A PROPOSAL FOR HARMONISING TYPES, INHERITANCE AND POLYMORPHISM FOR OBJECT-ORIENTED PROGRAMMING

A J H Simons and A J Cowling, Department of Computer Science, Regent Court, University of Sheffield, 211 Portobello Street, SHEFFIELD S1 4DP.

1. INTRODUCTION

Object-Oriented Programming (OOP) has attracted a large following in recent years. Goals for the paradigm have included bringing computational implementations closer to abstractions (the HCI school, exemplified by Smalltalk [1]), partitioning spaces of concepts (the AI school, exemplified by CLOS [2]) and software engineering for reuse and extensibility (the SE school, exemplified by Eiffel [3]). The present trend is away from type-free or dynamically typed languages and towards strong static typing, with some dynamic binding. Attempts to provide formal typed models of OOP have been frustrated by the great flexibility in languages that permit unrestricted redefinition of class properties. The principal difficulty lies in devising a typed functional algebra that conforms to the operational definition of OOP languages, which are usually described in completely different terms.

We take the view that types, rather than restricting the usefulness of an OOP language, provide an expressive framework for defining the protocols of classes of object. Even languages that do not support typing start from a strong notion of an intrinsic set of attributes and operations that characterise a class. Concept spaces are defined by a process of explicit differentiation of class attributes and operations. The fact that the resulting class hierarchy often fails to conform to a type-subtype hierarchy has been an embarrasment to OOP practioners favouring the strongly typed approach [4,5]. The reasons for this are complex, involving issues in inheritance and polymorphism, but which, for the moment, we shall summarise as the insufficient axiomatisation of abstract types when designing concrete class specifications. It would seem self evident that a thorough-going type model for OOP would provide enormous benefits from the software engineering viewpoint. Programs would be demonstrably type-secure and optimising compilers could exploit, wherever possible, the static binding of operations to objects.

In the rest of this paper, section 2 presents a rationale for relating the notions of class and type. Section 3 reviews classic approaches to the mathematical definition of types. Section 4 considers strong and weak inheritance and defines a space of inheritance schemes that conform to strict subtyping under monomorphism. Section 5 introduces polymorphism to the model, contrasting various interpretations of

Strachey's original insight and proposing a scheme which subsumes parametric and some ad hoc polymorphism. Section 6 presents the synthesis of the above arguments, assembling the full composite model. We draw some conclusions on the future directions for strongly-typed object-oriented languages in section 7.

2. CLASSES, TYPES AND ABSTRACTION

The observation that most class-based OOP languages fail to conform to more mathematical treatments of types has led some to suggest that objects might have 'class' and 'type' independently [6]. This view, which is interesting, defines a universe of mappings from class hierarchies into type hierarchies. The mappings are many-to-one, corresponding to the notion that several different class structures might satisfy the same abstract specification. One demonstration of this is the choice facing the programmer whether to provide a new class by subclassing (extending previous classes by inheritance) or alternatively by composition (constructing a class from several component classes), the outcome of which results in markedly different inheritance graphs. In such a case, the resulting class may end up with the same external interface (defined by the protocols for its methods) no matter what the implementation. Generally, groups of classes acting in concert (which we might compare with the notions of a *cluster* [7] in CLU or *system* in Eiffel) may provide the same facilities, as a group, while factoring out behaviours in completely different ways among themselves. It is therefore possible, in the general case, for relationships between individual classes and abstract types to be non-homomorphic. This view treats the class as a module of implementational convenience; which though formally lax does have advantages. The decoupling of class from type enables, among other things, a maximum reuse of implementation through inheritance, at the expense of creating some idiosyncratic abstractions. In most OOP systems, this economises on levels of indirection in structures and levels of nesting in remote calls.

We shall develop a different view here, which attempts to relate formally the notions of class and type, while preserving the possibility of differences in implementation. We appeal to the philosophical argument, in the first instance, that classes and types aspire to the same goal, namely abstraction. Any move away from this position seems to frustrate the aims of the HCI and AI schools in OOP, not to mention the security of the software engineers. Smalltalk would appear to have advocated the 'implementable abstraction' to a generation of programmers more accustomed to arrays, variable counters and (just emerging) record types. Flavors' [8] introduction of multiple inheritance provided much stronger support for the factorisation of common behaviour of objects; indeed one could argue that concept differentiation in AI is precisely the same task as that faced by designers of coerceable typing systems, a proposition recognised as important by the designers of CLOS [9] but not fully harmonised in its current specification.

A second philosophical argument is that type systems, as they are implemented in most strongly typed languages today, are far from complete in that they seldom venture to express relationships between general, abstract types. To draw an ethical analogy, these languages are not so much pure as fortunate that they have not been tempted. The Algol-68 experiment [10] in implicit type coercion was a notable

exception, but proved to be theoretically so difficult that subsequent languages ceded this hard-won territory. They cannot claim to hold the high moral ground; they have hardly entered the battle yet. We are convinced that any language that seeks to implement a truly general system of abstract types will face exactly the same difficulties encountered in class factorisation in OOP.

The real enemy in OOP is the kind of salesmanship that recommends the solution to a new problem as 'simply' the extension of existing classes. The argument of convenience is so beguiling that it masks the genuine difficulty in deciding how to provide a certain package of functionality in OOP. Indeed, it runs the risk (with apologies to Cox [11], who meant this in a different sense) of being truly an evolutionary approach to the development of software systems. Evolution implies growth and adaptation under highly localised constraints, no sense of destiny (unlike Dawkins' [12] biomorphs whose genetic pool is interpreted by design and whose survival is determined by long-term goals) and no sense of posterity. Anyone who, like the authors, have progressed to depth five or more in the implementation of a hierarchical class library, only to find that the previous 'simple' extensions need to be reconfigured totally from the root to receive a new application, will appreciate that there is no substitute for good, abstract design. Classes are things that should be wrestled with, argued over, survivors of the most rigorous scrutiny before they are accepted into libraries. Like all good ideas that survive the test of time, they are painful to come by and leave many casualties on the way. This is where OOP's strength in rapid prototyping really wins out; the point is never emphasised enough that most of the current adaptations are meant to die out.

3. TYPES, SETS AND AXIOMS

Types may be considered minimally as schemes for interpreting bit strings and their behaviour in a machine. Some argue strongly [13] for their separate existence apart from computation, a view which we can only endorse, since it supports our view that natural classificatory activity is related to typing. Reynold's argument here is that types are not constructed on sets of values, which would tie them too specifically to one domain in computation; rather *they are syntactic disciplines for enforcing levels of abstraction*. The same argument was made earlier by Morris [14], who showed that in principle a type can be represented, or implemented by a variety of sets; in this sense a set of values is insufficient to determine a type.

The designers of the language Russell [15,16] favoured types as values in a different sense, namely as elements of a universal Scott domain. This well-known construct [17] for avoiding Russell's paradox permits the admission of recursion, the admission of data types themselves as values in the same value-space through the notion of a retract, a function $f : D \rightarrow D$ such that $f = f.f$ and, by using retracts as the single operation for interpreting values in the universal space, a final algebra semantics for the typed lambda calculus. This was elegant and convenient, both for solving the typing problems associated with recursion and providing by the same token a rationale for treating the types so constructed as values in the semantics of the language, a feature upon which they constructed a model of polymorphism.

Russell treats a data type as *a set of operations specifying an interpretation of values of a universal value space.* This definition describes in more detail what it means to be an abstraction, drawing heavily on the algebraic specification approach of Guttag [18,19] which is still generally practised. By appealing to the retract again, the 'types as sets of operations' view can be shown to map onto the 'types as sets of values' view, thereby demonstrating the greater generality of the former view. This can be exemplified by the following inadequate definition of a simple ordered type by the supposed enumeration of the values which are members of the type:

SimpleOrdinal = {0, 1, 2, 3 ... }.

Another perfectly legitimate (and still indadequate) attempt at a description of the simple ordinal type might be:

SimpleOrdinal = {a, b, c, d ... }.

More precisely, there is no such thing as *the* set of simple ordinals; rather the type *SimpleOrdinal* denotes an abstraction that can be realised by a variety of carrier sets (cf Reynolds). The more common approach is to define *SimpleOrdinal* as the abstract type over which the operations *First()* and *Succ()* are meaningfully applied:

SimpleOrdinal = ord . { First : → ord;
Succ : ord → ord ... }.

This definition is both more general and more precise than our previous two attempts. Note that we are using a notational construction to describe the apparent recursive definition of the type *SimpleOrdinal*. The token *ord* is a placeholder for the type, awaiting its full definition. Such definitions are usually regarded as being existentially quantified. It would therefore not matter what we called this token, since the meaning of the type would be the same.

