Chapter 3

Object Types and Subtypes

This chapter explores what it means for an object to have type.

Two main schemes for describing object types are introduced. The first is the algebraic method for characterising abstract types using existential quantification. The second relies on primitive constructions in the typed lambda calculus, modelling objects as closures. A key focus of interest is to discover under what conditions an object having one type may safely be bound to a variable expecting a different type. Subtyping is considered as a potential model for classifying object types, offering a rudimentary kind of polymorphism.

3.1 Abstract Types

Types have for long been used to reason about the formal properties of computer programs. What is a type? There is more than one possible view. Types may be considered minimally as schemas for interpreting bit-strings in machine memory. This is a *concrete* view, readily accepted by programmers working close to implementations. For example, the bit-string:

01000001

is 'A' if interpreted as a CHARACTER; or else 65 if interpreted as an INTEGER. In this view, type schemas are tied closely to the values (bit-strings) they interpret. This approach, while practical, is of limited formal use since it does not promote mathematical reasoning about type.

3.1.1 Constructive and Algebraic Approaches

The common mathematical treatments of type break down into the *constructive* approaches, exemplified by the intuitionistic proofs of Martin-Löf [Mart80], and

the *algebraic* approaches, exemplified by Goguen's OBJ family of specification languages [FGJM85].

In constructive type theory, types are modelled by sets of values. Program operations are modelled as functions and relations acting on these sets. Model-based specification methods such as VDM [Jone86] and Z [Spiv88, Spiv89] follow this approach up to a point. Their common intuition is that programs whose formal properties are not known may be modelled in terms of simple mathematical constructs whose properties are known. But whereas Z might use a non-computable equation to define a function (*eg* specifying the square root using its inverse $y = x^2$), a constructive proof must have a computable algorithm (such as the Newton-Raphson method). This is immediately appealing, since constructive proofs can always be implemented. A disadvantage is that certain desired constructive proof procedures may not exist; or else proofs may be so constrained by the need for computability that they distract from the theorem they assert (*eg* is it immediately obvious that the Newton-Raphson algorithm computes a square root?)

Algebraic type theory [Reyn74, Reyn75, Gutt75, Gutt77, GH78] strives to distance itself as much as possible from concrete and model-based notions. A mathematical type - known as an *algebra* - is a pair of a *sort* and a set of functions acting on the sort. For example, we might represent a BOOLEAN algebra as:

$$\text{BOOLEAN} = \langle \text{BOOL}, \{\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow\} \rangle$$

The notion *sort* will be defined in a moment. The behaviour of the type is represented chiefly by the functions, and the meaning of the type is represented by equations linking certain invocations of these functions. This has the advantage of being an entirely syntactic representation of type, not tied to any representation.

In model-based approaches, a system is represented using state variables standing for sets and sequences which are subject to modification. Individual operations are defined in terms of their effects on the state model. Proof procedures have a local validity, stated in terms of pre- and post-conditions to individual operations. In algebraic approaches, the state of the system is recorded as some sequence of function applications. It is easier to trace the series of operations that led to a given system state.

3.1.2 Sets, Sorts and Carriers

Reynolds has argued strongly [Reyn83] for the separate existence of types apart from computation, a view which we endorse since it concurs with our opinion that typing relates to other kinds of human classificatory activity. Types are not constructed on sets of values, which would tie them too specifically to one domain in computation; rather they are "syntactic disciplines for enforcing levels of abstraction".

Earlier, Morris [Morr73] showed that although it may often seem convenient to model types in terms of sets:

$$x : T \Leftrightarrow x \in T$$

in principle a type can be represented, or implemented, by a variety of sets; in this sense a particular set of values is insufficient to characterise a type. Consider two alternative representations for a simple ordinal type:

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

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

and it is clear that either set is an appropriate carrier for the type - no one set deserves preeminence. Furthermore, an early concrete representation for a type may introduce unwanted concerns: is the set finite or infinite; and therefore how do operations act on the least or greatest elements of the type? More precisely, there is no such thing as *the* set of simple ordinals; rather the *sort* ORDINAL denotes an abstract collection of objects that can be realised by a variety of *carrier sets*. We can think of a sort as:

> "an uninterpreted identifier that has a corresponding carrier in the standard (initial) algebra" [DT88], p52.

