Chapter 4

Polymorphic Inheritance

This chapter explores what it means for a type to inherit functions.

After successive attempts to describe classes as types and inheritance as the incremental creation of subtypes, the focus moves to a different model of classification. In the new approach, classes are not types; rather they are polymorphic type families. The most widely-known treatment of polymorphism is Girard-Reynolds Universal Quantification. Two variants on this have been proposed for handling polymorphic inheritance in object-oriented languages, these being Bounded Quantification and F-Bounded Quantification.

4.1 Covariance and Contravariance

When Cook reported, in his *Proposal for Making Eiffel Type-safe* [Cook89b], that there were loopholes in *Eiffel*'s type system, this came as a surprise to those who look to *Eiffel* as a model for the strongly-typed object-oriented languages. The so-called "*Eiffel* type failure" problem created a stir at the time. The crux of the problem stems from *Eiffel*'s intention to support subtyping, while not managing to obey all the rules necessary to achieve this. The mathematical arguments in favour of subtyping are weighed against the practical objections to implementing them in *Eiffel*.

4.1.1 On the Impact of Contravariance

In *Eiffel*, a class *conforms* to another if it inherits from it [Meye92, p219], whether directly or transitively. *Eiffel*'s *conformance*[1] deviates from subtyping

---

[1] A term of pure *franglais*, since the noun from "to conform" is "conformity" ☺.

by allowing the uniform specialisation of function arguments and results in descendent classes. This is in violation of contravariance, which insists on the generalisation of argument-types in replacement functions. Programs may therefore be passed as type-correct, but hide run-time type failure [Cook89b]. This may be illustrated with a simple example involving the covariant redefinition of a function argument type in figures 4.1 and 4.2.

In figure 4.1, a simple Cartesian POINT class possessing *x* and *y* cooordinates and a procedure to *move()* points is extended in the inheriting class HOT_POINT by adding the *selected* attribute and procedures to *select* and *deselect* the point. HOT_POINT represents the class of interactively-selectable points in some display environment. If we disregard the typing of *equal()*, HOT_POINT is in all other respects a subtype of POINT and objects of type HOT_POINT may safely be passed to variables of type POINT.

```
class POINT                          -- simple Cartesian point
feature { ANY }
        -- data declarations
x, y : INTEGER;
        -- functions and procedures
move (nx, ny : INTEGER) is
   do x := nx;  y := ny end;
equal (other : POINT) : BOOLEAN is
   do Result := (x = other.x and y = other.y) end
end -- POINT


class HOT_POINT                      -- selectable Cartesian point
inherit POINT
   redefine equal                    -- to test extra selected attribute
feature { ANY }
        -- additional data declarations
selected : BOOLEAN;
        -- additional procedures
select is
   do selected := true end;
deselect is
   do selected := false end;
        -- redefined equal accepts HOT_POINT argument
equal (other : HOT_POINT) : BOOLEAN is
   do Result := (x = other.x and y = other.y and
        selected = other.selected) end
end -- HOT_POINT
```

Figure 4.1: Covariant Argument Redefinition

The salient fact here is that the function *equal()* defined in POINT, which accepts an argument in the same type POINT, has been replaced in HOT_POINT by a function expecting a subtype argument HOT_POINT, in violation of contravariance. This seems reasonable at first, since objects should be compared with other objects of the same type. The apparent wisdom of this is further indicated in the redefined body of *equal()*, in which the

additional attribute *selected* is compared with that of the argument, which must therefore be a HOT_POINT.

However, figure 4.2 illustrates what happens when a variable of type POINT is passed an object of type HOT_POINT. In the program fragment, a variable *p* is passed both a POINT instance and a HOT_POINT instance on different occasions. Initially, the general application of the *move()* procedure works successfully as one might expect; however the general application of *equal()* has disastrous results. Here, since *p* now legally contains an instance of HOT_POINT, *p.equal(point)* invokes the replaced function defined in HOT_POINT. This call was checked statically with respect to POINT, yet the replacement function seeks to access the *selected* attribute of a POINT, which it clearly does not possess. The combination of covariant argument redefinition, aliasing and dynamic binding lead to the point of type failure.

```
                                -- Simple program fragment


p, point : POINT;
hotpt : HOT_POINT;


!! point;                       -- default initialisation to (0, 0)
!! hotpt;                       -- default initialisation to (0, 0, false)
p := point;
p.move(3, 4);
p := hotpt;                     -- ok since HOT_POINT conforms to POINT
p.move(3, 4);
... p.equal(point);             -- run-time type failure!
```

Figure 4.2: Type Failure due to Aliasing

While insisting that *Eiffel* should obey the contravariant rule for argument redefinition, Cook ruefully admits that:

> "[Contravariance] has the unfortunate effect of making argument type redefinition almost useless, since it is usually not very useful to allow a redefined method to accept a larger class of arguments" [Cook89b, p62].

Contravariance is a counter-intuitive finding because it prevents the uniform specialisation of function arguments and results. It forbids the replacement of a function $f : \tau \rightarrow \tau$ closed over a type τ by a function $g : \sigma \rightarrow \sigma$ closed over a subtype $\sigma \subseteq \tau$ [Card86].

4.1.2 On the Avoidance of Contravariance

Cook's many suggested amendments to *Eiffel*'s type rules [Cook89b] were intended to enforce strict subtyping. These included linking exports with inheritance, forbidding the redefinition of attribute types, inverting the function argument redefinition rule to observe contravariance, judging type compatibility between parameterised types after replacing the type parameters and introducing an explicit type attribute scheme to handle *Eiffel*'s anchored types.

In his reply to Cook [Meye89], Meyer objected to the linking of exports with inheritance, which he called impractical, and especially to the adoption of contravariance. Rather than change *Eiffel*'s type rules to obey strict subtyping, Meyer introduced a patch, called the "global system validity check", to catch type errors retrospectively in situations where polymorphic aliasing would lead to run-time type failure. Meyer's patch [Meye89, p14-17] monitors aliasing of the kind *p := hotpt*; and in this context retypes the features of POINT with the most restricted types of any object it aliases, anywhere in the system. For this reason it is called a "global" check. Here, *equal()* would be retyped with the signature POINT $\rightarrow$ (HOT_POINT $\rightarrow$ BOOLEAN). At system assembly time, *p.equal(point)* would therefore raise a type error, where the retyped *equal()* is passed too general an argument. The published patch relies on a pessimistic, global flow analysis and errors are detected at a late stage. Recently, Meyer has proposed a different patch [Meye95], which is based on the idea of flagging covariant argument-type redefinitions, such that unsafe combinations of aliasing and polymorphic invocation are detected immediately. The same technique is used to trap the polymorphic invocation of routines which have been removed from the interface of descendent classes. Technically, Meyer's solution works, although from a mathematical standpoint it is unsatisfying, since it fails to address the basic soundness issue.

Meyer's refusal to adopt contravariance arose initially from observing the regularity captured by *Eiffel*'s *anchored types*:

> "Examples such as the above, of which there are thousands in practical *Eiffel* applications, make it very hard to imagine how significant object-oriented software can be written in a typed language without a *covariant* policy. Many of these examples use declaration by association, which is only a syntactical abbreviation, but in practice an essential one; its very availability for routine arguments is only possible because of the covariant rule" [Meye89, p12].

