DIVISION OF COMPUTER SCIENCE

An Investigation of Types leading to an Examination of some aspects of F-bounded Interfaces and the Type Classes of Haskell

Technical Report No. 154

K. M. Buchanan
R. G. Dickerson

April 1993

An Investigation of Types leading to an Examination of some aspects of F-bounded Interfaces and the Type Classes of Haskell

Mary Buchanan and Bob Dickerson
School of Information Science, University of Hertfordshire
College Lane, Hatfield, Herts AL10 9AB

April 1993

Abstract

The treatment of types in programming and in type checking is considered briefly and types are distinguished from classes. The subtyping relation is discussed with particular emphasis on the consequences of combining the inheritance and subtyping hierarchies in object-oriented programming. Less restrictive type relationships for inheritance are discussed and it is concluded that there are advantages in separating implementation inheritance from subtyping. Further work is needed to establish the significance of such a separation throughout the system development process.

1 Introduction

Any discussion of object-oriented concepts inevitably involves the use of terms such as types, classes and inheritance. However, there are no universally accepted definitions of these terms and consequently it is essential to be clear on the interpretation intended in any particular context.

We discuss various interpretations given to types in order to illustrate these differences. Classes and inheritance are considered in the context of types. Some knowledge of object-oriented concepts and the use of types in programming languages is assumed.

2 Why are types useful?

If we consider untyped universes, for example λ-expressions in the λ-calculus or sets in set theory, types tend to arise informally as a means of categorising objects according to their usage and behaviour [1]. In λ-calculus, some functions are chosen to represent integers and others to represent booleans. In set theory, some sets are chosen to denote ordered pairs and some sets of ordered pairs are chosen to represent functions. However, without a type system, there is no means of enforcing any such type distinctions. A type system imposes constraints on the usage of types such that their correct usage can be enforced. Over and above this, type systems have sometimes been imposed in order to overcome logical inconsistencies. For example, typed set theory arose from the need to overcome inconsistencies such as Russell's paradox (described in [2]) in naïve set theory.

3 What is a type?

In general terms, we can consider a type as being a means of classifying values. The type to which a value belongs defines the set of contexts in which the value may be used correctly. The classification can be derived merely from the values themselves or from the values and the ways in which the values can be used.

At the very least, a type can be considered as a *Predicate Type* [3] which simply denotes a set of values. For example, the type Bool denoting the values {true, false}. In terms of many sorted algebra, a predicate type is a sort.

More often, a type is considered as a set of values together with operations on those values. An algebraic specification of an *Abstract Data Type declaration* [3] defines one or more predicate types with a collection of operations on the type. For example, the type *Nat* with the operation *add* defined as:

$$add : Nat \times Nat \rightarrow Nat$$

In many sorted algebra, an abstract data type declaration is a signature.

If we wish to provide a meaning for an abstract data type, then we can do so in terms of many sorted algebra by naming an algebra such that the operations are interpreted in terms of semantic constructs in the algebra. For example, the signature (Nat, {succ, add ...}) could map onto the algebra of natural numbers. This provides a denotational model for the signature though it should be borne in mind that this is not the only possible model.

3.1 Types in programming languages

Types are used in programming languages to help establish the correctness of source code. We can consider the types identified above in terms of a strongly typed language such as Modula-2 or Ada:

- A Predicate type = Modula-2 type or Ada type
- Abstract Data Type declaration = Modula-2 module definition or Ada package specification
- Abstract Data Type model = Modula-2 module implementation or Ada package body

The concept of hiding implementation, such that a type is used through operations which are defined in an interface, is widely adopted. We need to remember that it is possible in Modula-2 to show the type representation in the definition module; it is only by using opaque types that the type representation can be hidden in the implementation module. The operation implementations can always be hidden in the implementation module but if the type representation is declared in the definition module, there is nothing to prevent a user from accessing the type directly, rather than using the given operations.

A type can be completely hidden in Modula-2 such that not even its name appears in the definition module but this is restricted to a single instance of the type which is represented as a hidden variable in the implementation module. In general, however, programming languages implement abstract data types such that the abstract data type manipulates data but does not store data. This is consistent with the algebraic model of abstract data types (described above in section 3) which models the manipulation of data but not its storage. Such a model contrasts with the object-oriented programming approach in which objects store the data they manipulate.

3.2 In system development

Types can be used as a means of defining intuitive ideas about a system under development. At the analysis stage, types (sometimes called classes) can be identified such that values in the problem domain can be classified according to kind and behaviour. During design, extra types may well be added to incorporate hardware/software constraints and the types derived by analysis may be altered due to constraints within the implementation language.

Types can be used as a basis for the choice of modules in a structured framework. Information concerning which types communicate with which other types can be used to improve the efficiency of an implementation.

3.3 Type interfaces

A type interface, also called a protocol, specifies the functionality of the operations associated with the type. If such a specification does not declare the implementations of the operations, the interface provides a boundary between implementations of the type and users of the type. "Ideally, interfaces should contain a formal description of the behaviour of operations, for example as a logical theory" [4]. Without such formality, there is the risk that the intended purpose of the operations may be ambiguous.

A formal specification could be a signature but this is purely syntactic and gives no information about the intended behaviour of the operations. The operations *add* and *mult* have the same rank, that is they have the same number and sort of arguments and the same result sort:

$$add : Nat \times Nat \rightarrow Nat$$

$$mult : Nat \times Nat \rightarrow Nat$$

However, although the operations have the same signatures, they do not have the same semantics in that the intended behaviour of the operations is different. For interfaces to be used by clients, the semantics of the operations should be given. However, for type checking, syntactic information alone may be adequate.