It is commonly accepted that this kind of semantics for abstract data types is still insufficient. Consider the following applications of operations to instances of the type:

Succ(1) → 1
Succ(b) → a

which still yield valid results according to the function signatures. The relationships that hold between the instances of type *SimpleOrdinal* are inadequately captured. For this, logical axioms are necessary, so that we could write assertions such as:

$$\forall x.\text{SimpleOrdinal}(x) \Leftrightarrow (\text{Succ}(x) \neq x) \Lambda (\text{Succ}(x) \neq \text{First}()).$$

This axiomatisation treats the type as a predicate over its instances, where the type is further constrained by the relationships that hold between (some of) its member functions. We shall therefore treat the data type as *a set of operations, that are constrained by a set of axioms.* Both the functional-algebraic and logical-axiomatic parts are necessary to complete the specification of an abstract type.

Of course, in some contexts we do need to go further and require a concrete type rather than an abstract one: that is, one in which a

specific carrier set (viz a set of instances, as above) has been chosen, rather than any suitable carrier set being acceptable. Where concrete types are required, however, there will generally be a 1-1 mapping between the abstract types being specified and the particular concrete types implementing them (as above). Confusion is unlikely to arise, provided that we are consistent in working either in terms of the abstract types or in terms of the concrete ones.

4. INHERITANCE AND SUBTYPING

In treatments of inheritance, a contrast is usually drawn between *essential* and *incidental* inheritance [20], or *strict* and *non-strict* inheritance [21]. Whatever the nomenclature, the strong variety of inheritance implies at least a sharing of class specification, which we would want to include the class's functional interface and class axioms by which all subclasses should be bound. The weak variety of inheritance implies only implementation sharing (opportunistic reuse of code and declarations for storage allocation). The strong kind usually occurs with some implementation sharing, since most languages strive to map in a fairly straightforward way from abstract datatypes onto their concrete counterparts. Some writers prefer to segregate the two varieties, while others allow the stronger to subsume the weaker, to varying degrees.

Anyone who has wondered how this can affect system design should observe, in the geometrical shapes described in Figure 1, that the hierarchy on the left is motivated from the viewpoint of domain analysis, whereas the hierarchy on the right is the result of incremental software development. From the abstract viewpoint, rectangles and triangles are kinds of polygon. Functions defined for polygons, such as *translate*, *rotate*, or *reflect* are valid for all subtypes, in the sense that any instance of *TRIANGLE* could be substituted for any instance of *RECTANGLE* in programs using only *POLYGON*'s functions. In contrast, the concrete parameterisation of a rectangle by the two cartesian point objects that make up its opposing corners leads to a natural extension of three points for a triangle and *n* points for a n-vertex polygon. Readers who find the latter alarming should be aware of the widespread practice of this kind of extension in OOP (particularly Smalltalk).

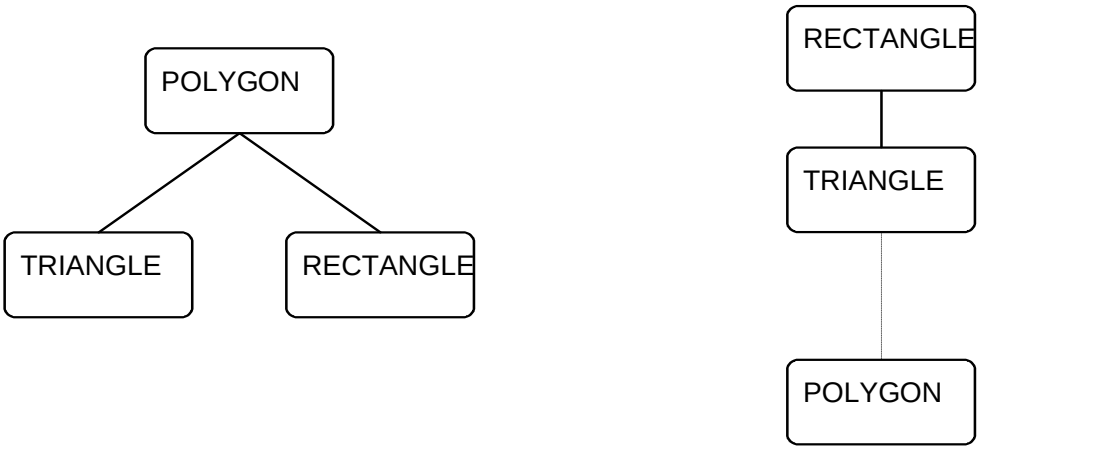

Figure 1: Contrasting Strong and Weak Inheritance

Such obvious and gross opportunistic development clearly invalidates any subclass-subtype homomorphism; however the real temptation to type-constrained OOP comes at a subtler level. Eiffel, for example, strives to maintain hierarchies of classes corresponding to efficient implementations of properly axiomatised abstract types. One such subtree reflects our preferred design for polygonal shapes from Figure 1. Eiffel also has orthogonal exporting and inheritance mechanisms. This means that a superclass may export a routine that is subsequently hidden from clients of its subclasses. One example is where the class *POLYGON*'s routine for adding to its vertices is subsequently hidden in *RECTANGLE* (which obviously cannot add to its vertices). The result is a kind of 'selective inheritance' of exported routines, which is clearly at variance with the notion of types being defined by their functions. This is not so much an instance of the recently popular 'type failure' problem [4], as an example of implementation creeping into abstract design. Eiffel's *POLYGON* tries too hard to be both an abstract class for all closed shapes and a means of implementing polygons with an arbitrary number of vertices. While such criticisms are easy to make, the issues are extremely difficult to resolve practically, since enforcing proper abstraction inevitably leads to a proliferation of classes.

We shall be making the argument that, where choices exist in how to provide a package of functionality in a given set of terminal classes, then a maximum priority should be given to proper analysis of the domain. This will typically result in a greater trend towards software reuse by composition and a reservation of inheritance to denote strict subtyping relationships. This trend can only be strengthened by an approach based on axiomatising class specifications, a theme to which we shall return. First, we investigate under what conditions inheritance may conform to subtyping.

Our intention is that the class-subclass relationship can be shown to correspond, under certain restrictions, to a type-subtype relationship. Cardelli and Wegner [22,23] have developed a simple model of classes-as-types, based on the notion of record subtyping. Later, we shall show that their scheme applies only to monomorphic types; or else to polymorphic but non-recursive types.

They, along with others [24,25], formally treat an object as a record whose components are functions representing methods. This model also provides a simple transparent interpretation of access to storage as the invocation of nullary functions delivering a result, ie:

```
CartesianPoint = { x : Integer; y : Integer }
```

is viewed as a record:

```
CartesianPoint = Rec {
        x : → Integer;
        y : → Integer
        }
```

where x and y are nullary functions. In this notation, reflecting the practice in OOP, the point-instance is not included as an argument to x and y since these functions implicitly return unique values for the object represented by the whole record.

In this simple algebra, the assignment of values to objects is side-stepped by treating all modifications as returning new records, thereby avoiding the issue of side effects:

CartesianPoint = Rec pnt . {
x : → Integer;
y : → Integer
moveBy : Integer x Integer → pnt;
equal : pnt → Boolean
}

Here, under the existential quantification of the token *pnt*, the *moveBy*-operation is viewed as returning a new object of the same type; the *equal* operation is defined to accept another argument of the same type.

Cardelli identified several important rules for record subtyping, three of which are summarised below. In this notation, 'A $\subseteq$ B' denotes an inclusive subtype relationship where A is the same type, or a proper subtype of B. The same semantics apply whether A is an immediate, or eventual subtype of B in a chain.

(1) Basic Record Subtyping Rule

$$\{ x_1{:}s_1, \ldots x_k{:}s_k, \ldots x_n{:}s_n \} \subseteq \{ x_1{:}s_1, \ldots x_k{:}s_k \}$$

This rule says that if a record has fields x_i in the types s_i, i = 1..n, then in particular it has fields x_i in the types s_i, i = 1..k. Therefore any operation that can be meaningfully applied to records of type

$$\{ x_1{:}s_1, \ldots x_k{:}s_k \}$$

may also be meaningfully applied to records of type

$$\{ x_1{:}s_1, \ldots x_n{:}s_n \}.$$

In consequence, a record subtype may add monotonically to the fields (or functions) provided by its supertype; ie it may never delete any. This rule describes specialisation by extension; but fails to account for the possibility of overriding inherited functions with more specific versions.

(2) Covariance Rule for Function Result Types

$$\frac{s_1 \subseteq t_1, \ldots s_k \subseteq t_k}{\{ x_1{:}s_1, \ldots x_k{:}s_k \} \subseteq \{ x_1{:}t_1, \ldots x_k{:}t_k \}}$$

This rule says that subtyping is preserved between records whose field types enter into a subtype relationship. If fields are viewed as nullary functions, then this rule applies to the result type of the functions. This means that a record subtype may specialise monotonically in the

result types of its functions; ie it may never generalise in these, with respect to its supertype.