Anchored types are those declared "by association". The most common case is where an argument is said to have the type *like Current* (*ie* it is anchored to the type of *Current*, *Eiffel*'s name for *self*). Anchored types express something intuitive about classification which one would want to preserve in an object-oriented language, namely that a function f : $\tau \rightarrow \tau$ closed over the class τ can be inherited by a class σ, in which it is automatically retyped f : $\sigma \rightarrow \sigma$ and closed over the new class.

```
class POINT                          -- simple Cartesian point
feature { ANY }
        -- data declarations
x, y : INTEGER;
        -- functions and procedures
move (nx, ny : INTEGER) is
   do x := nx;  y := ny end;
equal (other : like Current) : BOOLEAN is
   do Result := (x = other.x and y = other.y) end
end -- POINT
```

Figure 4.3: Anchored Type Definition

In figure 4.1 fixed types were deliberately given to *equal()* and its redefinition to illustrate the problems contingent on adopting a covariant retyping policy. Figure 4.3 illustrates an alternative flexible typing in *Eiffel*, using an anchored type declaration. Here, the argument to POINT's *equal()* has the type *like Current*, standing for the same type POINT. When inherited by HOT_POINT, this type would adapt implicitly to HOT_POINT without the need for explicit redefinition. Meyer assumed that this mechanism was merely a "syntactic abbreviation" for type redefinition, subject to an interpretation in a simple subtyping model of inheritance [*cf* SOM93]. As a result, he was forced to conclude that a covariant policy should be observed elsewhere for argument redefinition. This conclusion is wrong, not least because it is based on a false assumption: anchored types are *not* syntactic abbreviations, they are better explained using a different mechanism.

Ironically, Cook discovered the *Eiffel* type failure problem while researching an alternative mathematical model of class inheritance, F-bounded quantification [Cook89a, CCHO89a, CCHO89b] which is different from subtyping [CHC90]. F-bounds were devised chiefly to explain the evolution of the *self*-type under inheritance. The F-bounded model is discussed more fully in the latter part of this chapter.

4.2 Inheritance as Subtyping

For those languages which support subtyping in full [SCBK86, RW92] or in part [Omoh94, SOM93, Stro91], a properly developed mathematical model of subtype-based inheritance is no mere academic matter. To facilitate this, a naïve model of inheritance is first explored, in order to examine the effects of adding incrementally to a type. From this, a pure subtyping model of inheritance is constructed, in which certain limitations are observed. Subtyping restricts the useful type information available, both in languages with subtype-inheritance and in those with independent type hierarchies [Amer90, BHJL86].

Both inheritance models presented in this section are based on the simply-typed λ-calculus [Chur40]. For convenience and brevity, only the types are considered, rather than values with their types. To model the extension of types, the existence of a simply-typed record combination operator $\oplus$ is assumed, which extends a record by appending to it a partial record of

additional fields. For the moment, this operator is only required to have an additive effect. Cook *et al*. [Cook89a, CP89, CHC90] have a combination operator with override; later we shall define our own version of this. A full λ-calculus derivation of record combination from first principles is given in Appendix 1.

4.2.1 A Naïve Typing of Inheritance

The existence of a universal type TOP is assumed, having the empty signature:

TOP = {}

It is vacuous, since it offers no information about how its instances should behave. All values are members of this type, since it is the logical supertype of all other types. We now wish to define the subtype OBJECT $\subseteq$ TOP, being the type of all objects, those values having a testable identity[2]:

OBJECT = {identity : OBJECT, equal : OBJECT $\rightarrow$ BOOLEAN}

OBJECT is clearly a recursive type. The existence of such a type is motivated by appealing to the standard approach to solving recursive equations [Read89, CP89]. By abstracting over the point of recursion, a construction is obtained, which is generally known in the λ-calculus as a *functional*. Functionals have the advantage that their definitions are not self-referential:

ΦOBJECT = $\lambda\sigma$.{identity : σ, equal : $\sigma \rightarrow$ BOOLEAN}

In the context of types, ΦOBJECT is also known as a *generator* [CHC90] for the type OBJECT. ΦOBJECT is a function from types to types - it is designed for application to a single type argument, which will replace the formal argument σ. It so happens that the type we would like to replace σ is in fact OBJECT, the very type we are trying to define. This leads to the observation that:

OBJECT = ΦOBJECT [OBJECT]

or, OBJECT is unchanged by the application of the generator ΦOBJECT. Types which exhibit this property are called *fixed points*, or *fixpoints* of their generator. Under certain conditions, it is possible to define a unique fixpoint, called the *least fixed point*, as the convergent limit of a sequence of self-applications of a generator:

OBJECT = ΦOBJECT [ΦOBJECT [ΦOBJECT [...]]]

---

[2] where *identity :* σ is of course a short-hand for the function *identity : UNIT* $\rightarrow \sigma$. We assume the technique described in chapter 2 using the UNIT type to avoid semantic problems with non-convergence when taking fixpoints.

Starting with the trivial application ΦOBJECT [$\perp$], where $\perp$ is the undefined value, a series of approximations to OBJECT may be constructed which, at the limit of convergence, is equal to the desired recursive type. A function called the *fixpoint finder*, $\mathbf{Y}$, is used to establish the fixpoints of generators. $\mathbf{Y}$ has the property:

$$\text{OBJECT} = (\mathbf{Y}\ \Phi\text{OBJECT}) \Leftrightarrow \text{OBJECT} = \Phi\text{OBJECT}\ [\text{OBJECT}]$$

in other words, applying the fixpoint finder to a generator yields the recursive type which is that generator's fixpoint. One definition of $\mathbf{Y}$ is given by:

$$\mathbf{Y} = \lambda f.(\lambda s.(f\ (s\ s))\ \lambda s.(f\ (s\ s)))$$

in which the trick of embedded self-application is used to produce the sequence of calls of the generator. Note that $\mathbf{Y}$ is not itself recursive, but establishes recursion from first principles. Semantic models supporting the existence of unique fixpoints depend on the construction of domains in which convergence can be proven [Scot76, Stoy77, MS82, MPS84, BM92].

The standard notation for the recursive type OBJECT, obtained by taking the fixpoint of the generator ΦOBJECT, is more usually given as:

$$\text{OBJECT} = \mu\sigma.\{\text{identity} : \sigma,\ \text{equal} : \sigma \rightarrow \text{BOOLEAN}\}$$

in which μ is used to bind the recursion variable σ. This economical notation is deemed equivalent to the infinite unrolling of the recursion in the type:

$$\text{OBJECT} = \{\text{identity} : \text{OBJECT},\ \text{equal} : \text{OBJECT} \rightarrow \text{BOOLEAN}\}$$