We must consider whether, in any given situation, a type is being viewed as:

- just a signature
- a signature AND semantics
- a signature, semantics AND an implementation.

4 What is type checking?

Type checking is used to detect type errors which can arise from attempts to apply an invalid operation to a value. An operation is invalid if the actual parameter passed to the operation is not compatible with the formal parameter defined for the operation. Similarly, a value assigned to a variable must be type compatible with the variable type.

A programming language can restrict the ways in which a value can be bound to a name (or a variable or a parameter). In static type checking, the compiler checks that names are bound to values of the permitted types; this guarantees that operations will be applied to names which refer to values of the appropriate type. Dynamic type checking is required when names can be associated with values of more than one type and the actual type is not known until run-time.

Run-time type checking can be done manually by means of conditional (case) statements which are written by the programmer and which identify all the permitted types with which a name may be associated. Alternatively, run-time type checking can be done automatically by the programming language and run-time environment.

Instance variables (names) in all known object-oriented languages can be used to store references to objects. The instance variables are polymorphic if they can refer to instances of several different classes during the life of a program [5]. When a message is sent to such an instance, dynamic binding is used to invoke the particular implementation of the method which is appropriate to the class to which the instance refers at that time. In Smalltalk such polymorphism and dynamic binding are permitted without restraint, allowing any object to be assigned to any instance variable. There is therefore no way of the compiler checking whether an object will possess a method to correspond to the message sent to it. If there is no such method, dynamic type checking will detect this and a run-time error will occur.

To prevent this happening, instance variables can be constrained to accept only certain object assignments and static typing can then be used to ensure that the constraints are not violated. The assignments which an instance variable may accept can be determined by *conformance*. A type (or class) conforms to another if it can be used in all the contexts where the other is expected; that is, it can respond to all the messages handled by the other type. In theory the ability of a compiler to check conformance should depend only on the interface of a type/class and not on the implementation of the methods. However in practice what is actually checked by the compiler varies from language to language. In Eiffel classes have to have their implementations related by inheritance if they are to be type compatible.

5 Polymorphic types

Monomorphic type systems constrain values to belong to only one type and hence to exhibit only one type of behaviour. Polymorphic type systems permit values to belong to more than one type. Strachey subdivided polymorphism into universal and ad hoc polymorphism. Cardelli and Wegner [1] have elaborated the Strachey classification: universal polymorphism is subdivided into parametric and inclusion polymorphism and ad hoc polymorphism is subdivided into overloading and coercion. Coercions map one type to another before operations are applied whereas in overloading, different code is invoked for arguments of different types. In a parametric function the same code is invoked by arguments of different types which share a common structure; for example a list of integers has the same structure as a list of strings and operations such as *head* and *tail*

can be applied to either. Parametric polymorphism is also known as functional polymorphism [6].

Subtyping is a form of inclusion polymorphism in which a value of a subtype may be used in any context which requires a value of the supertype. The semantics of subtyping are based on the mathematics of partial orders; subtype relations are reflexive, transitive and antisymmetric [7]. Reflexivity means that all types conform to themselves. Transitivity means that if a type T1 is a subtype of type T2 (expressed T1 $\leq$ T2) and T2 $\leq$ T3, then T1 $\leq$ T3. Antisymmetry ensures that two types are equal if they are subtypes of each other.

For any particular type system involving subtypes, it will be necessary to have some means of establishing whether two types have a subtype relationship. The interpretation of integer subranges as subtypes of integers is conceptually simple. Cardelli and Wegner [1] have identified rules for record subtyping and for function subtyping. A record is a subtype of another if its fields (defined by label and type) are a superset of the other's and the types of the common fields are in a subtype relation. As an example [6, page 55] we consider the records defining a `vehicle_type` and a `car_type`

```
vehicle_type = <age:int; fuel:int>

car_type = <age:int; fuel:int; speed:int>
```

Since the fields of `car_type` are a superset of the fields of `vehicle_type` and the types of the common fields are in a subtype relation since they are identical, `car_type` is a subtype of `vehicle_type`. The subtype relationship is totally dependent on the implementation of the records; subtype relationships could arise purely as a result of coincidental representations and similarly records which are conceptually in a subtype relationship may not have a record subtype relationship in a particular implementation [6, page 55].

Function subtyping is more complicated and less intuitive. The rule for subtyping between two functions is given by [7]:

$$(F_i : B \rightarrow C) \leq (F_j : A \rightarrow D) \Leftrightarrow (A \leq B) \wedge (C \leq D)$$

For F_i to be a subtype of F_j, the result type, C, of F_i must be a subtype of the result type, D, of F_j. This requirement expresses the fact that the result type, C, of F_i must be acceptable in any context where the result type, D, of F_j is acceptable; it seems intuitive that this will be so if C is a subtype of D. The requirement for the argument types can at first sight seem counter intuitive: for F_i to be a subtype of F_j, the argument to F_i must be a *supertype* of the argument to F_j, that is $A \leq B$. On reflection it is clear that if F_i is to be used anywhere F_j can be used, then it must be able to accept any parameter that F_j can accept. For this to be possible, the domain of F_i must not be less than the domain of F_j. Taking an example from [7, page89], $F1 : 2..7 \rightarrow 10..15$ is

a subtype of $F2 : 3..6 \rightarrow 7..20$ since 2..7 is a supertype of 3..6 and 10..15 is a subtype of 7..20. The requirement that the argument of the subtype function must be a supertype of the argument to the supertype function is known as *contravariance.*