Rules 1 and 2 may be combined by imposing rule 2's restrictions on the first k fields of an extended subtype record $\{ x_1{:}s_1, \ldots x_k{:}s_k, \ldots x_n{:}s_n \}$.

(3) Contravariance Rule for Function Argument Types

$$\frac{s' \subseteq s \quad t \subseteq t'}{s \to t \ \subseteq \ s' \to t'}$$

The rule for function subtyping [26,27] says that for a function $s \to t$ to be a subtype of another function $s' \to t'$, then the result types must obey the covariant rule (as above) but the argument types must be contravariant. This is indicated in the transposition of notation in the hypothesis $s' \subseteq s$ with respect to the terms in the rest of the rule. This means that a record subtype may generalise monotonically in the argument types of its functions; ie it may never specialise in these, with respect to its supertype.

So far, we have distinguished types only by their function signatures. As we observed above, this fails to capture the full semantics of types. America [28] notes, for example, that signature information alone is not sufficient to distinguish a *STACK* from a *QUEUE*. Axioms are needed to describe the LIFO property of *STACK*s and the FIFO property of *QUEUE*s with respect to their otherwise identical *push* and *pop* signatures. From the observation that an object can be considered as a machine with a state and behaviours (Meyer [3]), the role of axioms is to ensure the integrity of an object's state and the correct functioning of its behaviours.

A consensus view is that axioms may be classified under the headings *data type invariants* (Hoare [29]), *pre-conditions* and *post-conditions* (Jones [30]). Invariants describe the permanent semantic properties of a type, such as the intrinsic ordering of a sorted list. Post-conditions describe time-varying semantic properties of a type and ensure the correctness of an object's state after the execution of one of its behaviours, such as the presence of an element in a set after it has been added. Pre-conditions are a consequence of partial functions which cannot be applied to all members of the domain, such as empty stacks to which the *pop* and *top* functions may not strictly be applied.

Finding ourselves in agreement with America and Meyer, we include the following subtyping rules for axioms, which mimic in their form the rules given above for functions. In this notation, '$A \Rightarrow B$' denotes an entailment relationship such that satisfying axiom A automatically entails the satisfaction of axiom B. B may either be equal to A, or some less stringent condition whose satisfaction is entailed by that of A.

(4) Basic Axiom Subtyping Rule

$$\{a_1, \ldots a_k, \ldots a_n\}_{inv} \subseteq \{a_1, \ldots a_k\}_{inv}$$

This rule says that if a type satisfies axioms a_i, i = 1..n, then in particular it satisfies axioms a_i, i = 1..k. Therefore any type whose

invariant properties satisfy all n axioms will necessarily satisfy the first k axioms. This means that a subtype may add monotonically to the set of invariant properties of its supertype; ie it may never remove axioms. By the same reasoning, a subtype may add monotonically to the set of postconditions on functions inherited from its supertype.

(5) Covariant Entailment Rule for Invariants and Postconditions

$$\frac{b_1 \Rightarrow a_1, \ldots b_n \Rightarrow a_n}{\{b_1, \ldots b_n\}_{post} \subseteq \{a_1, \ldots a_n\}_{post}}$$

This rule says that subtyping is preserved between types whose invariant and variant semantic properties enter into an entailment relationship. If the satisfaction of each axiom a_i is entailed by the satisfaction of the corresponding b_i then the subtype satisfies at least all the axioms of its supertype. This means that a record subtype's invariants and postconditions may become monotonically stricter; ie they may never become less strict than those declared in the supertype.

Rules 4 and 5 may be combined by imposing rule 5's entailment restrictions on the first k axioms of an extended set $\{a_1, \ldots a_k, \ldots a_n\}_{post}$.

(6) Contravariant Entailment Rule for Preconditions

$$\frac{b_1 \Leftarrow a_1, \ldots b_n \Leftarrow a_n}{\{b_1, \ldots b_n\}_{pre} \subseteq \{a_1, \ldots a_n\}_{pre}}$$

This rule says that subtyping is preserved between types whose pre-conditions enter into a contravariant entailment relationship. This is indicated in the reversal of the entailment symbol '$\Leftarrow$' with respect to the subtype symbol '$\subseteq$'. This means that a record subtype's preconditions may become monotonically less strict; ie they may never be stricter than those declared in the supertype.

These rules define a space of possible treatments of inheritance which may be seen to preserve strict notions of type in the language concerned. At the most stringent end, a language might be so committed to the uniqueness of types that it forbids inheritance altogether, as in CLU (Liskov [7]). Inheritance is such a powerful notion, however, that abandoning it seems a waste of expressive power, a failure of nerve. Inheritance is now correctly recognised as the single unique feature of all truly object-oriented languages [31]; CLU might be described more accurately as a 'class-based' language [32,33]. The greatest single formal advantage inheritance offers to programming languages is the ability to represent relationships between generalised, abstract types.

A second possibility is to permit only the inheritance of abstract specifications. Emerald [34] is one such language in which implementation reuse is by composition. Emerald supports an attractive intuitional model where the specification of a composite class declares the types of component necessary to complete it. Type conformity among alternative component implementations is supported.

A third possibility is to permit implementation sharing where this does not invalidate type conformity. Such an approach maximises the benefits of the 'open-closed' principle (Meyer [3]) whereby completed classes are nonetheless open to further modification through subclassing.

The degree to which an object-oriented language treats inheritance as a kind of subtyping relationship corresponds to the progressive introduction of the rules described above. The first rule describes those languages that forbid explicit overriding but permit extensions. The second rule describes those languages which allow some redefinition of attribute types and function results according to a restricted scheme, such as Eiffel. A language which obeys the third rule is Trellis/Owl [35]; many languages, including Eiffel, break it. This third result is counter-intuitive for OOP: it states that methods which take additional formal arguments must generalise, rather than specialise the typing of these when they are redefined, if subtyping is to be preserved in the subclass. Such a facility is almost useless in practice.

Eiffel is one of the few languages which gives first-class status to axioms. It treats them as executable specifications checked at run-time. Eiffel obeys our rules 4 to 6, justifying the covariant and contravariant entailment rules in terms of Meyer's 'programming by contract' metaphor. All services that are guaranteed by one class must also be guaranteed by its descendent classes, therefore the post-condition on which the success of a service depends must not be weaker, but may be stronger. This is analogous to a supplier agreeing to deliver a product to a better specification than was demanded by the client. On the other hand, all messages which one class understands must also be understood by its descendants, therefore the pre-condition on which acceptance of a message is contingent must not be stronger, but may be weaker. This is analogous to a client agreeing to accept terms less strict than those originally drawn up with the supplier.

So far, we have only covered the relationship between inheritance and subtyping for simple monomorphic record types. The model described above is still inadequate to account for the operational behaviour of object-oriented languages in the context of polymorphism. We address this issue later.

5. CLASSICAL AND OBJECT-ORIENTED APPROACHES TO POLYMORPHISM

Polymorphism is a mechanism for expressing type compatibility. The earliest references to the term come in the theoretical writings of Strachey [36]. Here and in his subsequent work [37,38] he recognises two kinds of polymorphism which he calls *parametric polymorphism* and *ad hoc polymorphism*. Most of the subsequent research into polymorphic types has stemmed from Strachey's insight and uses his terminology, although we suspect that different authors take slightly different interpretations of Strachey's original observations. We shall, no doubt, do likewise below.

In Strachey's terms, a *parametric polymorphic* function is one that, upon different occasions, accepts arguments of different types and behaves in a semantically uniform way on each call, notwithstanding the type of the arguments supplied. The idea is that some functions perform identical, generic operations, no matter what the type of their argument(s). This kind of polymorphism is known as *genericity* in languages like Ada and Eiffel.

By contrast, an *ad hoc polymorphic* function is one that, upon different occasions, accepts arguments of different types and may behave in a semantically non-uniform way on each call, depending on the type of the arguments supplied. The idea is that some function names may be used to invoke different operations, appropriate to the type of their argument(s). This kind of polymorphism is widely known as *overloading*, from the ability to overload names with multiple meanings.

We are insisting here on the notion of semantic uniformity as the prime criterion for distinguishing between the two kinds of polymorphism. Reynolds ([13], p 519) only appears to be taking the most extreme interpretation of Strachey when he declares that

> 'a parametric polymorphic function behaves in *the same way for all types*, while an ad hoc polymorphic function may have unrelated meanings for different types.'