A POINT type is now defined to inherit from OBJECT. In the naïve interpretation of inheritance, the POINT record type is a straightforward extension of OBJECT, using $\oplus$ to compose OBJECT with the extra field types desired for a POINT. Let us suppose that it is reasonable to define free-standing extension record types, representing the additional fields to be incorporated in a base type. If the extension record type for a MOVEABLE object is defined as:

$$\text{MOVEABLE} = \mu\sigma.\{x : \text{INTEGER},\ y : \text{INTEGER},\ \text{move} : \text{INTEGER} \times \text{INTEGER} \rightarrow \sigma\}$$

a naïve attempt to derive a POINT from an OBJECT may be given by:

$$\text{POINT} = \text{OBJECT} \oplus \text{MOVEABLE}$$

This derivation is not especially useful, because the resulting POINT type is schizophrenic in its self-reference. For some *p* : POINT, the result type of *p.identity* is OBJECT, whereas the result type of *p.move(3,4)* is MOVEABLE. On inspection of the unrolled POINT type, it is clear why this is the case:

$$\begin{aligned}\text{POINT} = \{&\text{identity} : \text{OBJECT}, \text{equal} : \text{OBJECT} \to \text{BOOLEAN},\\ &x : \text{INTEGER}, y : \text{INTEGER},\\ &\text{move} : \text{INTEGER} \times \text{INTEGER} \to \text{MOVEABLE}\}\end{aligned}$$

In this naïve scheme, objects have "aggregate types" constructed from the set of extension record types inherited from their ancestors. Type correctness is judged in terms of whether a given function call can be typed in a member record of the aggregate. The disadvantage of this scheme is that any function returning *self* only has the type of a single member record from the aggregate. Further expressions involving this result will in all likelihood be impossible to check statically, since they will almost certainly involve functions declared in a different member record. All the *self*-types of the extension records are incomparable, since they own mutually exclusive functions.

4.2.2 A Subtyping Model of Inheritance

Trellis [SCBK86], *Sather* [Ohmo94, SOM93] and *Oberon* [RW92] respect subtyping in their inheritance rules. *C++* also has a variant of inheritance which corresponds to deriving a subtype incrementally [Stro91]. The naïve λ-calculus model of inheritance introduced above is now modified to capture this kind of pure subtyping (*cf* Cardelli's alternative *object calculus* model [Card92]).

The chief problem with the naïve model is that it aggregates object types from the unrelated types of "difference objects", the extension record types. To construct a model with subtypes, the type of the parent must be embedded inside the child type. Stroustrup means something like this when he says:

> "An object of a derived class has an object of its base class as a subobject" [Stro91, p183].

To achieve this in the λ-calculus model, we must avoid fixing the type of the extension record until we are ready to close the POINT type. One way of doing this is to define a generator ΦPOINT which binds the *self*-type of the extension record only in the result of record combination:

$$\begin{aligned}\Phi\text{POINT} = \lambda\sigma.(\text{OBJECT} \oplus \{&x : \text{INTEGER}, y : \text{INTEGER},\\ &\text{move} : \text{INTEGER} \times \text{INTEGER} \to \sigma\})\end{aligned}$$

Here, the simply-typed record combination operator $\oplus$ may still be used, since it combines records whose types are constant. The type POINT is created by fixing the generator ΦPOINT:

$$\text{POINT} = (\Upsilon\ \Phi\text{POINT})$$

$$\begin{aligned}= \mu\sigma.\{&\text{identity} : \text{OBJECT}, \text{equal} : \text{OBJECT} \to \text{BOOLEAN},\\ &x : \text{INTEGER}, y : \text{INTEGER},\\ &\text{move} : \text{INTEGER} \times \text{INTEGER} \to \sigma\}\end{aligned}$$

which, by unrolling the recursion, is equivalent to:

POINT = {identity : OBJECT, equal : OBJECT $\rightarrow$ BOOLEAN,
x : INTEGER, y : INTEGER,
move : INTEGER $\times$ INTEGER $\rightarrow$ POINT}

The type checking scheme offered by this approach is more useful than the fragmentary scheme above, since the child POINT's *self*-type now includes the parent OBJECT's *self*-type. Inherited functions returning *self* still only have the type of the parent: for some *p* : POINT, the call *p.identity* has the type OBJECT; however similar functions defined locally in the extension record have the type of the child, encompassing the parent: *p.move(3,4)* has the type POINT. This means that expressions having the child type POINT are also legitimate arguments to inherited functions: the call *p.move(3,4).identity* can be type-checked statically and has the type OBJECT. However, we still cannot check *p.identity.move(3,4)* statically, since *move()* is not a legal field of the OBJECT type.

4.2.3 Subtyping and Type Checking

It turns out that a calculus of inheritance based on pure subtyping is strictly less expressive than one would like. As a consequence of the *least fixed point* solution for the recursive type OBJECT, any type τ inheriting from OBJECT will obtain functions in the types:

identity : $\tau \rightarrow$ (OBJECT)
equal : $\tau \rightarrow$ (OBJECT $\rightarrow$ BOOLEAN)

While the *POOL* family [Amer87, Amer90] and *Emerald* [BHJL86, RL89] do not have inheritance in the same sense, they do support a type-compatibility between components based on subtyping. In these languages, it is impossible to propose any more specific types for *equal()* without violating contravariance. For example, assuming that the type relationship POINT $\subseteq$ OBJECT is desirable, it would be impossible to give POINT an *equal()* function with a covariant argument type:

equal : POINT $\rightarrow$ (POINT $\rightarrow$ BOOLEAN)

and still legally substitute POINT objects into program variables expecting an OBJECT. In order to ensure full *behavioural subtyping* [Amer90], it is only safe to assume that POINT equality has the type:

equal : POINT $\rightarrow$ (OBJECT $\rightarrow$ BOOLEAN)

This is a useful, but liberal typing scheme that strictly fails to capture the essence of strong typing. Such an *equal()* function could be used to compare POINTs with objects of any type $\tau \subseteq$ OBJECT. Elsewhere in *Emerald* and the *POOL* family, it is possible to provide more specific types for functions like *identity()*, since specialisation of the result is allowed by the covariant rule:

identity : POINT $\rightarrow$ (POINT)

In languages like *C++* [Stro91] and *Oberon* [RW92], one is not allowed to redefine the type of an inherited function. In any case, it would seem wasteful to have to reimplement *identity()* for each inheriting type. In these languages, functions are inherited with their result types unchanged:

identity : POINT $\rightarrow$ (OBJECT)

This again is rather too liberal: such an *identity* function could be used to map POINTs to objects of any type $\tau \subseteq$ OBJECT.

In a subtyping scheme, functions are only typechecked over their least upper bounds, so type information is lost when applying them to subtypes. This is a significant disadvantage. Type loss may occur even quite locally, leading to a need for dynamic type checks. For some *p* : POINT, an expression of the form:

p.identity.move(3, 4)

would require a dynamic type check on the result of *identity* to ensure that it was some type $\tau \subseteq$ OBJECT owning a function *move()*. The problem of type-loss is something endlessly debated in *C++* and leads to unsafe programming tricks to recover type, such as *downcasting* in the type hierarchy [Meys92, p135-142].

4.3 Bounded Universal Quantification

A better typing model would assert that, when applied to INTEGERs, *identity()* maps to INTEGERs; and when applied to BOOLEANs it maps to BOOLEANs [DT88]. This suggests some kind of polymorphic typing model, rather than a simple subtyping model. It is clear that a form of quantification over all types owning an identity and equality function is desired, so it is quite natural to consider typing the *identity()* and *equal()* functions using a bounded form of universal quantification:

identity : $\forall(\tau \subseteq \text{OBJECT}).\tau \rightarrow (\tau)$
equal : $\forall(\tau \subseteq \text{OBJECT}).\tau \rightarrow (\tau \rightarrow \text{BOOLEAN})$

The notion of bounded quantification was introduced by Cardelli and Wegner [CW85] in their experimental language *Fun*, as an extension to ordinary universal quantification. Universal quantification is used to define parametric polymorphic functions in the Girard-Reynolds style [Gira72, Reyn74].