The contravariant rule guarantees that a subtype instance can be used anywhere a supertype instance is specified. Contravariance can be used in object-oriented languages to prevent the possibility of dynamic type errors [8]. Subtypes are usually intended to portray a specialised version of a supertype and in this respect contravariance has its limitations. It is not often useful to redefine a function in a subtype so that its domain is larger than that of the supertype function since this has connotations of generalisation rather than specialisation. In contrast to contravariant rules, *covariant* rules for function subtyping do permit the arguments of a subtype function to be specialised to subtypes of the arguments to the supertype function. However, covariant rules do not prevent the possibility of dynamic type errors and in addition there is no static covariant type checking algorithm that precisely separates all type-correct programs from type-incorrect programs. In contrast, there are algorithms for reliably separating type-correct from type-incorrect programs under contravariant rules.

It would be desirable to have the security of dynamic type checking as provided by contravariance together with the ability to specialise subtypes as provided by covariance. This might be achieved by extending the present concept of subtyping so that it encompassed more than the inclusion relation. Alternatively, we require that the relationship between superclasses and their subclasses is not limited to the subtype relation; we return to this view in section 8.

6 Classes and Types

6.1 Are classes the same as types?

Types can be concerned with abstract behaviour as defined by signatures or interfaces which are independent of implementation. Thus a type may be implemented in more than one way but still remain the same type: a *Stack* implemented as an array and a *Stack* implemented as a linked list have the same type. Implementation is irrelevant in that a *Stack* implemented as an array (which is itself a type) and a *Queue* implemented as an array are different types.

Classes as used in object-oriented programming languages are implementation dependent. A class which implements a *Stack* as an array and a class which implements a *Stack* as a linked list are *different* classes even though they have the same type. Classes provide templates from which objects of the class are created and classes have a type; a type however need not have a class, it can be implemented by other means.

Consider two classes, `StackA` and `StackL`, which implement a *Stack* as an array and a linked list respectively. An object `s1` of class `StackA` and an object `s2` of class `StackL` have the same type *Stack*. If types and classes are kept separate, then it would be possible to use object `s1` in place of object `s2` because they have the same type. However in many languages, of which Eiffel is one, types and classes are not separated; `s1` and `s2` would not be considered to have the same type and would not therefore be dynamically interchangeable.

6.2 Subtypes and subclasses

There is a distinction between subtypes and subclasses. A subtype encompasses all the *behaviour* of a supertype. A subclass inherits *implementation* from a superclass and this may or may not result in the subclass being a subtype of the superclass. A subclass may specialise a superclass by extension, new methods are added and the subclass is a subtype of the superclass. If a subclass redefines superclass methods by changing the implementation, then the subclass and the superclass have the same type. If a subclass restricts the behaviour of a superclass by only inheriting some of the superclass methods, then the subclass will be a supertype of the superclass.

It is also possible for two classes to have a subtype relationship but not a subclass relationship. In other words, the behaviour of the classes is in a subtype relation but the classes are not related in the implementation hierarchy.

It is apparent that behaviour hierarchies as defined by types and implementation hierarchies as defined by classes need not necessarily coincide. If types and classes are not distinguished, as in Eiffel, then type and class hierarchies are forced to coincide.

7 Inheritance

7.1 Inheritance as subtyping

Taking the view of inheritance that it is a type composition mechanism means that the properties of one or more types can be reused in the definition of a new type [1]. The structure imposed upon a collection of related types can enhance the conceptual clarity of a system specification.

If the implementation of a type is ignored, then inheritance can be viewed as interface inheritance; that is, inheritance of the signature and maybe of the semantics.

However, in many object-oriented languages (Eiffel, C++, Modula-3, Trellis) inheritance of implementation via classes is tightly coupled with subtyping such

that the class inheritance hierarchy determines the subtype relation. In untyped languages such as Smalltalk, the relationship between classes in an inheritance hierarchy is purely one of code reuse.

7.2 Inheritance is more than subtyping

However, inheritance is not just subtyping [9]. When inheritance is considered to include implementation, then a new class created by inheritance may not stand in a subtype relation to the class from which it inherited. The erroneous assumption that a subclass is always a subtype of the superclass from which it inherits caused early versions of Eiffel to have an insecure type system [8]. If the subtype relation is to be based on the implementation inheritance hierarchy then, in order to maintain a secure type system, inheritance must be restricted such that subclasses always maintain a subtype relation with their superclasses. Cook et al [9] propose a type system for inheritance which does not restrict subclasses to being subtypes. They define three forms of inheritance:

1. Object inheritance which subsumes both delegation and the traditional concept of class inheritance; new objects in classless languages reuse the implementation of the prototype objects which delegated to them and new classes reuse the implementation of the classes from which they inherit.

2. Class inheritance which is related to meta-classes and supports incremental definition of classes for instance creation.

3. Type inheritance used for the definition and checking of recursive types. Type inheritance is a means of constructing the types of objects produced by object inheritance.

In type inheritance, F-bounded quantification [10] is used to enable functions to be written such that they can be applied to variables of different types which are related by implementation (object) inheritance but which are not in a subtype relationship to each other. The advantage of this approach is that it is possible to have the protection of static type checking but still to have the flexibility to inherit implementation without the constraint that the subtype relation must be maintained between classes related by implementation inheritance.

In order to understand the Cook inheritance model, we need to consider polymorphic functions in more detail.