The notion in question is what is meant by 'all types'. We can only conceive of a few programming operations to which this literally applies: for example, declaration of, assignment to and obtaining the address of a static variable; or any operation that treats with a datum as though it were merely a bit-string in computer memory.

If we take the usual interpretation of Strachey's definition, 'all types' refers to the arbitrary typing of some primitive element out of which a new, generic type is constructed. Parametric polymorphic functions may then be defined for the generic type which behave in the same way *regardless of the type(s) out of which it is constructed*. The ubiquitous example cited here is the notion of a *Stack*. Whether the *Stack* contains integers, reals, characters or any other arbitrary type of element, the operations *push*, *pop*, *clear* and *top* have the same semantics; but these operations may of course only be applied to objects that conform to the generic type *Stack*.

Tennent [39] developed the notion of a formal type parameter (from which, intuitively, we get the term *parametric*), starting from a set of principles for the design of programming languages. These principles include *procedural abstraction*, *completeness* and *correspondence*. Procedural abstraction demands that any sequence of in-line statements can be abstracted over and made into a procedure, called from any part of the program. Completeness demands that all data types are first-class citizens (Strachey's term) in a language. Correspondence demands that any in-line declaration can be parameterised with no loss of uniformity or expressive power.

It is a short step from here to propose that types themselves can be abstracted over and parameterised. An even simpler example, using Eiffel syntax, illustrates this:

```
class ARRAY [T] export
        lower, size, upper, item, put ...
feature
        ...
        item (i : INTEGER) : T is
                -- value at array index i
        ...
        put (value : T;  i : INTEGER) is
                -- assign value to array at index i
        ...
end     -- class ARRAY [T]
```

The element-type of the array has been abstracted over and parameterised. It is represented by the formal type parameter *T*. Any type which eventually instantiates the parameter *T* must be the same type for all *T* in the declaration. This means that a static type checker could ensure that a call of the form:

```
        j : INTEGER;
        string1 : ARRAY [CHARACTER];
        ...

        string1.put (j, 3);
        ...
```

would be flagged as an error at compile time, since it attempts to instantiate *T* simultaneously with the types *INTEGER* and *CHARACTER*.

We prefer to view generic types like *ARRAY [T]* as unary type constructors, rather than actual types. This is exactly how such a declaration would be treated in Milner's approach to polymorphism [40] adopted by Burstall et al for the language Hope [41]. A Hope polymorphic function declaration such as

```
        map : (alpha → alpha) # list alpha → list alpha;
```

is considered to contain incomplete type information; the label *alpha* is a syntactic entity abstracting over all types in a static fashion. The *map* function is not fully specified until the variable parts are fully specified. This occurs when the compiler encounters a static call to the function, at which time the various arguments are examined and their intended use verified. This is done by propagating the types of the actual arguments into the function's expressions to check for consistency.

The strict form of parametric polymorphism insists that the type parameter can be satisfied by any actual type. This is seen as important to preserve the independence of the generic type's operations. Such a view rules out the possibility of constructions like *SORTED_LIST [Q ⊆ COMPARABLE]*, where *Q* is a bounded formal type parameter ranging over all those types which have an intrinsic ordering defined over them. We take the view that this generalisation is a valid one. We shall assume, for the moment, that bounded type variables are generally admissible.

Our argument here centres on two points. Firstly, the notion of independence is relative to the level of abstraction at which you view

the generic type. The closer you get to the implementation, the more dependent it is seen to be. For example, a declaration *ARRAY [T]* needs to know how large in bytes *T* is, in order to allocate arrays, or to compute offsets into arrays. Secondly, the general form of parametric polymorphism may be shown to preserve the important property of semantic uniformity. In our example, we can express this as a dependency relation of the form:

> The function *Insert* behaves in a semantically uniform manner over all possible instantiations of *SORTED_LIST [Q* $\subseteq$ *COMPARABLE]* iff all ordering functions *(such as* < and >) behave in a semantically uniform manner over all permissible instantiations of *Q* $\subseteq$ *COMPARABLE*.

By contrast, an overloaded function name may come to have unrelated meanings for different types. A function denoted by the plus-sign '+' might add integers and reals, perform OR on boolean values, compose functions, concatenate two strings and append two lists. There is no reason why these operations should be pairwise semantically related. The overloading rule in C++ [42] simply requires two static occurrences of an overloaded function name to be distinguishable on the basis of either the number, or types of arguments supplied. The compiler will ensure that the appropriate run-time code is inserted for those arguments and types.

We prefer to view overloading as defining a union of types. From this, we determine that overloading is undisciplined in two ways. Firstly, it allows the arbitrary extension of the union type. The union is extended every time a function, defined over a new type, is added to the overloaded name. The notion of an unbounded type seems at variance with the notion of typing as a classificatory activity. Secondly, there is no guarantee that semantic uniformity will be preserved pairwise across the set of functions denoted by the overloaded name. Addition and appending clash on at least one axiom, namely the commutativity of arithmetic.

These properties have led many to ban any kind of overloading from a type-consistent treatment of polymorphism. Unrestricted overloading is clearly too powerful a mechanism. However, if we examine the use to which overloading is put in computer languages in general, we often find a surprising degree of semantic consistency within overloaded names. This is in large part due to the designer's intuitions, since it is not enforced by the language.

Overloaded names may happen to contain subsets of functions for which a common semantics may be constructed. In our example above, we might define axioms for addition, common to reals and integers, calling these the properties of the polymorphic type *NUMBER*; equally we might define axioms for appending, common to lists and strings, calling these properties of the polymorphic type *SEQUENCE*.

We wish to abstract out and constrain the polymorphic type of object over which some commonality in names and behaviour is defined. In this respect, we share the goals of the designers of Russell [15, 16], Alphard [43], Euclid [44] and CLU [7]. Accordingly, we require a bounded polymorphic type which enforces a level of abstraction over the common features of its subtypes and provides for them a

restricted type-compatibility. We permit *REAL* and *INTEGER* to be subtypes of polymorphic *NUMBER*; likewise *LIST* and *STRING* are subtypes of polymorphic *SEQUENCE* (cf Common Lisp). Of course, we have not yet determined what kind of mathematical modelling would be adequate to capture the semantics of such a polymorphic type; at this point we are merely postulating that such types exist.

In some cases, successful type factorisation will reduce the need for overloading. For example, a set of comparison functions defining a *SIMPLE_ORDINAL* polymorphic type might be implemented over binary values one byte long and therefore be directly applicable to instances of *CHARACTER* and *SMALL_INTEGER*. Generally though, bounded polymorphic types will only be able to express the functional-algebraic interface and axiomatic specification for operations. In such a scheme, the concrete types *REAL* and *INTEGER* will almost certainly have separate, overloaded functions for addition; likewise the concrete types *STRING* and *LIST* for appending, as an almost inevitable consequence of their implementation differences.

A bounded polymorphic type describes behaviour that has a common semantics for all its subtypes. This provides a rationale for permitting some overloading in a type-consistent treatment of polymorphism. Any subtype is free to re-implement features, *provided that it satisfies the functional and axiomatic algebra for subtyping.*

Our discussion begs the question whether the kind of type-consistent polymorphism we are proposing for object-oriented languages falls into the category of *parametric* or *ad-hoc*, in Strachey's terms. In fact, it is something different again, but which bears similarities (on occasion) with both of these views. We find it helpful here to consider separately the mechanisms for implementing polymorphism in a programming language and the mathematical machinery necessary to describe such polymorphism formally.

From the implementation viewpoint, the distinction between parametric and ad-hoc is often signalled by the ability to define literally one generic function operating on a family of related types, as against a need to overload the function name with several different definitions. A true generic function accesses only that part of a data structure that is common to the entire family of types. Thus, the parametric polymorphism in typed functional languages based on a common list-cell representation for all kinds of typed lists bears a strong similarity with the inheritance-bounded polymorphism in object-oriented languages based on the shared record-structure included in all descendants. Overloading, by contrast, is seen as a necessary consequence of data types having a mutually exclusive representation.

However, concentrating on the mechanisms whereby polymorphism is provided in a language can sometimes cloud the issue. For example, Ada's generic packages must be explicitly instantiated before use and a compiler may therefore duplicate all generic functions for each separate instantiation in the target code. In consequence, genericity is provided by a kind of implicit overloading: generic functions do not really exist since they are expanded by the compiler.

Considering polymorphism from a more mathematical viewpoint, the only important concern is the notion of abstraction over types. If we

accept that inheritance-bounded polymorphism (sometimes known as *inclusion polymorphism*, a phrase coined by Cardelli and Wegner in [23]) follows some kind of systematic pattern of relationships between types, then we would expect to be able to express this using one or more type parameters. A key goal is to uncover the role played by type parameters in the object-oriented approach.