4.3.1 Universal Quantification

Whereas the simply-typed λ-calculus [Chur40] is a *first-order* calculus with *term-abstraction*, the *second-order* λ-calculus [Gira72, Reyn74] also includes *type-abstraction*, to capture the intuitive concept of a function that takes a type as a parameter. *Universal quantification* is used to quantify over all types. For

instance, the identity and equality functions, which we defined over an OBJECT type above, can be given the alternative *universally quantified* types:

$$\text{identity} : \forall\tau.\tau \to \tau$$
$$\text{equal} : \forall\tau.\tau \times \tau \to \text{BOOLEAN}$$

in which τ may range over *any type whatsoever*. A universal polymorphic *equal()* function accepts as its first argument any type τ and returns the monomorphic *equal()* function that tests for equality among objects that are all of type τ.

In the ideal model [MS82, MPS84] it is possible to give meanings to functions with universal quantification $\forall\tau.\tau \to \tau$ based on a theory of infinite intersections in the domain V. In the following summary, the set of all types (ie set of all ideals in V) is known as *TYPE*, a lattice ordered by $\subseteq$ whose greatest element is TOP, a type containing exactly all the elements of V.

> "Let D $\bullet\!\!\to$ E be the set of total functions in V that map elements of the ideal D to elements of the ideal E. Stated more precisely:
>
> $$D \bullet\!\!\to E \equiv \{ f \in TOP \to TOP \mid x \in D \Rightarrow f(x) \in E \}.$$
>
> For any D and E that are ideals, this set is also an ideal and is therefore a valid type in the ideal model. There is an ideal (a type) in V that contains all total functions that map BOOLEANs to BOOLEANs, represented as BOOLEAN $\bullet\!\!\to$ BOOLEAN. ... *Identity* is in this ideal, so we write *identity* $\in$ BOOLEAN $\bullet\!\!\to$ BOOLEAN; but we could also write *identity* $\in$ INTEGER $\bullet\!\!\to$ INTEGER. In fact, for any type T, *identity* $\in$ T $\bullet\!\!\to$ T. Because identity is in all of these ideals, it is in their intersection, so we can write *identity* $\in \bigcap_{T \in TYPE} T \bullet\!\!\to T$ " [DT88, p54].

This is taken as the meaning of the expression $\forall\tau.\tau \to \tau$.

The Girard-Reynolds style of parametric polymorphism is readily adapted for use in an object-oriented context. For example, a list containing homogenous elements may be given the polymorphic type:

$$\text{LIST} = \forall\tau.\mu\sigma.\{\text{cons} : \tau \to \sigma,\ \text{head} : \tau,\ \text{tail} : \sigma\}$$

in other words, LIST is the recursive type with a *cons()*, *head()* and *tail()* function that are valid for all element types τ. In this construction, it is important to bind the universal variable τ before the recursion variable σ. This is so that LIST's recursion variable σ is bound with the element type τ in scope, ensuring that the tail of the LIST also contains further elements of type τ.

To say "LIST is a type" is rather loose. LIST is in fact a function from types to types, expecting as its first argument a type to replace the parameter τ. Applying LIST to the type INTEGER yields an INTEGER_LIST:

INTEGER_LIST = LIST [INTEGER]
= $\mu\sigma$.{cons : INTEGER $\rightarrow \sigma$, head : INTEGER, tail : σ}

In recognition of this, LIST is more correctly called a *type constructor*, since it maps elements of type τ into lists of type LIST [τ]. To obtain the more usual Girard-Reynolds style polymorphic types for LIST's functions, the universal quantifier is brought back outside the method selection function:

cons : $\forall\tau$.LIST [τ] $\rightarrow$ ($\tau \rightarrow$ LIST [τ])
head : $\forall\tau$.LIST [τ] $\rightarrow$ (τ)
tail : $\forall\tau$.LIST [τ] $\rightarrow$ (LIST [τ])

It should be obvious that this translation merely introduces externally a new universal type variable and then partially applies LIST to it.

Object-oriented languages have adopted this style of parametric polymorphism to abstract over internal components of a type. Parameterised class definitions are known in *Eiffel* as *generic classes* [Meye88] and in *C++* (from version 3.0) as *template classes* [Stro91]. A similar polymorphic mechanism existed earlier for the functional languages *ML* [Miln78], *Hope* [BMS80] and the generic packages of *Ada* [IBHK79], although the machinery necessary to describe *ML* is strictly simpler than the second order λ-calculus [CW85].

4.3.2 Bounded Quantification

Cardelli and Wegner introduced bounded quantification as a conservative extension to universal quantification [CW85]. *Bounded quantification* allows the definition of polymorphic functions over all types that are subtypes of a given type. Intuitively, one wants the functions *identity()* and *equal()* to be applicable polymorphically to the type OBJECT, or to some subtype:

identity : $\forall(\tau \subseteq$ OBJECT).$\tau \rightarrow (\tau)$
equal : $\forall(\tau \subseteq$ OBJECT).$\tau \rightarrow (\tau \rightarrow$ BOOLEAN)

The subtype constraint $\tau \subseteq$ OBJECT is the *bound*, or condition, restricting the actual type that may instantiate the parameter τ to a subtype of OBJECT. This is essentially all that distinguishes bounded quantification from universal quantification, in which no constraint is placed on parameters.

Bounded quantification finds an equally intuitive interpretation in the ideal model. Before, a denotation for the type of identity was given as: *identity* $\in \bigcap_{T \in TYPE} T \bullet\!\!\rightarrow T$, meaning that for all types T, identity is in the intersection of the ideals $T \bullet\!\!\rightarrow T$. As a consequence of the lattice structure of *TYPE*, an equivalent denotation for the type of identity is: *identity* $\in \bigcap_{T \subseteq TOP} T \bullet\!\!\rightarrow T$. By generalising over the syntax for universal quantification, we have

identity : $\forall\tau$. $\tau \rightarrow \tau \quad \Leftrightarrow \quad$ identity : $\forall(\tau \subseteq$ TOP) . $\tau \rightarrow \tau$

and it is this more general syntax that suggests bounded quantification in which the type TOP may be replaced by some other type in the lattice [DT88, p54], as in:

identity : $\forall(\tau \subseteq \text{OBJECT}).\tau \rightarrow \tau$

Universal quantification is a special case of bounded quantification, where the trivial bound $\tau \subseteq$ TOP is placed on the type that may instantiate the parameter.

Bounded quantification allows types to be given to many more parametric polymorphic functions than universal quantification, since constructed types may now depend on their constituents' possessing certain operations, something strictly prohibited in the Girard-Reynolds system[3]. It is possible to define a SORTED_LIST:

SORTED_LIST = $\forall(\tau \subseteq \text{COMPARABLE}).$
$\mu\sigma.\{\text{insert} : \tau \rightarrow \sigma,\ \text{head} : \tau,\ \text{tail} : \sigma\}$

such that the insertion function:

insert : $\forall(\tau \subseteq \text{COMPARABLE}).$
SORTED_LIST $[\tau] \rightarrow (\tau \rightarrow \text{SORTED\_LIST}\ [\tau])$

depends on the list element type having *at least* the comparison function < defined for COMPARABLE and all its subtypes. This is similar to the type system sketched in the experimental language *Russell* [DDS78, DD79] and also to the *constrained generic classes* offered in *Eiffel* (from version 3.0) [Meye92], in which an inheritance constraint (*conformance*, rather than *subtyping*) is placed on a type parameter:

class SORTED_LIST [T -> COMPARABLE]
feature ... **end**

These forms of polymorphism, though not central, are also of considerable interest in object-oriented programming. For reasons given later, the construction of types around bounded polymorphic variables is not as expressive as this first appears. Fortunately, this may be replaced by a slightly different construction.