A sort has the force of an abstract set; its name acts as a syntactic placeholder awaiting the full definition of a type - in this sense it is an uninterpreted identifier. Most mathematical theories of types construct primitive *domains* [Scot76, Stoy77] of values, in which carrier sets exist for certain basic sorts. The standard algebra usually contains carriers for the natural number and boolean sorts:

$$NAT \leftrightarrow \{0, 1, 2, ...\}$$
$$BOOL \leftrightarrow \{0, 1\}$$

Each element in the carrier stands for an abstract object in the sort. An *initial* algebra is one whose constructions map onto all other algebras in the same family - an important property for the carrier to have universal validity. This is a category-theoretic concept [BW94, SG82] which is not discussed further here.

3.1.3 Function Signatures and Axioms

Abstracting away from concrete sets does not yet say anything precise about the way in which the ORDINAL type behaves. If we choose NAT as the sort on which to base this type, it is clear that the carrier set has strictly more properties (such as addition) than we require. It is more usual to define ORDINAL as the *abstract type* over which the functions *first()* and *succ()* are meaningfully applied:

ORDINAL = $\exists$ ord.{first : $\rightarrow$ ord, succ : ord $\rightarrow$ ord}

This specifies the external behaviour of the type as a set of functions. The type definition is *abstract*, in the sense that it characterises its member objects only down to the level of detail that they may be generated or manipulated using *first()* and *succ()*. Such a definition is usually regarded as being existentially quantified [MP85], because the representation type is hidden. In the body of the definition, the token *ord* is a syntactic placeholder for some set of values. Any sort whose carrier set offers the necessary properties may be used, such as:

ORD $\leftrightarrow$ {0, 1, 2, ...}
ORD $\leftrightarrow$ {a, b, c, ...}

It is common to categorise the functions of an abstract type as *observers*, *constructors* and *extenders* [GH78]. Observer functions simply inspect the type. Extender functions describe its more elaborate behaviour, but do not generate any new or unique objects. The constructor functions are a special category, because it must be possible to generate every member object of the type using them. Both *first()* and *succ()* are constructors.

Function signatures alone are insufficient to characterise abstract types. The intended meanings of *first()* and *succ()* are not yet captured through the syntactic discipline enforced by their type signatures. Function applications might yield the following valid, albeit undesirable, results:

succ(1) = 1
succ(b) = first() = a

To express the sequentially-ordered property ORDINAL objects, logical axioms are needed to describe the semantics of functions operating on objects of the type. In this case, the following axioms:

$\forall$(x : ORDINAL).succ(x) $\neq$ x $\wedge$ succ(x) $\neq$ first() $\wedge$
(succ(x) = succ(y) $\Leftrightarrow$ x = y)

plus the principle of induction are exactly enough to ensure that the type behaves like an ORDINAL. All potential object-object transitions are illustrated in the state diagram in figure 3.1, where those transitions actually permitted by the axioms are shown in bold, yielding a monotonic sequence of abstract objects x_i:

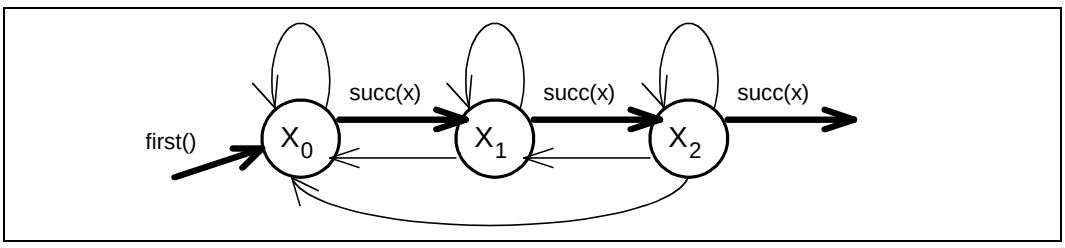

Figure 3.1: State diagram for ORDINAL

The subscript number has no order-significance, though for convenience the distinct objects x_i are indexed in order of generation. The economy of this description is admirable. The axioms governing *succ()* clearly rule out self-transitions and regression to the first object; and the principle of induction takes care of the rest. For example, *succ()* cannot map x_2 back to x_1, since by unrolling applications of *succ()* this would violate $succ(x) \neq first()$. As a result, *first()* will always generate a unique object and successive applications of *succ()* are guaranteed to generate a new and distinct object each time.

The algebra ORDINAL is therefore a pair of a sort and a set of functions whose meaning is given by axioms:

$$ORDINAL = \langle ORD, \{first : \rightarrow ORD, succ : ORD \rightarrow ORD\}\rangle$$

The constructor functions will generate every object of the type. Once the algebra is defined, the carrier used to model the sort can be disregarded. This is because it is now possible to represent every member of the type in a purely syntactic way:

```
first();  succ(first());  succ(succ(first())); ...
```

and these abstract descriptions are fully amenable to mathematical reasoning. Proof procedures merely have to rely on syntactic pattern-matching in order to perform substitutions and other syntactic manipulations. Algebraic data types are both more general and more exact than concrete types. The ORDINAL type shown here is inhabited by a monotonic sequence of abstract objects; and that is exactly all that the specification expresses.