8 Polymorphic functions

8.1 Polymorphism and bounded quantification

Bounded polymorphism was used by Cardelli and Wegner [1] as a way of typing functions that operate uniformly over all the subtypes of a given type. Cardelli and Wegner modelled objects as records whose fields contained functions representing methods. Subtypes are obtained by adding fields. The type of a record, which describes the interface of the object, is a mapping of the field labels to the field types. If we let such a mapping be called `T` then any type which contains `T` as a subset of its own record type is considered to be a subtype of the type with `T` as interface. Thus if a type, `T1`, that merely prints a representation of itself is defined as:

$$\texttt{T1} = \{\texttt{print: void} \rightarrow \texttt{string}\}$$

Then any type which has a print record of this type is considered to be a subtype of `T1`. Thus,

$$\{\ldots, \texttt{print: void} \rightarrow \texttt{string}, \ldots\} \leq \texttt{T1}$$

It is possible to write a polymorphic function which is quantified such that its use is restricted to subtypes of `T1`:

$$\forall\ \texttt{t} \leq \texttt{T1}.\ \texttt{list[t]} \rightarrow \texttt{list[string]}$$

The function will take a list of subtypes of `T1` and print a list of the type representations.

This form of bounded polymorphism works well for simple types whose definitions are not self referential. Recursive types are defined by self reference; the signature of a recursive type contains at least one function in which the type itself appears as an argument and, or a result. The type `Point` defined below in section 8.2 is recursive since reference to `Point` is made in the functions *move* and *equal*. In object-oriented languages, an object refers to itself by the use of a pseudo variable such as *self* (Smalltalk), *current* (Eiffel) and *this* (C++). For recursive types bounded-quantification is too restrictive. A problem that can arise is that application of a polymorphic function to a recursive type can result in loss of information due to a supertype being returned when the desired result is for a subtype to be returned. The reasons why this is so are outlined below.

8.2 Polymorphism and F-bounded quantification

Cook et al [9] use a functional language based on typed polymorphic lambda-calculus to describe the typing of inheritance. In their model record types indicate exactly what fields a record contains. This is in contrast to the Cardelli subsumption model in which a record type represents all records that have at

least the specified fields. Classes are distinguished from types. Types are defined as syntactic interfaces such that interface compatibility can be checked at compile time. Classes are implementations of types and a given type may have more than one corresponding class. A type relation (an F-bound) is defined between type interfaces such that types in an F-bound relationship are polymorphic. The F-bound relation is used to characterise the extended types that may be created by inheritors and is less restrictive than the subtype relation.

To illustrate, we take an example from Cook et al [9]. A type `Point` is defined to specify the interface of movable planar points. Since the specification is functional, when a point is moved a new point is returned at the new location. Nevertheless, the definition assumes the existence of a point object which responds to the operations; without such an assumption, each of the operations requires an extra argument of type `Point`.

```
interface Point
     x : Real
     y : Real
     move(Real, Real) : Point
     equal(Point) : Boolean
```

The interface is recursive since the interface defines *move* as returning a `Point` type (positive recursion) and *equal* as having a `Point` type as argument (negative recursion).

The interface can be generalised by defining a type function, `F[t]`, so that it can apply to any type `t`.

```
F[t] =
      {x : Real,
       y : Real,
       move : Real, Real -> t,
       equal : t -> Boolean}
```

The interface `Point` is obtained by binding `t` to `Point` in `F[Point]`.

The recursive structure of the type `Point` can be extended incrementally to define a `ColourPoint` by using *type inheritance*. In type inheritance only syntactic interfaces are inherited; semantics and implementation are ignored.

```
interface ColourPoint
     inherits Point
     colour : Colour
```

In type inheritance not only are new fields added to the type, here a field has been added to hold a method to return the type `Colour`, but also the

type of inherited fields may be changed. If such a change happens, then the new type will not necessarily be a subtype of the type from which it inherited. It is therefore possible to inherit a type interface without having to maintain a subtype relationship. This is analogous to implementation inheritance by classes in which the subtype relationship may not be maintained.

The type inheritance defined by `ColourPoint` is regarded as an extension of the type function `F[t]` and is expressed by the type function `G[t]`.

```
G[t] = F[t] + {colour : Colour}
```

When we bind `t` to `ColourPoint` in `G[ColourPoint]`, then all references to `t` in `F[t]` are also bound to `ColourPoint`:

```
interface ColourPoint
     x : Real
     y : Real
     move(Real, Real) : ColourPoint
     equal(ColourPoint) : Boolean
     colour : Colour
```

`ColourPoint` is not a subtype of `Point` since the *equal* method is contravariant. If `G[t]` had been defined so that `ColourPoint` was a subtype of `Point`, then we would have in effect:

```
G'[t] = F[Point] + {colour : Colour}
```

`G'[ColourPoint]` would then expand to

```
interface ColourPoint
     x : Real
     y : Real
     move(Real, Real) : Point
     equal(Point) : Boolean
     colour : Colour
```

The *move* method could be applied to a `ColourPoint`, due to the subtype relationship, but the type that would be returned would be a `Point` with the result that the information concerning the colour of the `ColourPoint` instance would have been lost. The ability to change the types in the inherited interface via `G[t]` prevents the loss of such information. In addition, the equality method can be redefined so that when applied to an instance of type `ColourPoint`, the colour of points is compared as well as the location. (However, there is no need

to do this; an alternative would be to add another method, *sameColour*, to `ColourPoint` to compare colours and a method, *identical*, which could invoke *equal* and *sameColour*.)