Bounded quantification was originally intended to provide polymorphic types for inherited methods in object-oriented programming. Although it is explained below why this only partly succeeded, one useful insight gained from this work was that class types seemed to involve *type families constrained by a bound*. Cardelli called these *powertypes*, by analogy with powersets [Card88b] - a class has a polymorphic type which is the set of all subtypes of a given type, the least

---

[3] The claim that Girard-Reynolds polymorphic functions can be written "in ignorance" of the type instantiating the parameter starts to crumble the closer you get to the implementation [SC92].

upper bound. Other type treatments which only consider sets of identifiers adopt this approach [PS94].

4.3.3 System $F_{\leq}$ ("F sub")

Cardelli-Wegner bounded polymorphism has influenced the development of other type systems [Pier92a, SP94, Comp94]. The language *Fun* integrated ideas from Girard-Reynolds polymorphism [Gira72, Reyn74], a formulation of the second-order lambda calculus later to become known as "System F", with Cardelli's first order calculus of subtyping [Card84, Card88a] using formal techniques developed by Mitchell [Mitc88]. *Fun* was simplified and slightly generalised by Bruce and Longo, then by Curien and Ghelli [BL88, CG92]. Curien and Ghelli's latter formulation became known as "minimal bounded *Fun*" or $F_{\leq}$ ("F sub"), accepted as a standard for the bounded second-order λ-calculus. The first examples of bounded quantification were given in [CW85] and more were developed in Cardelli's study of power kinds [Card88b]. $F_{\leq}$ has been extended to include record types [CM92, Card92] and forms the basis for the programming language Quest [CL91]. A survey of $F_{\leq}$ and some related approaches is given in [Ghel90b].

Ghelli [Ghel90a] implemented a type checker for $F_{\leq}$ which he initially claimed was both sound and complete, although the algorithm presented in [CG92] was later found to be only semi-decidable. The approach taken was to compute the minimum type of a term and determine whether one type was a subtype of another. The algorithm converged only in cases where subtyping was provable, otherwise it could diverge. Pierce [Pier92b] eventually proved that the problem of determining whether one type was a subtype of another in $F_{\leq}$ is undecidable. Notwithstanding, Bruce *et al*. [BCMG93] developed a typechecker for a restricted language based on $F_{\leq}$ in which all terms could be guaranteed to have a minimum type, as a result of adding extra type information to class terms. Further decidable variants of $F_{\leq}$ include [CW85, KS92, CP94], all of which are subject to restrictions.

4.3.4 A Bounded Model of Inheritance

The original exposition of bounded quantification in [CW85] depended on motivating examples which, by chance, did not include any recursive types. Canning, Cook *et al*. [CCHO89a] later discovered that polymorphic functions quantified over recursive types could not be adequately described in the Cardelli-Wegner system. Some of their counter-examples are developed here. Bounded quantification does not provide the same degree of flexibility in the presence of type recursion as it does for non-recursive types. The process of taking fixpoints drives a wedge between the bounded quantified variable and the type's recursion variable. The effect is that functions lose their polymorphic types and can only be typed in their least upper bounds, making bounded quantification no more expressive than pure subtyping.

Consider a recursive type MOVEABLE, the common ancestor of several relocatable shapes such as SQUARE and CIRCLE. We would like a *move()* function:

$$\text{MOVEABLE} = \mu\, mv.\{\ \text{move} : \text{INTEGER} \times \text{INTEGER} \rightarrow mv,\ \ldots\ \}$$

to apply polymorphically to all descendants of MOVEABLE, such as SQUARE and CIRCLE. However *move()* does not have the type:

$$\text{move} : \forall(\tau \subseteq \text{MOVEABLE}).\tau \rightarrow (\text{INTEGER} \times \text{INTEGER} \rightarrow \tau)$$

but rather the type:

$$\text{move} : \forall(\tau \subseteq \text{MOVEABLE}).\tau \rightarrow (\text{INTEGER} \times \text{INTEGER} \rightarrow \text{MOVEABLE})$$

because the quantified variable τ and recursion variable *mv* are in fact independent. In the quantification $\forall(\tau \subseteq \text{MOVEABLE})$, the recursion variable *mv* is already out of scope. Inside the recursive definition, *mv* is bound by μ and therefore has been fixed at *mv* = MOVEABLE by the fixed point finder before τ ranges over this type. As a result, whenever we *move* SQUAREs or CIRCLEs we always obtain an object of exactly the type MOVEABLE (type information is lost). The algebra does not force the function's result type to mirror its polymorphic target.

Consider again that we would like COMPARABLE's comparison < function

$$\text{COMPARABLE} = \mu\, cp.\{\ < : cp \rightarrow \text{BOOLEAN},\ \ldots\ \}$$

to apply polymorphically to all descendants of COMPARABLE such as INTEGER and CHARACTER, which inherit the < operation. Now, the function < does not have the type:

$$< : \forall(\tau \subseteq \text{COMPARABLE}).\tau \rightarrow (\tau \rightarrow \text{BOOLEAN})$$

but rather the type:

$$< : \forall(\tau \subseteq \text{COMPARABLE}).\tau \rightarrow (\text{COMPARABLE} \rightarrow \text{BOOLEAN})$$

because μ fixes the type of *cp* = COMPARABLE before τ ranges over this type. As a result, whenever we compare INTEGERs, the < function always expects an argument of exactly the type COMPARABLE. The algebra does not force < to compare operands of the same type.

Looking at this case another way, we would like to consider CHARACTER or INTEGER as types possessing their own versions of <. Unrolling the inherited type definitions for CHARACTER or INTEGER, it is possible to redefine the function <, forcing it to accept an argument in the desired type:

$$\text{CHARACTER} = \mu\, ch.\{\ \ldots;\ \text{print} : ch \rightarrow;\ < : ch \rightarrow \text{BOOLEAN};\ \ldots\ \}$$

By explicit redefinition, < now has the type

$$< : \forall(\tau \subseteq \text{CHARACTER}).\tau \rightarrow (\text{CHARACTER} \rightarrow \text{BOOLEAN})$$

To ensure CHARACTER $\subseteq$ COMPARABLE, subtyping must obtain between the two < functions in the pair of records:

$$\{ \ldots; \text{print} : \text{ch} \rightarrow; < : \text{CHARACTER} \rightarrow \text{BOOLEAN}; \ldots \} \subseteq \{ < : \text{COMPARABLE} \rightarrow \text{BOOLEAN} \}$$

requiring COMPARABLE $\subseteq$ CHARACTER in turn by contravariance! The condition on which success depends is precisely the opposite of what this example intended to show. CHARACTER $\subseteq$ COMPARABLE cannot be derived using the rules of subtyping unless in fact CHARACTER = COMPARABLE. A recursive type only has proper subtypes if the recursion variable occurs on the result-side of functions. If the recursion variable occurs on the argument-side, this effectively prevents the derivation of a proper subtype.