In the traditional approach to parametric polymorphism, a polymorphic type is declared by specifying the common parts of the structure and abstracting over the differences that remain (such as the element-type in a list or array). In the object-oriented approach, the opposite is true. A class defines a space of possible types bounded by the inheritance hierarchy and is, in this sense, an incomplete type specification. A class abstracts over the shared specification of all its potential descendants. Several contrasts with the traditional form of parametric polymorphism may be observed.

If we consider that a polymorphic type represents an incomplete specification of a family of types sharing some structure and behaviour, we may note firstly that the type parameter ranges over the unknown, disjoint parts of structure in the traditional form; whereas it ranges over the known, common parts of structure in the object-oriented form. Secondly, the type parameter is typically unconstrained in the traditional form; whereas it is necessarily bound by a partial specification in the object-oriented form. Finally, the parameter is explicit in the traditional form, whereas it is usually implicit in the object-oriented form. We might also add that the free generic type parameters in typed functional languages, Ada and Eiffel exploit more the idea of structural equivalence, an implementation concern, whereas the bounds imposed by inheritance in the object-oriented languages exploit more the idea of behavioural similarity, indicating a predominant concern with specification.

We may now define a space of languages and operations by their incorporation of some or all of five different kinds of polymorphism. We shall refer to these kinds of polymorphism more from a mathematical modelling perspective than according to the ways in which they are commonly constructed in programming languages:

(1) Universal Polymorphism: is exhibited by type-free languages and those operations that are applicable literally to any type (such as declaration and assignment). Universal type parameters may range freely over all types; and for each type in question the type parameter abstracts over the whole extent of the type.

Our use of the term *universal* is distinct from the sense in which it is employed by Cardelli and Wegner (op cit) where they use it to classify traditional parametric and object-oriented inclusion polymorphism apart from the more ad-hoc varieties of name overloading and type coercion. Their choice of term is motivated by the idea that the set of types which may eventually instantiate the parameter is potentially infinite. We agree with this assertion, yet prefer to use *universal* to denote a type parameter which ranges freely (unlike inclusion) over the whole extent (unlike parametric) of the abstracted types.

(2) Bounded Polymorphism: is a generalisation of the above, which defines bounded type parameters ranging over some set of types which share a subset of their abstract specification. Again, the type

parameter ranges over the whole extent of each type in the family of types; but unlike the universal parameter postulated above, it may only be instantiated by types satisfying certain constraints (the *bound* in question).

This is the kind of polymorphism exhibited originally by the language Russell [15, 16] whose explicit type parameters could only be satisfied by types meeting the constraints imposed by a set of function signatures. Russel's approach went hand-in-hand with a view of types as values which could be inspected like any other value. Strictly, the run-time inspection of types is only necessary in languages which permit dynamic binding of functions to arguments.

Object-oriented languages exhibit this kind of polymorphism operationally through the inheritance of common operations; or else through the planned overloading of a family of functions which satisfy a common set of axioms, usually because they descend from a common abstract or *virtual* function prototype. The type parameter is usually implicit, ranging over the full extent of each type in some family of types. The bound on the parameter is expressed as a node in a type hierarchy, whose purpose as an abstract type is to confer its specification upon all its descendent nodes. As a result, this kind of polymorphism has also been referred to as *inclusion polymorphism* since it may yet bear a direct relationship with inclusion in the single-sorted algebras [47].

Again, we use the term *bounded* in a slightly different sense from that found in Cardelli and Wegner, where the term is used to describe the constraints on a particular type capable of receiving objects of more than one type. We assert that the bounds apply to a type parameter (ie a higher-order entity) whose self-referential and recursive structure must also be preserved when it is instantiated by actual types.

Finally, our term *bounded polymorphism* is more general than *inclusion*, in that it may be used to describe bounded parameters in languages without inheritance and which therefore do not (necessarily) support systematic sets of relationships among partially-described types.

(3) Universal Construction: is the creation of type constructors whose element type(s) are parameterised. Type parameters therefore abstract over only a part of the extent of the whole constructor concerned, but are allowed to range freely over all types as candidates to complete this part. This is what other authors have traditionally referred to as *parametric polymorphism*. Such a scheme is provided in strongly typed functional languages such as Hope, Standard ML and Miranda. This kind of polymorphism is known as *genericity* in Ada and Eiffel, where the type-constructor is known as a *generic type*. The term *generic* arises from the fact that the type-constructor may be used to create a family of semantically related types; the complete family is constructed as the result of instantiating the formal parameter(s) by all possible actual types.

As we have observed above, the existence of the universal parameter is only possible because certain properties are required to be possessed by all types. This aspect is usually implicit, although it is expressed explicitly in Ada in the notion of attributes for types.

The combination of universal construction and bounded polymorphism may lead to interesting complexities. In a strongly typed functional language which only has the former variety, the instantiation of the parameter in *LIST [T]* by an actual type *P* typically restricts the elements of such a list exclusively to objects specifically of type *P*. In strongly typed object-oriented languages that have both varieties of polymorphism, the instantiation of the parameter is considered to impose a further bound on the list element-type such that it is assignment-compatible with all those objects whose type is *P* or a proper subtype of *P*. Among other things, this allows the construction of heterogenous lists, whose elements may arbitrarily belong to different subtypes of *P*, since they all satisfy the bound imposed by *P*. We believe that this combination therefore is the formal equivalent of bounded construction, described below.

(4) Bounded Construction: is a generalisation of the above, which defines type constructors whose element types are parameterised in a restricted way, such that all those types satisfying the bounded parameter share a subset of their abstract specification. This is the kind of polymorphism present in Alphard, CLU and all languages committed to a view of type-identifiers as syntactic entities rather than full values in the Russel sense. Bounded parametric polymorphism was introduced explicitly in the second major released version of Eiffel [45] where it is known as *constrained genericity*. Here, the family of types is constructed as a result of instantiating the formal parameter(s) by all those actual types that are subtypes of the bound.

Bounded construction is a generalisation in the sense that it permits the declaration of a larger set of type-constructors than universal construction, since the additional constructors may now be seen to depend on some property of their element-type(s), something which in theory was excluded before. Constructors such as *SORTED_LIST [T* $\subseteq$ *COMPARABLE]* are now possible, where the bound on *T* is expressed as another type whose operations any type instantiating *T* must at least supply. The instantiation of the parameter leads to homogenous lists of a constrained type in functional languages; and may still lead to heterogenous lists in object-oriented languages.

(5) Arbitrary Union: is the free union of types where these share no commonality that can be axiomatised; or share operations for which the functional or axiomatic specifications clash. This accounts for the remaining applications of overloading not covered under bounded polymorphism.

This kind of polymorphism may also cover certain styles of explicit coercions between types provided in some languages. Arbitrary coercion may, or may not, lead to sets of types being related which otherwise would have mutually exclusive semantic interpretations.

All kinds of polymorphism except the fifth variety may be incorporated in a type-consistent treatment of class inheritance. Any language only need implement varieties 2 and 4, since varieties 1 and 3 are limiting cases of these, respectively, where the bound is drawn at the root node of the type hierarchy in order to encompass all types.

This is still not the whole story, however. We shall want to examine further the role of the type parameter, especially where it abstracts over the whole extent of a type, in the context of inheritance and

polymorphism. In particular, we shall want to revise again the algebra devised in section 4 above to encompass parameterised recursive and re-entrant structures.

6. INTERACTIONS BETWEEN TYPES, INHERITANCE AND POLYMORPHISM

The first thing to consider in our composite model is the practical effect of requiring inheritance to conform to behavioural subtyping. What models of inheritance are sufficient to handle a fully axiomatised and abstract treatment of types?

If you consider the domain of abstract types as an infinitessimally divisible continuum reaching from the most general to the most specific, then particular types are represented by fixed points in this domain. A polymorphic type is not a fixed point, but rather a sub-space in this domain, bounded by an upper limit; this limit is a fixed point representing the most general type that can satisfy the bound. There is a type-compatibility relationship between polymorphic types and all fixed points falling within their bound. There is also a very similar relationship between abstract types and concrete types which implement them; but for the moment we will focus on abstract types.

Now, there are many possible selections of fixed points within the type domain which may prove to be of practical use. There is no requirement for any particular type to have a 'canonical' status or an *a priori* right to be represented, provided that the correct homomorphisms exist between the fixed points selected for use as specific types and those used to represent upper bounds on polymorphic types.

However, it is consistent with our approach to demand that any new functionality be introduced at single fixed points in the domain. If this were not so, the fixed points selected by the programmer would represent sub-optimal expressions of type for the task in hand. The consequence of this is that there should be no duplicate introduction of functions in an optimal, type-consistent inheritance graph. In an object-oriented language that adheres to this formal requirement, functions are typically introduced at one class in the hierarchy and are directly applicable to all eventual descendants. Where this is not physically possible, due perhaps to a mutually exclusive implementation among the descendent classes, then a full axiomatic specification for some set of overloaded functions may be introduced at a common (partly) abstract ancestor class.