Cook considers that even though `ColourPoint` is not a subtype of `Point`, the fact that `ColourPoint` does have the same recursive structure as `Point` (`ColourPoint` $\leq$ `F[ColourPoint]`) and the fact that for all `t`, `G[t]` $\leq$ `F[t]`, means that an inheritance relationship is indicated.

Classes, `cart-point` and `colour-point`, can be written to implement the types `Point` and `ColourPoint` respectively:

```
class cart_point (x : Real, y : Real)

        implements Point

        method x : Real
               return x

        method y : Real
               return y

        method move (dx : Real, dy : Real) : Point
               return new myclass(self.x + dx, self.y + dy)

        method equal (p : Point) : Boolean
               return (self.x = p.x) and (self.y = p.y)
```

It is clear that class and type are separate; instances of class `cart_point` have type `Point`. *Self* and *myclass* are analogous to *self* and *self class* in Smalltalk. When the *move* and *equal* methods are invoked, the pseudo variable *self* is bound to the receiver of the message such that *self.x* returns the receiver object's `x` coordinate. The pseudo variable *myclass* is bound to the class of the receiver object and is used to create an object of the same class as *self*. Instances of `cart_point` are recursive because they send messages to *self*. The class `cart_point` is also recursive because the *move* method uses *myclass* to create a new point at a given distance from itself.

The class `colour_point` is defined by inheritance from the class `cart_point`. The methods x, y and *move* are inherited. An additional method, *colour*, is defined using an additional instantiation parameter for the class. The *equal* method is redefined to compare the colour of two points as well as their location.

```
class colour_point (x : Real, y : Real, c : Colour)

        implements ColourPoint

        inherit cart_point(x,y)
               translating new myclass(x',y')
                        to new myclass(x',y',self.colour)

        method colour : Colour
               return c

        method equal(p : ColourPoint) : Boolean
               return super.equal(p) and (self.colour = p.colour)
```

The `inherit cart_point(x,y)` statement indicates how to instantiate the inherited `cart_point` class. The subclass parameters (`x`, `y` and `c`) are translated to the form required by the superclass; in this case the parameter `c` is ignored. The statement `translating new myclass...` indicates how recursive calls within `cart_point` are to be translated to construct `colour_point` objects. Thus when `new myclass` is called from the *move* method but with a `colour_point` object, a `ColourPoint` type object will be created by the *move* method and the `colour_point` object will retain its colour. The modified *equal* method uses *super* to invoke the original method defined in `cart_point` and to add a new constraint for colour equality.

The *move* and the *equal* methods defined in class `cart_point` take a Point type as argument but can be invoked with a `ColourPoint` as argument because of the F-bound relationship. The methods in the class `cart_point` can be invoked by any type such that $\texttt{t} \leq \texttt{F[t]}$. The `colour_point` class methods can be invoked by any type such that $\texttt{t} \leq \texttt{G[t]}$. Since $\texttt{G[t]} \leq \texttt{F[t]}$ we have that $\texttt{t} \leq \texttt{G[t]} \leq \texttt{F[t]}$ and hence that $\texttt{t} \leq \texttt{F[t]}$. In general, any type satisfying $\texttt{t} \leq \texttt{G[t]}$ also satisfies $\texttt{t} \leq \texttt{F[t]}$. Hence we have that since `ColourPoint` satisfies $\texttt{t} \leq \texttt{F[t]}$ as required for the `cart_point` class, methods defined for the type `Point` in the `cart_point` class can be invoked by objects of type `ColourPoint`.

The type constraint $\texttt{G[t]} \leq \texttt{F[t]}$ must be maintained by inheritance and it is this constraint which ensures that inherited objects can be used as arguments to F-bounded polymorphic functions just as values of subtypes can be used as arguments to subtype bounded polymorphic functions.

Even though there is no subtype relation between `Point` and `ColourPoint`, if inheritance maintains the F-bound relationship, then functions defined for objects of type `Point` can be applied to objects of type `ColourPoint`. However, the F-bound concept does not permit the assignment of instances of class `colour_point` to instances of class `cart_point`; such assignments are only pos-

sible if the subtype relation holds. Consider the following program fragment:

```
p: cart_point
cp1, cp2: colour_point

new cp1(2.0, 3.0, red)
new cp2(2.0, 3.0, blue)
```

Since there is no subtype relation, it is not acceptable to make the assignment

```
p := cp1
```

This in turn prevents the legality of type (unsafe) statements such as

```
p.equal(cp2)
```

in which the *equal* method invoked would be that for `cart_point` (since `p` is of class `cart_point`), resulting in `p` and `cp2` being declared equal. The fact that their colours are different would be ignored by the `cart_point` *equal* method.

In the F-bound model of typed inheritance there is a clear separation of concerns; inheritance is separated from subtyping and types are separated from classes.

The ideas of Cook et al are based on the interpretation of interfaces as signatures and as such express an elegant approach to inheritance. However, since the behaviour of methods is not expressed by semantics, it is necessary to have an intuitive feel about the intended behaviour of methods so that one is aware, for example, that *equal* compares colour in interface `ColourPoint` but not in interface `Point`. In the absence of such intuition, it becomes necessary to look at the implementation of *equal* in the `colour-point` class.

The F-bound concept of type inheritance gives rise to the possibility of three types of inheritance hierarchies in object-oriented systems:

- type hierarchies related by subtypes
- type interface hierarchies related by F-bounds and possibly but probably not by subtypes
- class hierarchies related by implementation and maybe by subtypes and maybe by F-bounds.