Object-oriented class designs typically define recursive objects with *binary methods*, or functions expecting another argument of the same type, such as *equal()* or < above. Bounded quantification is strictly too weak a model to explain the polymorphic inheritance of such functions in object-oriented languages. To obtain polymorphism in the Cardelli-Wegner model would necessitate the elimination of recursive types; to retain recursive types would require sacrificing polymorphism [Simo94a].

4.4 Function-Bounded Quantification

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 [SC92]. Cardelli-Wegner bounded quantification does not have this property, being of the form:

$$\forall(\tau \subseteq \mu\sigma.F[\sigma]).e(\tau)$$

in which the type of the recursive variable is fixed before bounded quantification. Instead, we desire a form of quantification which permits full type recursion in the body of the type definition *and* in the expression denoting the type bound:

$$\forall(\tau \subseteq F[\tau]).e(\tau)$$

and this is called *function-bounded quantification* by Canning, Cook, Hill, Olthoff and Mitchell in their seminal paper [CCHO89a], or more commonly *F-bounded quantification*. An F-bound is a special kind of constraint which can be made to apply successfully to recursive types. Instead of insisting that a type is a straightforward subtype of another established type $\tau \subseteq \mu\sigma.F[\sigma]$, an F-bound

insists that a type is a subtype of an adapted recursive type, obtained by *applying a generator to itself* $\tau \subseteq F[\tau]$.

4.4.1 Deriving a Function-Bound

An F-bound describes family of types (a *kind*) which all have to satisfy the property $\forall(\tau \subseteq F[\tau])$, for some generator F. The derivation of this condition is the result of considering the intuitive types that we should wish polymorphic functions to have when they are inherited [CCHO89a, Simo94a]:

Consider again the polymorphic *move()* function. Working backwards, we seek the condition on a type *t* so that for any variable *x : t* we can derive "$\vdash$" that *x.move(1, 1)* is also of type *t*.

$$x : t \vdash x.move(1, 1) : t \qquad \{ \text{ by assumption } \}$$

Using two type rules for function application and record selection, *eg* [CW85, Card88b], this condition may be established. Below, Γ is the current set of type assumptions. For the purpose of the example, Γ can be ignored. Chaining backwards through the function application rule:

$$\text{APP} \quad \frac{\Gamma \vdash f : \sigma \to \tau \,,\ v : \sigma}{\Gamma \vdash (f\ v) : \tau}$$

yields the type which we would like to show *x.move()* to have:

$$x : t \vdash x.move : (\text{INTEGER} \times \text{INTEGER} \to t)$$

Chaining backwards through the record selection rule:

$$\text{SEL} \quad \frac{\Gamma \vdash r : \{\ \alpha_1{:}\tau_1, \ldots, \alpha_n{:}\tau_n\ \}}{\Gamma \vdash r.\alpha_i : \tau_i} \quad i \in 1..n$$

yields an upper bound on the record type that we would like to show *x* to have:

$$x : t \vdash x : \{\ move : \text{INTEGER} \times \text{INTEGER} \to t\ \}$$

This is only the minimal condition on the record type of *x*. Using the record subtyping rule, we can introduce many more record types τ with additional fields, such that *x.move(1,1)* delivers a result in the same type *t*:

$$\frac{\tau \subseteq \{\ move : \text{INTEGER} \times \text{INTEGER} \to t\ \}}{x : t \ \vdash\ x : \tau}$$

Essentially, we may legitimately derive that *x* has any of the types τ based on our original assumption. Since the type τ does not occur in any other

assumption, this may be simplified using the substitution { t/τ }, discharging our original assumption, yielding the requirement:

$$t \subseteq \{ \text{move} : \text{INTEGER} \times \text{INTEGER} \rightarrow t \}$$

which cannot be proved without additional assumptions. Expressing this condition as $t \subseteq \Phi\text{MOVEABLE}\ [t]$, where ΦMOVEABLE[4] is a *generator*, or type function for a recursive type:

$$\Phi\text{MOVEABLE} = \lambda\sigma.\{\text{move} : \text{INTEGER} \times \text{INTEGER} \rightarrow \sigma\}$$

it is clear that this condition fits the format for the kind of quantification desired.

Generators of the kind ΦMOVEABLE are central to F-bounded polymorphism. If the type MOVEABLE is understood to be the result of fixing the generator:

$$\text{MOVEABLE} = (\Upsilon\ \Phi\text{MOVEABLE})$$

the F-bound $\forall(t \subseteq \Phi\text{MOVEABLE}\ [t])$ expresses a constraint on the parameter *t* which insists that all types in the family have the same recursive structure as the type MOVEABLE and offer at least the functional interface of MOVEABLE. This captures exactly the notion of a class [CHC90].

4.4.2 Inheritance as Mutual Recursion

Inheritance is an incremental modification mechanism for recursive types. The recursive type POINT:

$$\begin{aligned}\text{POINT} = \mu\sigma.\{&\text{identity} : \sigma, \text{equal} : \sigma \rightarrow \text{BOOLEAN},\\ &x : \text{INTEGER}, y : \text{INTEGER},\\ &\text{move} : \text{INTEGER} \times \text{INTEGER} \rightarrow \sigma\}\end{aligned}$$

defines an *move()* function which returns a POINT. When inheriting from POINT, the recursive type HOT_POINT must modify *self*-reference in POINT's functions, such that *move()* returns a HOT_POINT instead. This suggests a map from a POINT to a HOT_POINT. Now, a HOT_POINT needs a modified *equal()* function to compare the additional *selected* attribute. However, to compare the *x* and *y* attributes, it might as well use the original *equal()* function defined in POINT. This suggests a map from a HOT_POINT to a POINT.

Cook and Palsberg [CP89] describe the "aha!" experience of discovering a denotational model for object-oriented inheritance which suddenly corresponds exactly to one's intuitions. The insight which grounds polymorphic inheritance in a sounder mathematics is an analogy with mutual recursion. Consider a

---

[4] The notation used in [CCHO89a] is "F-Moveable". Here and elsewhere we systematically use Φ to identify generators for recursive types.

function F and a derived (modified) version M which depends on F. In the direct derivation of M, the encapsulation of F is preserved:

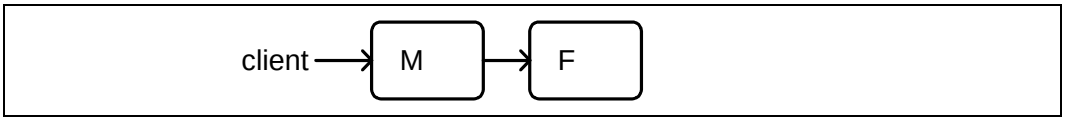

Figure 4.4: Direct derivation [CP89]

If F is now a simply recursive function, and M is derived from F such that recursive calls to F are not affected by M, the encapsulation of F is still preserved:

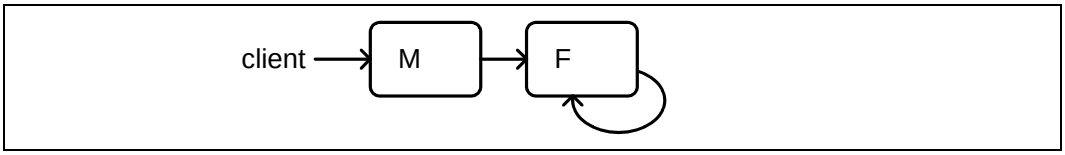

Figure 4.5: Naive recursive derivation [CP89]

because the modification only affects external clients, not recursive calls. Now, a derivation of M which is analogous to polymorphic inheritance is the following:

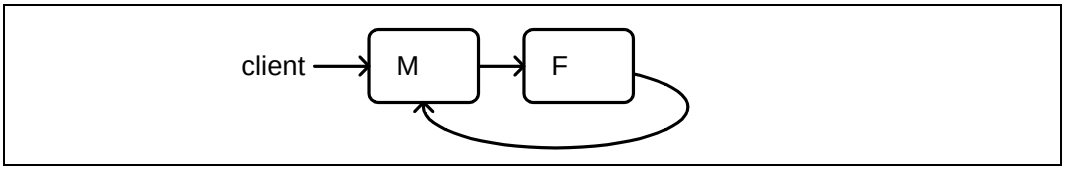

Figure 4.6: Derivation analogous to inheritance [CP89]

in which F is a *mutually recursive* function with M. Recursive calls to F are affected by the modification M, in exactly the same way that, under polymorphic inheritance, *self*-reference in the original type must be changed to refer to the modification. The dependency of M upon F reflects the way in which an inheriting type modifies the behaviour of the original type.

Inheritance seems to involve a map from types to types, in which recursion is preserved. There is a Category-theoretic machinery for solving recursive type equations [BW94, SG82] which can be extended to account for transformations in recursive structure, now that inheritance can be explained in terms of mutual recursion. F-bounded polymorphism seems to involve quantification over a family of functor-coalgebras [CCHO89a]. The map from subclass to superclass has also been described in terms of the operation of "forgetful" functors [Simo93].

4.4.3 A Function-Bounded Model of Inheritance

The authors in [CCHO89b] were the first to build "class wrapper" functions which have the property of modifying and extending type interfaces. The later article [CHC90] provides independent translation functions to extend objects, types and object-constructors separately, with mappings between levels. Chapters 5 and 6 will develop a combined typed model of inheritance which extends this work. Mitchell [Mitc90] has a slightly different calculus of linked record and type extensions.

Inheritance is a translation function that operates upon *classes*, rather than upon *types* individually. F-bounded quantification suggests that a class is a *family of types* associated with a type generator: $\forall(t \subseteq \Phi\text{OBJECT}\ [t])$, where

$$\Phi\text{OBJECT} = \lambda\sigma.\{\text{identity} : \sigma, \text{equal} : \sigma \rightarrow \text{BOOLEAN}\}$$

If this is the case, then a class is a true "generalisation" of the notion of type. Given a recursive type $\mu\sigma.F(\sigma)$, the corresponding generator F is found by abstracting over the point of recursion in the type. The class is then constructed by quantifying over all types that are subtypes of the type schema created by applying the generator to themselves: $\forall(\tau \subseteq F[\tau])$. In the reverse process, a type may be constructed from a class generator by taking the fixpoint: $\mu\sigma.F(\sigma) = (\Upsilon\ F)$.

The F-bounded model of inheritance may be described as an adaptation to the subtyping model of inheritance introduced above. Just as the subtyping model refused to fix the *self*-type of extension records, F-bounded inheritance also keeps the type of *self* in the parent class open, until it is ready to close the new type. This is achieved by extending a *generator*, rather than extending a *type*. For example, the generator for a point class ΦPOINT is derived by modifying the generator ΦOBJECT:

$$\begin{aligned}\Phi\text{POINT} &= \lambda\sigma.(\Phi\text{OBJECT}\ [\sigma]\ \oplus\ \{x : \text{INTEGER}, y : \text{INTEGER},\\ &\qquad\qquad \text{move} : \text{INTEGER} \times \text{INTEGER} \rightarrow \sigma\})\end{aligned}$$

$$\begin{aligned}&= \lambda\sigma.\{\text{identity} : \sigma, \text{equal} : \sigma \rightarrow \text{BOOLEAN}, x : \text{INTEGER}, y : \text{INTEGER},\\ &\qquad\qquad \text{move} : \text{INTEGER} \times \text{INTEGER} \rightarrow \sigma\}\end{aligned}$$

Here, the *self*-type of OBJECTs is redirected onto the new *self*-type σ of POINTs by applying the parent generator to σ: $\Phi\text{OBJECT}\ [\sigma]$. The result is a straightforward record type in which the old *self*-type has been replaced by σ. The simply-typed record combination operator $\oplus$ then combines two records in which σ refers uniformly to the *self*-type of POINTs in the result of record combination.

The recursive type POINT may be created by fixing the generator ΦPOINT:

$$\text{POINT} = (\Upsilon\ \Phi\text{POINT})$$

$$= \mu\sigma.\{\text{identity} : \sigma, \text{equal} : \sigma \rightarrow \text{BOOLEAN}, x : \text{INTEGER}, y : \text{INTEGER}, \text{move} : \text{INTEGER} \times \text{INTEGER} \rightarrow \sigma\}$$

which, by unrolling the recursion, is equivalent to:

$$\text{POINT} = \{\text{identity} : \text{POINT}, \text{equal} : \text{POINT} \rightarrow \text{BOOLEAN}, x : \text{INTEGER}, y : \text{INTEGER}, \text{move} : \text{INTEGER} \times \text{INTEGER} \rightarrow \text{POINT}\}$$

yielding exactly the result that we have been seeking: OBJECT's functions are retyped when they are inherited by POINT. Modifying generators for self-referential types captures precisely the notion of the anchored type *like Current* in *Eiffel* [Meye88], something previously recognised as an innovation in typing mechanisms [Cook89a, Cook89b]. From a formal point of view, inheritance allows a more flexible use of fixed points in the derivation of recursive types.

4.4.4 Type Checking using F-Bounds

Object-oriented programs can be type-checked using this approach. Inheritable functions are now polymorphically typed using F-bounded parameters standing for the *self*-type. Consider a class of relocatable objects, $\forall(\tau \subseteq \Phi\text{MOVEABLE}[\tau])$, whose associated generator ΦMOVEABLE is defined as:

$$\Phi\text{MOVEABLE} = \lambda\sigma.\{\text{move} : \text{INTEGER} \times \text{INTEGER} \rightarrow \sigma\}$$

We consider ΦMOVEABLE to be an F-bounded function from types to types. When applied to some legal type $\tau \subseteq \Phi\text{MOVEABLE}[\tau]$, it will generate a truncated record type $\Phi\text{MOVEABLE}[\tau]$, a schema bearing a similarity to the proper recursive type MOVEABLE. Bringing the quantifier back outside the method selection function, we obtain the polymorphic type of *move()*:

$$\text{move} : \forall(\tau \subseteq \Phi\text{MOVEABLE}[\tau]).\tau \rightarrow (\text{INTEGER} \times \text{INTEGER} \rightarrow \tau)$$

If we apply this polymorphic *move()* to a type, say SQUARE or CIRCLE, we generate monomorphic functions for moving SQUAREs or CIRCLEs:

$$\text{move}[\text{SQUARE}] : \text{SQUARE} \rightarrow (\text{INTEGER} \times \text{INTEGER} \rightarrow \text{SQUARE})$$

$$\text{move}[\text{CIRCLE}] : \text{CIRCLE} \rightarrow (\text{INTEGER} \times \text{INTEGER} \rightarrow \text{CIRCLE})$$

which is exactly the result we seek. A *move()* function will acquire progressively more constrained polymorphic types in inheriting classes, and will acquire an exact monomorphic type when used in a static type context.

