Chapter 1

Introduction

The goal of this work is the design of a "language with class", an object-oriented programming language supported by a formal theory of classification.

Programming languages as different as Ada, Smalltalk, ML and Eiffel offer various attractive, but partial views of type abstraction. The motivation for the current work arises from a dissatisaction with the number of different mechanisms used to explain type abstraction, such as "type constructors", "generic parameters", "classes", "inheritance", "polymorphism" and "overloading". Accordingly, the focus here is on finding a unifying framework within which all the above mechanisms are related. Central to this is a properly-constructed mathematical notion of "class".

1.1 In Search of Class

Back in 1982, Rentsch correctly predicted:

> "My guess is that object-oriented programming will be in the 1980s what structured programming was in the 1970s. Everyone will be in favour of it. Every manufacturer will promote his products as supporting it. Every manager will pay lip-service to it. Every programmer will practise it (differently). And no-one will know just what it is" [Rent82, p51].

Object-orientation was then, and still is, in its ascendancy. It is viewed variously as a technique for developing better models of computer-human interaction [GR83], for specifying robust, interchangeable software components [Meye88, IHBK79] or simply offering more flexible and extensible styles of programming [BDMN73, Stro86, Keen89]. Many new programming languages have appeared, supporting the notions of *classification* and

incremental program development through *inheritance*. However, these notions are still not well understood; and then only in naïve operational terms.

1.1.1 A Failure of Nerve

Early attempts to formalise classification foundered on certain naïve assumptions about type made by exemplar languages, such as *Smalltalk* [GR83]. A focus on incremental programming techniques forced an early division between the notions of *type* and *class* [Snyd87, Amer90], which were viewed as dual specification and implementation constructs. A class was regarded merely as an extensible implementation pattern, a concrete data type with no intrinsic theoretical importance. If classification were no more than this, object-oriented programming would be just another (slightly different) approach to system modularisation. The inability to characterise *class* persists to this day and is reflected in the Object Management Group's studious avoidance of the term, adopting instead the dual notions of *interface* and *type* [OMG91].

1.1.2 A Manifesto

It seems unfair that the programming paradigm which brought such fascinating notions as classification and inheritance out of the semi-formal playground of Artificial Intelligence into the more rigorous world of mathematical data types should be dismissed as theoretically uninteresting! No other group of programming languages has attempted to generalise over types in quite the same way, for though conventional *Hope* [BMS80], *ML* [MTH90] and *Ada* [IBHK79] introduced type parameters to abstract over parts of generic types, only the object-oriented languages tried to link all types in an ordered hierarchy. With these different views of type abstraction came different mechanisms for handling polymorphism. In the classical model, explicit type parameters were statically replaced at compile-time, reminiscent of type constructors, whereas the object-oriented model seemed to offer a mixture of static function inheritance and dynamic selection of one from a group of functions overloaded on the same name.

Could all these approaches be describing the same kind of polymorphism? Was there more to the object-oriented notion of *class* than simply an extensible data type? What was the mathematical foundation for inheritance? How could *Smalltalk* reject types and yet espouse an even more challenging notion of classification? How could *Eiffel* [Meye88] reconcile supporting both classical and modern mechanisms for polymorphism? These and other questions prompted our manifesto [SC92] setting out to harmonise notions of *class*, *type*, *inheritance* and *polymorphism*, in much the same spirit as an earlier, more famous, discourse [CW85]. In the current work, the notion of *class* turns out to be the single most illuminating and unifying concept behind all type abstraction. Unlike the impoverished view of *class* accepted widely today, this investigation supports an elevated view of *class* as something denoting a polymorphic family of types sharing the same recursive structure.

1.2 A Guide to the Thesis

Classification lies at the heart of the object-oriented family of programming languages. This notion permeates all other language concepts central to the object-oriented paradigm, namely: *objects*, *encapsulation*, *inheritance* and *polymorphism*. These key concepts are introduced in chapter 2, along with a presentation of the state of the art in object-oriented thinking.

1.2.1 A Practice Lacking a Theory

Chapters 2-4 present the argument that object-oriented languages have developed in advance of a formal theory of classification and, as a result, their treatment of *class* is muddled and ill-founded. The operational behaviour of existing object-oriented languages is often described incorrectly using a terminology of types which strictly does not apply. The type systems of languages affected by such misunderstandings are typically incorrect and inconsistent [Cook89b]; furthermore, they may contain redundant mechanisms for handling type polymorphism as a result of the central ambiguity surrounding the notion of *class* [Simo94a, Simo95]. Given the current popularity of the object-oriented approach, correcting these faults is a timely concern. Chapters 5-6 provide a sound formal basis for the notion of classification as found in object-oriented languages. Chapters 7-9 elaborate on a design for a programming language, which supports classification in a clear and consistent way.

1.2.2 A Theory of Class and Type

The approach taken here is based on properly relating the notions of *class* and *type*. Some unhelpful ways in which these notions have been related in the past are described in the latter part of chapter 2. The main theoretical treatment begins in chapter 3 by exploring the limits of Cardelli's first-order theory of types and subtyping [CW85, Card88a]. This is later supplanted in chapter 4 by Cook's second-order theory of F-bounded quantification [Cook89a, CCHO89a], which captures the notion of type inheritance and the evolution of the *self*-type in recursive types. In this approach, a *class* is shown to be a generalisation of the notion of *type*, standing for a family of different, but structurally related, recursive types.