8.3 Ad hoc polymorphism quantified

Another way of quantifying types without relying on a subtype relation has been developed by Wadler and Blott [11]. In order to provide a unified approach to the typing of overloaded arithmetic operators and functions defined in terms of these operators, Wadler and Blott have defined *type classes*. A *type class* is a class to which many types can belong and the concept is integrated into the type system of the functional programming language Haskell [12]. A type class declares the names and signatures of functions which are expressed in terms of a type variable which ranges over every type. To belong to a class, a type must have functions of the same names defined on it and the functions must be appropriately typed. Hence a type class contains an interface which is purely syntactic, whereas an instance of the class is a type which is an implementation of the interface.

An example from Wadler and Blott is used to illustrate the concept. A type class, `Num`, is defined with operators (+), (*) and *negate*:

```
class Num a where
        (+), (*) :: a -> a -> a
        negate   :: a -> a
```

The class declaration states that a type belongs to class `Num` if there are functions named (+), (*) and *negate*, which are bound to the appropriate type, defined on it. The type variable `a` can range over any type.

An instance, the type `Num` Int, of the class can be defined as:

```
instance Num Int where
        (+)  = addInt
        (*)  = multInt
        negate  = negInt
```

It is assumed that *addInt*, *multInt* and *negInt* have been previously defined on integers, if this is not the case then the definitions would occur in the type instance declaration. The type system used is based on that of Hindley/Milner [11] and as such type declarations can be inferred. For `Num Int`, the type inference algorithm must verify that the bindings of the operation names to the operation definitions do have the appropriate type. For example, *addInt* should have type `Int -> Int -> Int`. In the instance declaration, `Int` is a type constructor.

Another instance, the type `Num` Float, of the class can be defined similarly as:

```
instance Num Float where
        (+)  = addFloat
        (*)  = multFloat
        negate  = negFloat
```

If (+) is applied to integers, the *addInt* code will be invoked and if it is applied to reals, the *addFloat* code will be invoked. This is analogous to conventional overloading in languages such as Standard ML. It is when functions defined in terms of the overloaded operators are considered that the benefits of type classes over conventional overloading become apparent.

Functions can be defined for the class `Num` such that the functions can be applied to any type which is a member of the class. Consider the function *square* defined as:

```
square = x (*) x
```

The type of *square* can be inferred as:

```
square :: Num a => a -> a
```

In other words, *square* has type `a -> a` for every `a` such that `a` belongs to class `Num`.

It is now possible to apply *square* in ways such as:

```
square 4
square 4.25
```

In Standard ML it is not possible to define functions in terms of overloaded operators. In other approaches to overloading, a function such as *square* could be written but would not have a parametric polymorphic type; instead it would stand for two overloaded versions of *square* with types `Int -> Int` and `Float -> Float`. This may not seem much of a problem but exponential growth of the translations involved can soon occur. Consider the function *squares* defined as:

```
squares (x, y, z) = (square x, square y, square z)
```

The parameters `x`, `y` and `z` may each be typed as an Int or a Float which gives rise to eight possible types for *squares*. The elegance and simplicity of the type class approach is apparent.

A program containing class types and instance declarations can be translated at compile-time to an equivalent program that does not have class types but

which still has a valid Hindley/Milner type. To accomplish this, a new type is defined for each type class such that the type corresponds to a dictionary which contains the operations for the class. For the class `Num` the dictionary type, `NumD`, is defined as:

```
data NumD a = NumDict (a -> a -> a) (a -> a -> a) (a -> a)
```

Functions are defined to access the dictionary. For example,

```
add (NumDict a m n) = a
mul (NumDict a m n) = m
neg (NumDict a m n) = n
```

For each instance of the class, a value of the dictionary type is declared which binds the operations in the class dictionary to the implementations of the operations which are appropriate for the type:

```
numDInt     :: NumD Int
numDInt     = NumDict addInt multInt negInt

numDFloat   :: NumD Float
numDFloat   =  NumDict addFloat multFloat negFloat
```

A term such as `x + y` is then replaced by a corresponding term as:

```
x + y --> add NumD x y
```

If `x` and `y` are `Ints`, then *add* accesses the dictionary defined for `Num Int` whereas if `x` and `y` are `Floats`, then *add* accesses the dictionary defined for `Num Float`. Thus *add* will access `NumDInt` to retrieve *addInt* or `NumDFloat` to retrieve *addFloat*. Hence we have translations such as:

```
2 + 3 --> add NumDInt 2 3 --> addInt 2 3
```

If the type of a function contains a *class*, then this is translated into a dictionary which is passed at run-time rather than compile-time.

8.3.1 Subclasses of type classes

A type class can be defined such that it is a subclass of another type class. To be a member of a subclass, a type must also be a member of the superclass. Suppose a type class `Eq` is defined to express equality as in [11]:

```
class Eq  a where
        (==) :: a -> a -> Bool
```

with instances:

```
instance Eq Int where
        (==) = eqInt

instance Eq Char where
        (==) = eqChar
```

If we want every type that is an instance of `Num` to have the equality operator (==) defined on it as well, then we can do this by making `Num` a subclass of `Eq`:

```
class Eq a => Num a where
        (+)    :: a -> a -> a
        (*)    :: a -> a -> a
        negate :: a -> a
```

This type class declaration asserts that `a` may belong to class `Num` only if it also belongs to class `Eq`. Now the instance declaration `Num Int` is only valid if there is also an instance declaration `Eq Int` active within the same scope.