3.2 Object Types

Here, we investigate ways of modelling the types of objects using similar techniques. Objects have the properties of *identity*, *state* and *behaviour*. When modelling the types of objects, the most important property to capture is object behaviour, since we need to know how an object will react in a given situation when it is sent messages. Type compatibility among objects depends crucially on their having conformant behaviours.

3.2.1 Encapsulation of Methods and State

Perhaps the most immediate syntactic distinction between the manipulation of ordinary values and objects is that object functions (known as *methods*) must be selected before they can be applied. This is because objects encapsulate their methods along with their state. Whereas in a conventional imperative language, moving a simple integer point is accomplished using a free-standing function:

```
p : INTEGER_POINT;
movePoint(p, 3, 4);
```

called *movePoint()*, to which we may give the functional type:

$$\text{movePoint} : \text{INTEGER\_POINT} \times \text{INTEGER} \times \text{INTEGER} \rightarrow \text{INTEGER\_POINT}$$

in an object-oriented language we must select a method:

```
p : INTEGER_POINT;
p.move(3, 4);
```

called *move()*, from the point object and apply it to that object's state. The *move()* method considered alone has the functional type:

$$\text{move} : \text{INTEGER} \times \text{INTEGER} \rightarrow \text{INTEGER\_POINT}$$

alternatively, the type of the expression as a whole is considered to be:

$$\text{INTEGER\_POINT} \rightarrow (\text{INTEGER} \times \text{INTEGER} \rightarrow \text{INTEGER\_POINT})$$

Depending on the context, one or other notation will be preferred. The longer notation takes into account the object from which the method is selected and therefore the type of the selection-function is modelled as well. The shorter notation assumes that the owning object is already in scope. It is also important to observe the correct proprieties when accessing an object's state. Either the state has to be in scope, or else an argument must be supplied representing the object's state.

3.2.2 Existential Types and Functional Closures

Reynolds [Reyn75, Cook91] identified two complementary approaches to modelling object encapsulation: *procedural abstraction*, which relies on hiding state in private variables common to a collection of procedures, and *type abstraction*, which reveals the existence of state externally but prevents illegal access to it by hiding its type.

The latter approach is especially of interest to those investigating existential types [MP85], those wishing to trace the mappings between abstract types and concrete representation types in order to model the packaging mechanisms of languages like *Ada* [CW85] and those wishing to do without recursive types [PT92, HP92]. In these schemes, a simple integer point type is modelled using *existential quantification* to protect the state from external access:

$$\begin{aligned}&\text{INTEGER\_POINT} = \exists\ \text{Rep}.\{\ \text{state} : \text{Rep};\ \ \text{methods} : \{\\ &\quad \text{new} : \rightarrow \text{Rep},\\ &\quad \text{x} : \text{Rep} \rightarrow \text{INTEGER},\\ &\quad \text{y} : \text{Rep} \rightarrow \text{INTEGER},\\ &\quad \text{move} : \text{Rep} \times \text{INTEGER} \times \text{INTEGER} \rightarrow \text{Rep}\ \}\ \}\end{aligned}$$

The type is declared as a pair of *state* and *methods*, which is represented here as a record. By a process of *unpacking* [CW85], the abstract state may be bound to any suitable representation type. An advantage of this approach is

that types are not recursive [PT92], since the methods act on the abstract state, rather than on the type itself. A disadvantage is that method invocation requires a cumbersome syntax to access state:

```
p : INTEGER_POINT
p.methods.move(p.state, 3, 4);
```

and for this reason, the alternative approach is preferred here.

The former line of research initiated by [Card84, Redd88, Cook89a] models simple objects as *functional closures*, encapsulating the state of objects in private variables accessible only through their methods. In these schemes, objects are invariably elements of recursive record types [Mitc90, BL90], whose existence is established through the fixed point theory of recursion. The same point type illustrated above is usually encoded as:

$$\begin{aligned}&\text{INTEGER\_POINT} = \mu\ \text{pnt}.\{\\ &\quad \text{new} : \rightarrow \text{pnt},\\ &\quad \text{x} : \rightarrow \text{INTEGER},\\ &\quad \text{y} : \rightarrow \text{INTEGER},\\ &\quad \text{move} : \text{INTEGER} \times \text{INTEGER} \rightarrow \text{pnt}\ \}\end{aligned}$$

in which μ is used to bind the recursion variable, *pnt*. We shall expound this in more detail below, since significant problems arise when dealing with type recursion.