The F-bounded model of inheritance, presented in full in chapter 5, was originally developed by a research team at Hewlett-Packard in the late 1980s, who were working on a theoretical base language called *Abel*, named after Abelian groups, which aimed to support all the familiar concepts of object-oriented programming in a sound mathematical framework. The *Abel* team succeeded in describing the inheritance of object implementations, type interfaces and constructor functions [CP89, CCHO89b, CHC90]; however the project was curtailed leaving many important aspects of this work unfinished [Harr91]. In particular, though the F-bounded model had been applied to inheritance and polymorphism in the type of *self*, it stopped short of handling multiple inheritance and general polymorphism. This was partly due to self-imposed restrictions in the adoption of a simply-typed record combination

operator [CHC90], used to construct derived classes during inheritance. As a result, the full explanatory power of the F-bounded approach was never exploited.

In chapter 5, Cook's theory is extended with *dependent* second-order types, in order to motivate a second-order typing for Cook's record combination operator. Using this approach, a second-order typing is provided for freestanding extensions to classes, known as *mixins*, motivating a combination strategy that allows the derivation of classes from collections of typed mixins. Later in chapter 6, a similar approach is used to handle multiple inheritance. The same chapter investigates the extra complexity introduced by types with internal polymorphism, showing how the relationship between a type and a class is exactly the same as that between a type constructor and a class with polymorphic components. Whereas a second-order theory is appropriate when abstracting over types, a higher-order theory is required to abstract over type constructors. General polymorphism fits this pattern; accordingly, two higher-order record combination operators are defined to handle inheritance with overriding and multiple inheritance with conflict resolution.

An innovation in this approach is the development of a single parametric model for all systematic kinds of polymorphism, which previously have been analysed using different mechanisms such as subtyping, inclusion polymorphism (also known as *conformance*) and generic, or parametric polymorphism [CW85]. This provides significant insights into the formal understanding of type abstraction in programming languages. F-bounded quantification turns out to be as useful for describing the parameterised components of type constructors as it is for describing the *self*-type of recursive classes. Nested F-bounded quantification can be used to model classes with internal polymorphic components. The order of replacement of type parameters determines whether a recursive class or a type constructor is produced. Chapter 8 later illustrates how the order of quantification determines whether a recursively defined polymorphic sequence contains homogeneous or heterogeneous elements.

1.2.3 A Language with Class

Chapter 8 presents a programming language based on the general theory of classification, adopting a minimal textual style consistent with the goals of clarity, economy and consistency. All the concrete features of the programming language syntax are explained in terms of constructions in the mathematical model. As a principal feature, the language maintains the correct mathematical relationships between types, type constructors and classes, distinguishing these concepts unambiguously in its syntax. The language supports single and multiple inheritance among classes and the definition of fully abstract classes. The latter fulfil an important role in mapping out type spaces. Apart from generalising over families of types with disjoint implementations, they are useful in the resolution of multiple inheritance conflicts. All forms of polymorphism are handled using F-bounded type parameters. In particular, no false syntactic distinction is necessary to force the run-time resolution of polymorphism using dynamic binding, as against

compile-time resolution using static parameter replacement. Parameterised classes yield type constructors, whose parameters may be replaced either at compile-time or run-time, offering both the advantages of early type construction with static binding and delayed type instantiation with dynamic binding.

Apart from those aspects pertaining to classification, the language also supports important properties of objects, such as identity and encapsulated state. In the first part of chapter 7, encapsulation is handled in the formal model using functional closures. Object constructor functions, which abstract over the state of any particular instance, are used to generate closures. Identity is consequent on admitting assignment to object states, at which point it becomes relevant whether a value or reference semantics is intended when objects are passed as arguments or assigned to variables. The second part of chapter 7 analyses the costs and benefits of supporting both value and reference semantics and provides a novel strategy for handling aliasing that respects object identity without violating object state. Another important technical aspect of a programming language is the handling of control-flow. Chapter 9 compares the merits of admitting primitive selection and iteration constructs (as used in *C++* and *Eiffel*) against dynamic binding and encapsulated mapping functions (as used in *Smalltalk*).

Chapter 10 investigates how the language may be compiled, using a translation model that optimises space- and time-costs [SLN94]. A static analysis of closed programs permits the early binding of some 80% [Booc94] of system functions and allows various kinds of automatic inlining [CUL89]. A novel strategy is presented for collapsing inheritance graphs during the translation from library software into target production code. This may reduce the size of the data segment in the runtime image by some 30% [SLN94]. The new parametric type model presented earlier permits a finer-grained static analysis of bindings than current production compilers can effectively exploit. In order to profit from these kinds of optimisation, new strategies for generating and linking object-code modules must be developed. The latter task is a matter for further research and development.

1.2.4 Summary of Themes

The theory of classification presented here provides a coherent framework for explaining all kinds of type abstraction found in programming languages as diverse as *Ada*, *Smalltalk*, *ML* and *Eiffel*. In particular, operational descriptions of object-oriented languages [GR83] may be given a formal interpretation in this model. The notion of *class* is elevated to a higher-order typing construct with an associated implementation. The theory models classes, inheritance and polymorphism. Reflecting the theory, a programming language is developed whose major innovation is a single parametric treatment of all forms of polymorphism. An optimising compiler for the language is sketched, showing how type information may be exploited in full.