In fact, *move()* is properly typed *only* for the type family $\forall(t \subseteq \Phi\text{MOVEABLE}[t])$, emphasising the fact that it "belongs" to the class. We may verify this by applying ΦMOVEABLE to any of these types, say SQUARE and CIRCLE:

ΦMOVEABLE [SQUARE]
= { move : INTEGER $\times$ INTEGER $\rightarrow$ SQUARE }

ΦMOVEABLE [CIRCLE]
= { move : INTEGER $\times$ INTEGER $\rightarrow$ CIRCLE }

and then comparing the resulting instantiations of the ΦMOVEABLE schema with the unrolled recursive types we would like to give to SQUARE and CIRCLE:

SQUARE = μ sqr.{ ...; move : INTEGER $\times$ INTEGER $\rightarrow$ sqr; ... }
= { ...; move : INTEGER $\times$ INTEGER $\rightarrow$ SQUARE; ... }

CIRCLE = μ cir.{ ...; move : INTEGER $\times$ INTEGER $\rightarrow$ cir; ... }
= { ...; move : INTEGER $\times$ INTEGER $\rightarrow$ CIRCLE; ... }

in order to see from the subtyping calculus that the F-bound holds in each case, since the recursive types SQUARE and CIRCLE have strictly more fields:

SQUARE $\subseteq$ ΦMOVEABLE [SQUARE]

CIRCLE $\subseteq$ ΦMOVEABLE [CIRCLE]

As we would expect, *move()* is not *universally* polymorphic. An example of a type to which *move()* may not apply is OBJECT. By unrolling the type of OBJECT we see that it does not possess a *move()* field and therefore the required subtyping relationship with the ΦMOVEABLE schema does not obtain:

OBJECT $\not\subset$ ΦMOVEABLE [OBJECT],

because:

{identity : OBJECT, equal : OBJECT $\rightarrow$ BOOLEAN}
$\not\subset$ {move : INTEGER $\times$ INTEGER $\rightarrow$ OBJECT}

In fact, the most general type satisfying the F-bound is the type over whose body we abstracted:

MOVEABLE $\subseteq$ ΦMOVEABLE [MOVEABLE],

because:

MOVEABLE = ΦMOVEABLE [MOVEABLE],

by the fixed point theorem. In contrast with the Cardelli-Wegner model, we do not have any other *simple subtyping* relationships:

SQUARE $\not\subset$ MOVEABLE

CIRCLE $\not\subset$ MOVEABLE

since subclassing is not subtyping [CHC90]. *Type inheritance* is nonetheless a well-formed mathematical relation, whose properties can be induced from simple subtyping. The kinds of type checking algorithms that must be used are different from those used in subtyping schemes. The first examples of experimental languages with F-bounded type parameters standing for the *self*-type include *Abel* [Harr91] and Bruce's series [Bruc94, BCMG93, BSG94]. Type-checking schemes for these are still appearing. Bruce's *matching* rule [Bruc94] expressing the type-compatibility under inheritance, is discussed by Abadi and Cardelli in [AC95]. The Johns-Hopkins group perform a series of explicit translations [ESTZ94, EST95] to convert programs into a form where soundness and completeness are obtained.

4.4.5 System $F^{\omega}_{\leq}$ ("F omega sub")

Explaining the behaviour of F-bounds requires at least a second-order λ-calculus; modelling the translation-functions from F-bound to F-bound would seem to require a higher-order *kinded* calculus, mapping from type-constructors to type-constructors and types. Whereas Bruce [Bruc94] treats an F-bound as just a "funny kind of bound", quantifying over types in the usual way, Abadi and Cardelli [AC95] suggest that this may entail weaknesses and recommend a higher-order approach.

Higher-order models follow the work of Girard [Gira72], whose System F^{ω} is a typed lambda calculus with higher-order polymorphism. F^{ω} ("F-omega") includes the *term abstraction* of the simply typed lambda calculus [Chur40], the *type abstraction* of the second-order lambda calculus [Gira72, Reyn74] and the possibility of *type operators* mapping from kind to kind. A *kind* is the extra level introduced to describe the spaces inhabited by ordinary types and type operators. Thus, F^{ω} is higher-order rather than third-order. Various extensions to F^{ω} have been proposed, based mainly on Cardelli's power *kinds* [Card88b], to include higher-order subtyping. The extension of the subtype relation to type operators in $F^{\omega}_{\leq}$ ("F-omega-sub") was developed formally by Cardelli and Mitchell [Card90, Mitc90] although the same intuitions are present in the earlier works [CCHO89a, CCHO89b, CHC90]. Whereas Pierce and Turner proceeded to model objects using existential types instead of recursive types in $F^{\omega}_{\leq}$ [PT92, PT93], Compagnoni and Pierce [Pier92a, CP93] gave an extension to $F^{\omega}_{\leq}$ to include finitary intersection types and Bruce and Mitchell developed a more powerful model including recursive types [BM92]. This model turned out to describe the F-bounded quantification of Cook *et al*. [CCHO89a, CCHO89b, CHC90] which was originally thought to be a variant of $F_{\leq}$ but which actually may be constructed to reflect the kind of higher order polymorphism necessary to deal properly with the interactions between subtyping and type recursion.

While the second-order fragment $F_{\leq}$ of $F^{\omega}_{\leq}$ had been studied in detail, relatively little was known about ω-order calculi until recently. The analysis of $F^{\omega}_{\leq}$ was expected to be significantly more challenging than that of $F_{\leq}$; and questions regarding the decidability of subtyping and typechecking were

completely open. Compagnoni and Pierce [CP93, Comp94] have extended their calculus of intersection types, known as $F^{\omega}_{\wedge}$ ("F-omega-meet") and prove that subtyping in $F^{\omega}_{\wedge}$ is decidable, which *a fortiori* gives the decidability of subtyping in the $F^{\omega}_{\leq}$ fragment. Independently, Steffen and Pierce [SP94], by eliminating the "cut rule" of transitivity from the subtype relation in $F^{\omega}_{\leq}$, have proven the soundness, completeness and termination of algorithms for subtyping and typechecking.

4.4.6 Towards a Theory of Classification

A λ-calculus framework has been presented, within which different models of *type inheritance* were compared. These included a naïve model, a subtyping model, a model using bounded quantification and a model using F-bounded quantification. For reasons of brevity in the exposition, the λ-calculus presentation was restricted to *type functions* and *type generators*. This was apposite, since the focus of our treatment was to emphasise the points of similarity and difference between alternative *typings* of inheritance. Clearly, a fuller treatment must also take into account the modifications to object structure achieved by *implementation inheritance*.

The next chapter develops a full model of objects, classes and polymorphic inheritance. Unlike the separate treatments of type and implementation given in *TOOPL* [Bruc94] and *PolyTOIL* [BSG94], objects are to be linked directly to their types, in the style proposed by [CHC90]. The implementation and type constraints of the class hierarchy are to be combined in a single model. The goal is to provide a complete theory of classification.