Immediately, we can establish that this requirement cannot be met in languages that only have *single inheritance*, or a tree-like specialisation hierarchy. An example of this is exhibited in the *Collection* hierarchy in Smalltalk/V [46], in which the *add:* method is implemented variously for sets, bags, extrinsically ordered collections and intrinsically sorted collections. The common semantics of *add:* require the target collection to be extensible and to ensure that an element is present after it is added. However, the single inheritance scheme of Smalltalk/V chooses to partition the space of collections on the basis of whether the collection can contain multiple occurrences of an element, or whether the collection is indexable.

Smalltalk/V has no way to express the commonality of an extensible collection. In practice, Smalltalk/V defines a protocol for *add:* for all collections, but derails the application of this in the non-extensible classes (formally the equivalent of deleting a function). To obtain a distinction between fixed and extensible collections would require either a different partitioning of the hierarchy (with similar attendant problems), or an alternative inheritance scheme.

Multiple inheritance, or a graph-like specialisation hierarchy, resolves the problem due to partitioning, since it allows multiple abstract behaviours to be combined in descendent classes. It would be possible, for example, to define *ExtensibleCollection* and *UniqueCollection* as joint parents of *Set*. *IndexedCollection* and *FixedCollection* may likewise be joint parents of *Array*; and further pairwise combinations could give rise to unusual classes such as a *BoundedSet* or *ExtensibleArray*. Operationally, this is the kind of design approach recommended in Flavors, Eiffel and CLOS.

From a type-theoretic viewpoint, multiple inheritance of behaviours permits a finer-grained factorisation of notions of type. Minimally, the introduction of a single function or axiom would require a new, more specialised type; and this illustrates that our notion of type-space is quantal in functions and axioms. In practice, programmers select only a few of the many possible fixed points in the type domain, *revealing for the first time*, rather than *introducing*, sets of useful functions and axioms at these points.

Given a multiple inheritance scheme, it is desirable to define functions as generally as possible, so that these may apply polymorphically to all descendent classes. This practice is in fact enforced by our requirement to introduce novel packets of functionality strictly at single points in an optimal inheritance graph. Of course, we should like the polymorphic application of such generic functions to be type-correct, but we have not yet demonstrated that this is the case.

We need to examine the interaction between inheritance and polymorphism. As Canning, Cook and others have noted [47, 48] the straightforward Cardelli-Wegner record subtyping model does not quite encompass the operational behaviour of OOP languages in situations where functions are being invoked polymorphically. The simple algebra we have been using so far merely models *specific types* and their sub-types, rather than *generalised polymorphic types*. The effect of this is demonstrated in the following two examples (adapted from Canning).

Consider a simple, recursively defined type supplying an operation to move screen-relocatable objects:

Moveable = Rec mv . { move : Integer x Integer → mv }

Given that the types *Square* and *Triangle* can be derived as subtypes of *Moveable*, we should like *move* also to apply to these and return arguments of the correct types. However, interpreting the result as a *type-bounded variable* gives us the wrong answer: the function *move* applies to squares and triangles but always returns an object of the specific type *Moveable*. The algebra does not force the function's result type to mirror its polymorphic target. This problem is repeated

and subtly compounded when the recursively defined type appears on the left hand side of function signatures:

Comparable = Rec comp . { < : comp → Boolean }

Here, we should like *Integer* and *Character* to be subtypes of *Comparable*, inheriting the < operation. However, if our understanding is that < should always compare its target with an argument of the same specific type, this is not what we obtain under record subtyping:

Character = Rec char . { ...; < : Comparable → Boolean; ... }

which says that the inherited function compares with any type conforming to *Comparable*. Alternatively, if we were to override the inherited function deliberately with the one we really want, the subtyping relationship which establishes this:

{ ...; < : Character → Boolean; ... } ⊆ { < : Comparable → Boolean }

would in turn require *Comparable* ⊆ *Character* by the contravariance rule, which is precisely the opposite of what we intended. This manifestly prevents us from deriving *Character* ⊆ *Comparable*, unless in fact *Character* = *Comparable*. Canning et al. [47] refer to these cases as *positive recursion* and *negative recursion* in the definition of types. They show that Cardelli-Wegner bounded quantification strictly does not provide the same degree of flexibility in the presence of recursion as it does for non-recursive types.

Curiously enough, this problem is solved in the functional languages by treating all polymorphic references as *type parameters* (syntactic abbreviations for sets of actual types) bound throughout the expressions in which they appear. A simple denotational semantics can be constructed to handle ML-style polymorphism [49]. Of course, the functional languages do not require the kind of complex machinery needed in object-oriented languages for correct type inference under inheritance; this is because they do not generally make strong claims about systematic sets of relationships existing between types. However, it is intuitively apparent that some kind of subtyping relationship must exist between polymorphic type constructors and their family of instantiations.

The formal solution for object-oriented languages lies in treating class identifiers not literally as *bounded types* (declaring containers capable of receiving more than one kind of object), but as *type-bounded parameters* (syntactic abbreviations for some inclusive, ie non-disjoint, set of types). In fact, object-oriented languages must interpret class identifiers in two quite distinct senses; we shall return to this below.

Interpreting a class identifier as a *bounded parameter* has the following intuitive meaning: instead of denoting an actual, possibly quite general, type, it denotes a space of types satisfying a bound. This means that instead of treating a function like *move* as belonging to the type *Movable*, we need to treat it as belonging to some space of types constructed from *Moveable*, expressed using a parameter satisfying the bound imposed by *Movable*. Provided we can do this, objects of the actual type *Moveable*, or for that matter *Square* and

Triangle, can be shown to be type-compatible with the parameterised construction.

In object-oriented languages, there is a difficulty in constructing a quantification for type parameters which binds the type variable(s) both in the body of the type definition and in the expression denoting the type-bound itself. This problem is not apparent until you examine what should happen in the context of recursively defined types; and is precisely the situation uncovered by Canning et al. above. They call their solution *F-bounded quantification*, where the *F* stands for a typing function constraining the type parameter, *but which contains the parameter itself* (see Canning et al. [47] for a full derivation). The functional bound (F-bound) is constructed by working backwards from the desired conclusion - we desire a parameter t such that:

$$\forall t \subseteq F[t] \; . \; s(t)$$

where s stands for some set of function signatures describing the F-bound and which may include the parameter t. This permits the declaration of polymorphic typing functions of the form:

F-Moveable[t] = { move : Integer x Integer → t }

F-Comparable[t] = { < : t → Boolean }

which describe polymorphic type spaces rather than fixed-point types. The signatures of these typing functions now properly reflect the true underlying nature of polymorphic types as type-constructors, since they contain a parameter requiring instantiation.

Actual types are derived by the application of these polymorphic typing functions to specific types, during which process the type parameter is replaced. By this substitution, we can show that:

F-Moveable[Square] = { move : Integer x Integer → Square }

F-Moveable[Triangle] = { move : Integer x Integer → Triangle }

and if we wish to define the specific types *Square* and *Triangle* as having further unique behaviours in addition to being moveable:

Square = Rec sqr . { ...; move : Integer x Integer → sqr; ... }

Triangle = Rec tri . { ...; move : Integer x Integer → tri; ... }

we thereby demonstrate that:

Square ⊆ F-Moveable[Square],
Triangle ⊆ F-Moveable[Triangle] ... etc

Integer ⊆ F-Comparable[Integer],
Character ⊆ F-Comparable[Character] ... etc

In consequence, the application of *move* defined for *F-Moveable* is in fact type-correct for *Squares* and *Triangles*; also *Characters* or *Integers* may be compared pairwise using *F-Comparable's* function <. This construction solves the polymorphic typing problem for recursive types; it captures the notion of adding functions to a (polymorphic) type while preserving the recursive structure of the type for all instantiations of the type parameter. Canning et al note that F-bounded quantification is closely related to inclusion for single-sorted algebraic signatures.

7. CONCLUSION: CLASSES, TYPES, INHERITANCE AND POLYMORPHISM

We now come to the place where we have to decide what we understand by the notions of *class* and *inheritance*.

Earlier, we remarked on the view held by some that objects have *class* and *type* separately, if not independently. Although we do not support

this view, we agree that an informal and undisciplined use of classes presupposes such a view. Instead, we intend to bring the notions of *class* and *type* much closer together.

