible templates that can be used to create new objects. Objects are the fundamental components of computation, with computation taking place by sending messages to objects. The execution of methods of an object may update its state (instance variables), but no mechanism is provided to update or add methods to existing objects. In class-based languages methods in classes may be updated by using the mechanism of inheritance to create a new subclass with the updated (or added) method. In object-based languages, the methods of objects may be updated in place or (depending on the language) be updated in the creation of a new object based on the original.

In object-based languages, objects essentially play the role of both classes and objects in class-based languages. This causes complications in providing theoretical modeling of these languages, especially in providing support for method update or addition of methods in objects. At this point, it is hard to explain the technical reasons for these difficulties without going into a much more detailed discussion of the modeling of instance variables, methods, and, particularly, the modeling of `self` (written `this` in Java and C++), a keyword representing the object currently executing a method. We will discuss some of these difficulties later in Chapter 7; for now we hope the reader is satisfied with these explanations.

Not all other researchers agree with our views on this topic. For example, Abadi and Cardelli, in their very influential text, *A Theory of Objects* [AC96], argue that objects are more primitive than classes, and that mechanisms other than classes are useful in generating objects with common properties. Moreover they argue that classes are superfluous because they can be defined in terms of objects. This allows them to start with a very simple object calculus and define a variety of mechanisms (including classes) for generating objects. The associated cost is that it is more complex to model their object calculus in terms of the lambda calculus or denotational semantics in such a way as to preserve subtyping. (See Chapter 7 for a comparison.)

1.4 Foundations: A look ahead

We will begin this text by analyzing existing object-oriented programming languages, paying special attention to their type systems and impediments to expressiveness. We explore why type systems for these languages include what may at first seem to be rather arbitrary restrictions, and the consequences of ignoring these restrictions. It will become clear that there are a number of constructions that programmers would like to be able to express in these languages, but that are not currently supported in many existing statically typed object-oriented languages. In some cases, relatively simple extensions to these languages can greatly enhance expressiveness while preserving type-safety (see the discussion in Chapter 4 of the extension, GJ, of Java for one example). In other cases, attempts to add expressiveness have resulted in either type insecurities or the need to add dynamic type checking (see the discussion of Eiffel in the same chapter).

In Chapters 5 and 6 we examine the definitions of two key features of object-oriented languages: subtypes and subclasses. In particular we investigate conditions that guarantee that two types are subtypes. We also look at restrictions necessary to ensure that inherited methods in subclasses remain type correct.

We end the first part of the book with a discussion of different kinds of object-oriented languages (e.g., class-based, object-based, and multi-method languages) and an examination of statically typed object-oriented languages Simula 67, Beta, Java, C++, Smalltalk, Eiffel, and Sather with reference to our model languages and type systems.

In order to support a careful analysis of the type systems and semantics of object-oriented languages, we will introduce a prototypical object-oriented language, $\mathcal{SOOL}$, with a simple type system that is similar to those of class-based object-oriented languages in common use today. After a discussion of subtypes and subclasses (especially with regard to type restrictions on overriding methods), we begin an analysis of the foundations of object-oriented languages by providing a semantics. The semantics will allow us to precisely specify the meaning of these languages, enabling a more careful examination of the rules sufficient to guarantee the type safety of various programming constructs.

There are many alternatives available for providing the semantics of object-oriented languages. A denotational semantics would provide a mathematical specification of meaning. An operational semantics would specify the meaning of programs by providing instructions for an interpreter that would execute programs using a very simple virtual machine. One might also provide an axiomatic semantics that would provide rules for reasoning about programs. While there are advantages to each of these, and in other situations we have been quite happy with the provision of an operational semantics, we have taken a different approach here.

Our semantics provides the meaning of programming constructs by translating them to an extended typed lambda calculus. The main advantage of a typed lambda calculus is its simplicity. The core of the calculus is the representation of functions and function application; concepts that are learned quite early in mathematics courses. While the notation may initially be unfamiliar, the ideas behind the calculus should be familiar to all readers. Also rather than restricting ourselves to a stripped-down, "pure" lambda calculus, we add familiar programming constructs such as records, pairs, and references. We also extend the lambda calculus with less familiar notions, such as parametric polymorphism and existential types, that will help to model parameterized classes and information hiding.

Another advantage of providing a translational semantics based on the lambda calculus is that these calculi have been studied in great detail over the years. As a result, rather than providing very detailed and technically intricate proofs of type soundness and safety, we simply show that our translation preserves types. This will enable us to lift type soundness and safety results from the lambda calculus to our object-oriented language. While soundness and safety proofs are of interest in their own right, our goal here is to provide explanations of typing issues in object-oriented languages to a larger audience. Thus we include only the proofs we feel are most necessary in order to provide convincing evidence that our semantics are correct and that the type system is safe. As a result, we do not hesitate to base our results on systems that are intuitively (as well as provably) safe. We provide pointers to the literature for readers who are interested in complete proofs from first principles.

After the introduction to our extended lambda calculus in the second part of the book (Chapters 8 and 9), we begin the third part of the book with a careful formal definition of our prototypical language, $\mathcal{SOOL}$. In Chapter 11, we begin the task of modeling the semantics of $\mathcal{SOOL}$. While modeling of objects and classes will turn out to be rather straightforward, the modeling of subclasses is surprisingly tricky if we hope to preserve type safety. However, the correct modeling provides an explanation for the difficulties in type checking methods that arise if we wish to guarantee that inherited methods remain type safe in subclasses. As one might hope, our modeling of object-oriented languages will suggest the addition of new constructs to the language (e.g., `MyType`) as well as to help us understand the type-checking rules of object-oriented languages. This modeling leads into one of the most technical chapters of the text, Chapter 13, in which we prove that the type system is sound by showing that our semantics preserves typing information. We finish this part of the book by adding some common features that were omitted to simplify the original presentation and proof. These include references to methods in the superclass, the handling of null references, more refined information hiding, and multiple inheritance.

In the last part of the book (Chapters 15 through 18) we add desirable features that are not yet included in many statically typed object-oriented languages. These new features include parametric polymorphism (including what is sometimes known as F-bounded polymorphism), and a `MyType` construct. The combination of these features allows us to overcome many of the expressiveness limitations of existing statically typed object-oriented languages. We end the book with the sketch of a language that includes the `MyType` construct and drops subtyping for a slightly weaker relation, called matching.

There is much more material that could be included in a text on this subject. For example, we were tempted to include operational semantics for object-oriented languages, and we would have liked to include more material on virtual types and modules. However, our primary goal is to provide in a fairly compact form a good introduction to the concerns in designing safe, yet expressive, object-oriented programming languages. We hope that the following chapters will successfully achieve this goal. After completing this text, the reader should be prepared to go to the research literature to find information on these other topics.