A *closure* is defined as a function with an environment [ASS85, Redd88]. Functions are lexically scoped, making it possible to set up an environment by binding free variables at the time of function definition. A simple example of a function with updatable state is *store()*:

$$\begin{aligned}&\textbf{let}\ \text{state} = 0\ \textbf{in}\\ &\quad \text{store} = \lambda x.(\textbf{if}\ x = 0\ \textbf{then}\ \text{state}\ \textbf{else}\ \text{state} := x)\end{aligned}$$

$$\begin{aligned}&\text{store}(5) \Rightarrow 5; \qquad \text{store}(0) \Rightarrow 5;\\ &\text{store}(2) \Rightarrow 2; \qquad \text{store}(0) \Rightarrow 2;\end{aligned}$$

Calling *store()* with 0 inspects the state; calling with any other value updates the state. The advantage of this approach is that the state is completely hidden inside the function. No complicated invocation syntax is needed; simply a value which serves as a tag or label to select one or other operation. A function from labels to operations is exactly what we need to model objects with selectable methods.

3.2.3 Objects as Records

A simpler interpretation of this model, due to [Wand87] and used by Cook *et al.* [Cook89a, CCHO89a, CCHO89b, CHC90] to model certain behavioural properties of objects, ignores the issue of mutable state altogether. Again, objects are modelled as records of functions, representing methods. Access to

state attributes is handled as the invocation of nullary functions (*ie* functions accepting no arguments) and state update is bypassed by creating new objects, thereby ensuring a straightforward pure functional calculus.

The sources cited above take for granted that a record can be modelled in the λ-calculus as a finite function from labels to methods. A simple λ-calculus strategy for constructing records is to build tuples of values, protected by a λ-abstraction:

make_point = λa.λb.λf.(f a b)

p = (make_point 3 4) ⇒ λf.(f 3 4)

Using the technique of partial application, *make_point()* is applied to strictly fewer arguments than it expects. The resulting function, *p*, is a closure binding two values 3 and 4. The values are protected by the remaining λ-abstraction λ*f*. This is the standard representation of a pair in the pure λ-calculus. To release one or other value, it is necessary to apply *p* to projection functions:

x = λa.λb.a -- *first projection*
y = λa.λb.b -- *second projection*

(p x) = (λf.(f 3 4) x) ⇒ (x 3 4) = (λa.λb.a 3 4) ⇒ (λb.3 4) ⇒ 3

(p y) = (λf.(f 3 4) y) ⇒ (y 3 4) = (λa.λb.b 3 4) ⇒ (λb.b 4) ⇒ 4

The symbol "=" denotes the syntactic replacement of an abbreviation by its full λ-calculus form and "⇒" denotes one level of β-reduction (*ie* function application). It should be obvious that the application (*p x*) mimics exactly the kind of record field selection process required. The record braces can be considered a shorthand for the equivalent λ-calculus:

p = {x ↦ 3, y ↦ 4} ⇔ p = λf.(f 3 4)

p.x ⇔ (p x)
p.y ⇔ (p y)

An object having *n* fields can be modelled by an *n*-tuple in the λ-calculus, having *n* associated labels which are modelled as the *n* different projection functions. A serious restriction is that labels and tuples must have the same arity - projection functions are built in the knowledge of how many arguments they must consume. This will present a problem when polymorphism is introduced, since the calculus will expect to use labels with objects of different size. To get around this problem, a different construction in the λ-calculus must be adopted for records, for which selection is independent of record length or field-order. This is easily done using pairs to build linked lists and associative maps. Such a technique is described in Appendix 1.

3.2.4 Recursive Record Types

An object is modelled as a finite map from labels to functions, representing its methods. The type of an object is a record type, that is, a finite map from (the same) labels to function signatures, representing the types of its methods.