Attempts to describe the notion of *class* formally are frustrated by the conflicting requirements on the semantics of *class* obtained by inspecting the uses to which classes are commonly put in most OOP languages. Operationally, a class may serve three different purposes (Cox, [11]): to act as a template for creating the objects which are its instances; to act as a shared data structure storing values accessible by all its instances; and to provide the message-interface for all its instances. The first two roles concern the implementation of objects, the last concerns the specification of behaviour.

As we observed above, the behaviour of a type is defined by its functional and axiomatic specification. If we therefore treat external behaviour as the primary structuring mechanism for a space of classes, then we may show that classes conform in some way to types and inheritance relates more to behavioural subtyping than to the incremental extension of data storage cells. This last point is extrememly important - a class is not, first and foremost, a description of objects with identically-named attributes, rather it describes objects with similar behaviour. Given our argument so far, we cannot yet insist that classes *are* types; nor that inheritance *is* subtyping.

Our view of classes may seem to have a lot in common with the so-called abstract data types (ADTs) created in module-based languages such as Modula-2 and Ada. In these languages, great emphasis is laid on the external interfaces of ADTs, which describe (at least) the functional specification of types. However, we disagree in points of detail with the terminology and design approach taken in these languages. To clarify our use of terms, we shall note these points of disagreement.

Firstly, the word 'abstract' in the widespread use of the term *abstract data type* to denote a software component whose internal nature is partially concealed is strictly a misnomer. Such types are not abstract, they are in fact concrete since they are fully implemented. It would be better to refer to such constructions as *encapsulated data types*, where encapsulation includes both the notions of *information hiding* (the client programmer need not know about the internal details of a type's implementation) and *protection* (the client programmer may not tinker with the internal details of a type's implementation). In contrast, a properly abstract datatype provides a functional and axiomatic specification for that type's behaviour and absolutely no implementation.

Secondly, the mechanism governing the visibility of objects and operations in these languages relates not to types themselves, but to *modules* (in Modula-2; called *packages* in Ada). A module is simply a syntactic arrangement of datatype declarations and operations, without any obvious theoretical status. There is no requirement to devote a module to the implementation of a single encapsulated data type; in practice modules may contain several related data types. The fact that modules are in competition with types for control over levels of abstraction mitigates against a full and proper expression of type.

Thirdly, there is no serious attempt in these languages to establish systematic sets of relationships between types and generalised types. Ada has genericity and a complicated set of rules governing type coercion; Modula-2 renounces even the small amount of overloading available in Pascal. But otherwise there is no systematic notion of type compatibility, specialisation by extension, specialisation by restriction or progressive reification. These languages preclude the existence of homomorphisms between types and generalised types.

Needless to say, our understanding of the notion of *class* encompasses a spectrum from the fully abstract types to the fully concrete types, together with a systematic set of relationships among generalised polymorphic types constructed from actual types. Each type should have full and sole control over its own visibility. We cannot yet complete our definition of *class* until we have considered both *inheritance* and *polymorphism*.

Inheritance is usually taken to mean the same thing as *specialisation*. If *B* is a subclass of *A*, then *B* may specialise *A by extension* (it may add new functions) or *by restriction* (it may enforce stricter axioms, such as only permitting a subset of the original underlying value space); consequently *A* is said to *generalise B* in the sense that *A* subsumes *B*. A particular application of specialisation is reification, for example where an abstract type *PLANAR_POINT* offering specifications for the functions *{x, y, theta, rho}* is specialised by the alternative implementations given in *CARTESIAN_POINT* and *POLAR_POINT*. This gives us an operational definition of inheritance, but fails to explain what it is, in a formal sense.

The question remains: what are we to understand by the terms *specialisation* and *generalisation*? Have we simply replaced *inheritance* by another label? These terms arise from the AI community in connection with prototypes and cognitive theories of learning (chunking, hierarchical clustering). Often, *specialisation* is mistakenly compared with the stricter mathematical notion of *subtyping*; or else aspires to a 'semi-formal' version of the same thing. Fortunately, we are able to dispense with hopeful ignorance and construct a proper definition for specialisation, based on an understanding of polymorphic types. We shall summarise our argument so far.

Earlier, we experimented with a model of inheritance as record subtyping. This was done by gradually introducing constraints on what kinds of extensions and overriding were permissible in descendent classes, in order to ensure that they were still subtypes of their ancestors. We noted later that the relationships we had established were between fixed-point, monomorphic types. The record subtyping model did not extend sufficiently to cover either languages with polymorphism or languages with recursive types. Finally, we introduced a polymorphic typing function constructed around a parameter linked with an actual type, demonstrating that this produced type-consistent results under inheritance and polymorphism.

Having reached a similar point in the argument, other authors have stopped to consider how this discovery apparently drives a wedge between classes and types, reasoning in the following way: two types *A* and *B* may both satisfy the same F-bound, such that $A \subseteq F[A]$ and $B \subseteq F[B]$, and yet may not be in a subtype relation with each other -

neither $A \subseteq B$, nor $B \subseteq A$. This means that an F-bounded function may be applied to (ie inherited by) objects with incomparable types, showing that the inheritance hierarchy is distinct from the type hierarchy.

We think that this is looking in the wrong direction! Surely, the most important discovery here is that a systematic approach to constructing truly polymorphic types has been devised. For every *actual type,* at whatever level of generality, a corresponding *polymorphic type* may be constructed by abstracting over the body of this type using a parameter bounded by a function. Furthermore, there appears to be a simple inclusion relationship between polymorphic types (ie the spaces occupied by polymorphic types in the type domain contain other sub-spaces representing more constrained polymorphic types). This is indicated in that the sets of parameterised functions defining the related F-bounds turn out to be supersets and subsets of each other. Finally, a straightforward subtyping relationship exists between polymorphic types and all actual types that produce valid instantiations of the parameter.

To suggest therefore that inheritance is a weak notion because it does not conform to subtyping between simple fixed-point types is a nonsense. There was so little mileage to be had from that idea anyway; it would be difficult even to construct a type-correct polymorphic application of *plus* on a simple subrange of *Integer*, and still retain the subtype relationship. Or, to put it another way, the focus of a truly general theory of types is surely the construction of relationships between general spaces in the type domain and not the limited links that can be established between fixed points. Pursuing this prospect, Canning et al. relate F-bounded quantification to some family of F-coalgebras, using the Breazu-Tannen semantics-by-translation approach [50]. Observing that the recursive type *Rec t.F[t]* may be regarded as a particular F-coalgebra, they suggest that F-bounded polymorphism involves quantification over a category whose objects are properly regarded as generalisations of the recursive type *Rec t.F[t].*

Strict inheritance is therefore a much grander notion than some authors might lead you to believe - subtyping among parameterised type-constructors, interpreted formally as subsumption among categories. Inheritance is principally a relationship among type-constructors, extending finally to actual types.

As a consequence of this, we need to interpret the occurrence of class identifiers in object-oriented languages in two distinct senses. Typically, a class identifier may be used to denote a fixed-point type on the one hand; or a polymorphic type constructor on the other hand. The interpretation that we should choose is usually apparent from the context; however there are also some interesting issues that arise as we pursue this. In conclusion to our article, we shall discuss some of the implications of our approach to the interpretation of class identifiers.

Class hierarchies are always open-ended, subject to further subclassing. This means that any class description potentially denotes a bounded polymorphic type-space under the class concerned, for which the actual class is the most general fixed-point type satisfying the bound. Therefore all class identifiers used to type

function signatures should really be considered formally as *bounded parameters* to be instantiated later by actual types. An examination of the operational behaviour of current object-oriented languages allows us to infer that a single underlying parameter is what is usually intended. The instantiations of a single parameter t in:

F-Comparable[Character] = { < : Character → Boolean }

F-Comparable[Integer] = { < : Integer → Boolean }

allow us to compare like with like in a polymorphic way:

```
3 < 4;
'a' < 'd';
```

and this is usually the intention in object-oriented languages. However, a single underlying parameter does not admit of the possibility of mixed-type calls:

```
'a' < 102;
10 < 'c';
```

This is because a single parameter cannot be uniformly instantiated by a mix of types. To allow this, we would have to declare, in the polymorphic typing function, all of the parameters that were to be included in the F-bound:

F-Comparable[p,q] = { < : q → Boolean }

where *p* and *q* might be instantiated by distinct types both satisfying the F-bound. Of course, whether this is appropriate or desirable in a language will depend on the underlying implementation of distinct comparable types and may involve some

implicit coercions, or the application of a polymorphic hashing-function in every call to <.

Object-oriented programs manipulate objects created from specific type templates; so it is clear that the moment an object is created (using an expression such as 'new' or 'Create') then the polymorphic space denoted by the class is projected onto a fixed-point type, in fact the most general type satisfying the F-bound. In the vast majority of cases, this describes the required behaviour. A class could therefore be described as a polymorphic type with a default fixed-point instantiation.