If a function is defined over the type `Num a`, then the qualifier `Eq a` is implied and the equality operation can be used in the function definition.

A type class may have multiple superclasses and subclasses and the translations ensure that the appropriate dictionaries are passed at run-time.

8.4 The type class PlanarPoints

We shall apply the type class approach to the planar points example used for discussing F-bounds. We declare a type class `PlanarPoints` such that the types `CartPoint`, `PolarPoint` and `ColourPoint` could be used to create instances of the class.

```
class PlanarPoints a where
        x, y :: a -> Real
        move :: a -> Real -> Real -> a
        equal :: a -> a -> Boolean
```

```
instance PlanarPoints CartPoint where
        x = xCartPt
        y = yCartPt
        move = moveCartPt
        equal = equalCartPt

instance PlanarPoints PolarPoint where
        x = xPolPt
        y = yPolPt
        move = movePolPt
        equal = equalPolPt

instance PlanarPoints ColourPoint
        x = xColPt
        y = yColPt
        move = moveColPt
        equal = equalColPt
```

These definitions assume that the types `CartPoint`, `PolarPoint` and `ColourPoint` have been previously defined such that the definitions include implementations of the types. Thus *xCartPt* is the implementation of the function defined to return the x co-ordinate of a variable of type `CartPoint`.

A function, *movePoints*, could be defined to move each point in a list of points by the same amount regardless of whether the list of points contains `CartPoints` and, or `PolarPoints` and, or `ColourPoints`. Different code would be dynamically invoked to move points of a different type.

```
movePoints :: PlanarPoints a => [a] Real Real -> [a]

        movePoints [] m n = []
        movePoints (x cons xs) m n =
                        (move x m n)cons(movePoints xs m n)
```

The function that is invoked by *move* for a point `x` will depend on the type of `x`; for example, if `x:CartPoint` then *move* will invoke *moveCartPt*. The function works for lists which contain any type which is an instance of the type class `PlanarPoints`; the types do not have to be in a subtype relationship.

However, it would not be possible to compare points of different types for equality. Consider variables defined as:

```
cp : CartPoint
cp2 : CartPoint
pp : PolarPoint
colP : ColourPoint
```

The comparison *equal(cp, cp2)* is legal but *equal(cp, colP)* would be a type error since equal is only defined for arguments of the same type. This is desirable in this situation because a `CartPoint` does not have a colour and equality between `CartPoints` and `ColourPoints` is not meaningful. However it also means that *equal(cp, pp)* is also not legal which is perhaps not so desirable. This disadvantage could perhaps be overcome by type coercion.

Subclasses can be used to extend the behaviour of classes. It would, for example, be possible to declare a new type class, `NewPlanarPoints`, which extended `PlanarPoints` by adding an operation to compare the x coordinates of two points:

```
class PlanarPoints a => NewPlanarPoints a where
        samex :: a -> a -> Boolean

instance NewPlanarPoints CartPoint where
        samex (p1, p2) = xCartPt(p1) == xCartPt(p2)
        (assuming the equality operation == is defined for Reals)
```

A type instance can only be created for class `NewPlanarPoints` if the type constructor to which `a` is bound in `NewPlanarPoints` is also bound to the same type constructor in `PlanarPoints`, for example `CartPoint` bound to `a` in both. A new type has not been created, `CartPoint` for example stays the same, but the creation of a new type class has enabled code defined in the class `PlanarPoints` to be reused. The `PlanarPoints` class declaration and the class instances, which define class implementations, have been (strictly) inherited by `NewPlanarPoints`. (This concept of subclasses is somewhat analogous to enrichment in OBJ1 [13].)

9 Relating F-bounds and type classes

Class types are a form of bounded quantifier in that they limit the types that a type may be instantiated to in a polymorphic function. An F-bound is also a bounded quantifier. Neither type classes nor F-bounds require the existence of subtype relationships between types, although neither exclude the possibility of subtype relations.

Type class declarations are similar to Cook interfaces except that Cook interfaces are part object-oriented and part functional;

```
move: Real -> Real -> t
```

There is an assumption that an object of type `t` exists but the object is not mutable and when it is moved a new object is returned.

Type class definitions are only functional;

```
move: a -> Real -> Real -> a
```

Instance declarations (implementations) for type classes in Wadler and Blott are similar to class implementations of types in Cook et al. In both cases, the implementations can be changed while the type interfaces remain unaltered.

Wadler and Blott polymorphic functions are bound to type classes such that they apply to any type in the type class and for each type in the class, different code will be invoked. F-bounded polymorphic functions apply to all types such that `t` $\leq$ `F[t]` and the *same* code can be invoked for different types although different code can also be invoked.

The Wadler and Blott concept of subclasses enables the operations defined for a type class to be inherited by a subclass. Extra operations for the type can be defined in the subclass. However, the types declared in the superclasses are not specialised in the subclasses; the types remain the same. In F-bound inheritance, not only can new operations be defined in subclasses, but also the types themselves can be specialised.