However, there are a few cases where this projection does not account adequately for the operational behaviour of object-oriented languages. This is where the fixed-point type of the object to be created has to be determined dynamically, for example during a cloning or deep-copying operation. Typically, object-oriented languages have a special syntax to denote this case. In Smalltalk, you can reason about types at runtime:

```
theCopy := self class new.
```

whereas in Eiffel, there is a mechanism to anchor the type of the copy to the type of the current object:

```
theCopy : like Current;
theCopy.Create(...);
```

It turns out that this special syntax is necessary because of the default interpretation given to class identifiers used to type object-creation expressions. In our formal model, the dynamic nature of the result-type of a copying operation is apparent in the parameter:

$$\text{F-Copyable}[t] = \{ \text{copy} : \rightarrow t \}$$

Perhaps a better approach would be to admit explicit F-bounded parameters into the syntax of object-oriented languages, in order to make cases like this and the above example of mixed-type polymorphic calls less ambiguous. This is the challenge to the next generation of developers of object-oriented languages.

REFERENCES

[1] A Goldberg and D Robson (1983), *Smalltalk-80: The Language and its Implementation*, Addison Wesley.

[2] S E Keene (1989), *Object-Oriented Programming in Common Lisp*, Addison Wesley and Symbolics Press.

[3] B Meyer (1988), *Object-Oriented Software Construction*, Prentice Hall.

[4] W Cook (1989), 'A proposal for making Eiffel type-safe', *Proc. ECOOP-89*, 57-70. Reprinted in *Computer Journal 32(4)*, 305-311.

[5] B Meyer (1989), 'Static typing for Eiffel', *Internal Report, July*, Interactive Software Engineering Inc.

[6] A Snyder (1986), 'Encapsulation and inheritance in object-oriented programming languages', *Proc. ACM Conf. on OOPSLA-86*, 38-45.

[7] B H Liskov (1981), *CLU Reference Manual, Lecture Notes in Computer Science 114*, Springer Verlag.

[8] D A Moon (1986), 'Object-oriented programming with Flavors', *Proc. ACM conf. OOPSLA-86*, published as *SIGPLAN Notices, 21(11)*, November 1986, 9-16.

[9] D Bobrow, L DeMichiel, R Gabriel, G Kiczales, D Moon and S Keene (1988), *The Common Lisp Object System Specification: Chapters 1 and 2*, Technical Report 88-002R, ANSI X3J13 standards committee document.

[10] A van Wijngaarden, B Mailloux, J Peck, C Koster, M Sintzoff, C Lindsey L Meertens and R Fisker (1975), 'Revised report on the algorithmic language Algol 68', *Acta Informatica 5*, 1-236.

[11] B J Cox (1986), *Object-Oriented Programming - an Evolutionary Approach*, Addison Wesley.

[12] R Dawkins (1984), *The Blind Watchmaker*, Penguin.

[13] J C Reynolds (1983), 'Types, abstraction and parametric polymorphism', in: *Information Processing 83*, ed. R E A Mason, North-Holland, Amsterdam, 513-523.

[14] J H Morris (1973), 'Types are not sets', *Proc. ACM Symposium on Principles of Programming Languages*, Boston, 120-124.

[15] A Demers, J Donahue and G Skinner (1978), 'Data types as values: polymorpism, type-checking, encapsulation', *Proc. 5th ACM Symposium on Principles of Programming Languages*, 23-30.

[16] A J Demers and J E Donahue (1979), 'Revised report on Russell', *TR 79-389, Department of Computer Science, Cornell University*, Ithaca, NY.

[17] D S Scott (1976), 'Data types as lattices', *SIAM J. Computing, 5(3)*, 523-587.

[18] J W Guttag (1975), 'The specification and application to programming of abstract data types', *Technical Report CSRG-59, University of Toronto*, September.

[19] J W Guttag (1977), 'Abstract data types and the development of data structures', *Comm. ACM 20(6)*, 396-404.

[20] M Sakkinen (1989), 'Disciplined inheritance', *Proc. ECOOP-89*, 39-56.

[21] W W Y Pun and R L Winder (1989), 'A design method for object-oriented programming', *Proc. ECOOP-89*, 225-240.

[22] L Cardelli (1984), 'A semantics of multiple inheritance', in: *Semantics of Data Types, Lecture Notes in Computer Science 173*, Springer Verlag, 51-68.

[23] L Cardelli and P Wegner (1985), 'On understanding types, data abstraction and polymorphism', *ACM Computing Surveys 17(4)*, 471-522.

[24] W Cook (1989), *A Denotational Semantics of Inheritance*, PhD Dissertation, Brown University.

[25] P Canning, W Cook, W Hill and W Olthoff (1989), 'Interfaces for strongly typed object-oriented programming', *Proc. ACM Conf. on OOPSLA-89*, 457-467.

[26] J C Mitchell (1984), 'Coercion and type inference', in: *Proc. 11th ACM Symp. on Principles of Programming Languages*.

[27] L Cardelli (1988), 'Structural subtyping and the notion of power type', *Proc. 15th ACM Symp. on Principles of Programming Languages*, 70-79.

[28] P America (1987), 'Inheritance and subtyping in a parallel object-oriented language', *Proc. ECOOP-87*.

[29] C A R Hoare (1972), 'Proof of correctness of data representations', *Acta Informatica 1*, 271-281.

[30] C B Jones (1986), *Systematic Software Development using VDM*, Prentice Hall.

[31] B Stroustrup (1987), 'What is object-oriented programming?, *Proc. ECOOP-87*, pub. as *BIGRE 54*, 51-70.

[32] P Wegner (1987), 'The object-oriented classification paradigm', in: eds. B Shriver and P Wegner, *Research Directions in Object-Oriented Programming*, MIT Press.

[33] P Wegner (1987), 'Dimensions of object-based language design', *Proc. ACM Conf. on OOPSLA-87*, pub. as *ACM Sigplan Notices 22*, ed. N Meyrowitz, 168-182.

[34] R K Raj and H M Levy (1989), 'A compositional model for software reuse', *Proc. ECOOP-89*, 3-24. Reprinted in *Computer Journal 32(4)*, 312-322.

[35] C Shaffert, T Cooper, B Billis, M F Kilian and C Wilpolt (1986), 'An introduction to Trellis/Owl', *Proc. First ACM Conf. on Object-Oriented Programming Systems, Languages and Applications*, 9-16, Portland Oregon, September (1986).

[36] C Strachey (1967), *Fundamental Concepts of Programming Languages*, Oxford University Programming Research Group, (originally lecture notes, Int. Summer School in Comp. Prog., Copenhagen, August 1967).

[37] C Strachey (1973), *Varieties of Programming Languages*, Oxford University Programming Research Group.

[38] R Milne and C Strachey (1976), *A Theory of Programming Language Semantics*, Chapman and Hall.

[39] R D Tennent (1981), *Principles of Programing Languages*, Prentice Hall.

[40] R Milner (1978), 'A theory of type polymorphism in programming', *Journal of Computer and System Sciences 17*, 348-375.

[41] R M Burstall, D B MacQueen and D T Sanella (1980), 'Hope: an experimental applicative language', *CSR-62-80, Department of Computer Science*, University of Edinburgh.

[42] B Stroustrup (1986), *The C++ Programming Language*, Addision Wesley.

[43] W Wulf, R London and M Shaw (1976), 'Abstraction and verification in Alphard: introduction to language and methodology', *USC/ISI Research Report 76-46*, June.

[44] B Lampson, J Horning, R London, J Mitchell and G Popek (1977), 'Report on the programming language Euclid', *Sigplan Notices 12(2)*.

[45] B Meyer (1992), *Eiffel: The Language*, Prentice Hall, 200-204.

[46] Digitalk (1986), *Smalltalk/V - Tutorial and Programming Handbook*, Digitalk Inc.

[47] P Canning, W Cook, W Hill and W Olthoff (1989), 'F-bounded polymorphism for object-oriented programming', *Proc. 4th Int. Conf. on Functional Programming Languages and Computer Architecture*, Imperial College, 11-13 September, 273-280.

[48] W Cook and J Palsberg (1989), 'A denotational semantics of inheritance and its correctness', *Proc. ACM Conf. on OOPSLA-89*, 433-443.

[49] A Ohori (1989), 'A simple semantics for ML polymorphism', *Proc. 4th Int. Conf. on Functional Programming Languages and Computer Architecture*, Imperial College, 11-13 September, 281-292.

[50] V Breazu-Tannen, T Coquand, C Gunder and A Scedrov (1989), 'Inheritance and explicit coercion', *Proc. IEEE Symp. on Logic in Computer Science*, 112-133.