It is interesting to note that in an alternative approach to modelling inheritance with a static type system, Ghelli [14] models message passing by using overloaded functions which are not based on a record model. Inheritance is modelled by the incremental definition of overloaded functions and message passing by the application of these functions. When an overloaded function is applied to an object, the code invoked depends on the class which created the object. Significantly, the approach provides for both covariant and contravariant redefinition of methods. As a result, the covariant redefinition of the *equal* method as defined for `Point` is permitted for `ColourPoint`. It is accepted that if a `Point` object is compared with a `ColourPoint` object then both will be treated as `Points` and hence colour will be ignored. If this is not the desired behaviour, then it is also possible to program explicitly how the objects are to be compared; for example, `Point` and `ColourPoint` objects could always be regarded as unequal. Whereas with Wadler and Blott type classes, different code is invoked for compatible types within a type class, in the Ghelli approach the code invoked for compatible types can be the same or different.

Both type classes and F-bounds are concerned only with the syntactic representation of types. Consideration of semantics is obviously more complicated.

10 Conclusion

It seems desirable to have separate inheritance hierarchies for types and classes. If the type inheritance hierarchy was based on the subtype relation, it could reflect a semantic interpretation of is-a relationships. However, depending on the subtype model used, the semantic interpretations might be restricted by subtype requirements such as contravariance. The subtype hierarchy could be used to determine whether an object of a subclass could be used in any context (parametric or assignment) where an object of a superclass was expected. The class inheritance hierarchy could be used to reuse code where appropriate.

It would also be possible to have type interface hierarchies related by F-bounds; such hierarchies could reveal relationships between classes which while being conceptually meaningful (a `ColourPoint` is a specialised `Point`) are not subtype relationships. F-bound relationships show how implementations can be inherited and show when types created by inheritance can be used in parametric contexts requiring supertypes. The F-bounded types cannot replace one another in assignment contexts.

In the `Point` and `ColourPoint` example, the `ColourPoint` type would not be a subtype of the `Point` type so that it would not be possible to use a `ColourPoint` where a `Point` was expected. Thus undesirable behaviour such as comparing the equality of a `Point` object (without colour) with a `ColourPoint` object (with colour) would be prevented. On the other hand an implementation of class `ColourPoint` could inherit code from an implementation of class `Point` such that, for example, the *move* method from `Point` was inherited by the class `ColourPoint` and the *equal* method was inherited and extended to include the comparison of colour. Again, the *equal* method might be completely redefined in `ColourPoint` and perhaps only the methods for accessing the coordinates and for moving a point would be inherited from `Point`.

The significant advantage of the F-bound relationship over the subtype relationship is that the type of methods can be specialised in the subclasses since contravariance is not mandatory under the F-bound relationship. The necessity to obey the contravariant rule in the subtype relationship places severe limitations on the degree of specialisation that can be achieved in subclasses which are also required to be subtypes.

Wadler and Blott type class hierarchies show how implementations can be inherited and define which types belong to a class and hence which types are polymorphic. For types to belong to subclasses, the types must belong to superclasses so it appears that subclass instances can be used where superclass instances are expected.

Although the interfaces used in the F-bounded view of inheritance and in the Wadler and Blott type classes consist only of operation names and types and are

therefore weaker than complete behavioural specifications with semantics, they have the advantage that interface checking reduces to type checking for which there is an existing technology. The use of F-bounds and type classes would enable polymorphic functions to be statically type checked without requiring a subtype relation between inherited classes.

The existence of separate type and class inheritance hierarchies would seem to have implications throughout the development of a system. We will not discuss such issues here but we hope to address them at another time.

References

[1] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys*, 17(4), December 1985.

[2] Ben Potter, Jane Sinclair, and David Till. *An Introduction to Formal Specification and Z.* Prentice Hall International (UK) Ltd, 66 Wood Lane End, Hemel Hempstead, Hertfordshire HP2 4RG, 1991.

[3] R.Burstall and B.Lampson. A Kernel Language for Modules and Abstract Data Types. Technical report, Digital Systems Research Center, 130 Lytton Avenue, Palo Alto, California 94301, September 1984.

[4] Peter S. Canning, William R. Cook, Walter L. Hill, and Walter G. Olthoff. Interfaces for strongly-typed object-oriented programming. *OOPSLA '89 Proceedings*, 1(6), October 1989.

[5] Colin Atkinson. *Object-Oriented Reuse, Concurrency and Distribution.* Addison-Wesley, Wokingham, England, 1991.

[6] Scott Danforth and Chris Tomlinson. Type Theories and Object-Oriented Programming. *ACM Computing Surveys*, 20(1), March 1988.

[7] Gordon Blair, John Gallagher, David Hutchinson, and Doug Sheperd. *Object-Oriented Languages, Systems and Applications.* Pitman Publishing, 128 Long Acre, London WC2E 9AN, 1991.

[8] W.R.Cook. A Proposal for Making Eiffel Type-safe. *The Computer Journal*, 32(4), 1989.

[9] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is Not Subtyping. *Proceedings of the 17th ACM Symposium on Principles of Programming Languages*, January 1990.

[10] Peter Canning, William Cook, Walter Hill, and Walter Olthoff. F-Bounded Polymorphism for Object-Oriented Programming. *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, 1989.

[11] Philip Wadler and Stephen Blott. How to make *ad-hoc* polymorphism less *ad hoc*. In *Proceedings 16th ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM, January 1989.

[12] Paul Hudak and Joseph H. Fasel. A Gentle Introduction to Haskell. *ACM SIGPLAN Notices*, 27(5), May 1992.

[13] C.D.Walter, R.M.Gallimore, D.Coleman, and V.Stavridou. OBJ1 Manual Version 1.0. Technical report, UMIST, Department of Computation, UMIST, Manchester M60 1QD, 1986.

[14] Giorgi Ghelli. A Static Type System for Message Passing. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, pages 129–145. ACM, November 1991.