A simple non-recursive point object representing the coordinate (3, 4) and allowing access to its *x* and *y* values is modelled as a typed record of functions:

$$\{x \mapsto \lambda u.3,\ y \mapsto \lambda u.4\} \ :\ \{x : \mathrm{UNIT} \to \mathrm{INTEGER},\ y : \mathrm{UNIT} \to \mathrm{INTEGER}\}$$

The nullary (zero-argument) access methods to the fields *x* and *y* are modelled as functions accepting the dummy value *unit* which is the sole element of the trivial type UNIT. This is to observe the syntactic conventions of λ-functions, which require exactly one argument. Hereafter, neither *unit* nor its type will be notated, since their presence can be reconstructed automatically:

$$\{x \mapsto 3,\ y \mapsto 4\} \ :\ \{x : \mathrm{INTEGER},\ y : \mathrm{INTEGER}\}$$

An object's methods typically refer to each other. To demonstrate this, the above point is now provided with an equality-testing method, *equal()*, from which calls are made to the *x* and *y* methods:

$$\mu\ \mathrm{self}.\{x \mapsto 3,\ y \mapsto 4,\ \mathrm{equal} \mapsto \lambda \mathrm{other}.(\mathrm{self}.x = \mathrm{other}.x \wedge \mathrm{self}.y = \mathrm{other}.y)\}$$

$$:\ \mu\ \sigma.\{x : \mathrm{INTEGER},\ y : \mathrm{INTEGER},\ \mathrm{equal} : \sigma \to \mathrm{BOOLEAN}\}$$

The extended point is recursive at both the term- and type-level. To handle the recursion, we introduce two variables: *self* to stand for the whole object and σ to stand for the whole *self*-type, and we bind these using μ. Note that *self* and σ occur in different places in the object and type definitions. Although the type σ must clearly depend on the form of *self*, self-reference at term- and type-level are essentially independent.

It is not immediately apparent that recursive values and types can be proven to exist. This is because a recursive definition merely expresses an equation which some suitable value should satisfy - there might be no solution, or else there might be many solutions, *cf* the roots of a polynomial. The standard approach to providing recursive types with a denotational semantics appeals to Scott's domain theory [Scot76, Stoy77] in which partial orders are constructed among sets of values in the domain V of all computable values:

$$V = \mathrm{BOOLEAN} + \mathrm{NATURAL} + [V \times V] + [V \to V].$$

Certain sets of values in this domain, known as *ideals*, have the property of being downward closed and consistently closed under a complete partial ordering relationship ⊆, usually interpreted to mean *less defined than*. Ideals form useful carriers for recursive types [MS82, MPS84] since they yield a set-theoretic interpretation, in which induction is well-founded and recursive type equations have solutions [DT88]. The fixed point theory of recursion explains

how we can find solutions, called *fixpoints*, to recursive equations and pick out a unique, most natural one, namely the least defined one [Read89]. Such a fixpoint is then taken to be the real meaning intended by the recursive description, the least fixpoint corresponding to the most abstract denotation of the type.

In order to model recursive objects and types, we need to build domains such that fixpoints may be taken at both term- and type-level [Mitc90, BL90]. Bruce and Mitchell have constructed partial equivalence relation (PER) models which support this [BM92]. The only restriction is that an object may not contain a field with the value *self*. This is an infinite construction which defeats the convergence theorem for fixpoints. Fortunately, the fields of our objects consist only of functions and convergence is guaranteed under these conditions.

3.3 A Calculus of Subtyping

A major group of object-oriented languages [Stro91, Meye88, SCBK86] have identified the notion of *class* with *type* and *subclassing* with *subtyping*. Clearly, the possibility of a direct translation of object-oriented concepts into standard type-theoretic constructs is appealing. Others [Snyd86a, Amer90] have postulated the independent existence of *type hierarchies*. In both approaches, subtyping is used to provide a rudimentary model for polymorphic binding. A key notion is the oft-quoted *substitutability criterion* [CW85, Amer90], whereby an object of one type may be safely passed to a variable expecting a different type. Most work in this area originates in Cardelli and Wegner's first-order theory of subtyping [Card84, CW85, Card88a, Card88b], the precursor to all the later second- and higher-order theories. We explore here their mechanisms for determining syntactic subtypes and later make some additions of our own [Simo94b] to describe the effects of adding axioms to the model.

3.3.1 Subtyping for Set Types

According to Cardelli and Wegner:

> "a type A is included in, or is a subtype of another type B when all the values of type A are also values of B, that is, exactly when A, considered as a set of values, is a subset of B" [CW85, p508].

It is clear that subtyping in this model is identified with the subset relationship - the assertion above is the axiomatic definition of a subset:

$$\sigma \subseteq \tau \;\Leftrightarrow\; \forall x\,(x \in \sigma \Rightarrow x \in \tau) \qquad \text{[Rule 0: subtype is subset]}$$

This allows the construction of a complete partial order (CPO) relating all types (ie sets) in a lattice. To illustrate this, if the domain of types is the powerset P ({a, b, c}) then there exists a lattice ordering all these types under the relation $\subseteq$ as in figure 3.2:

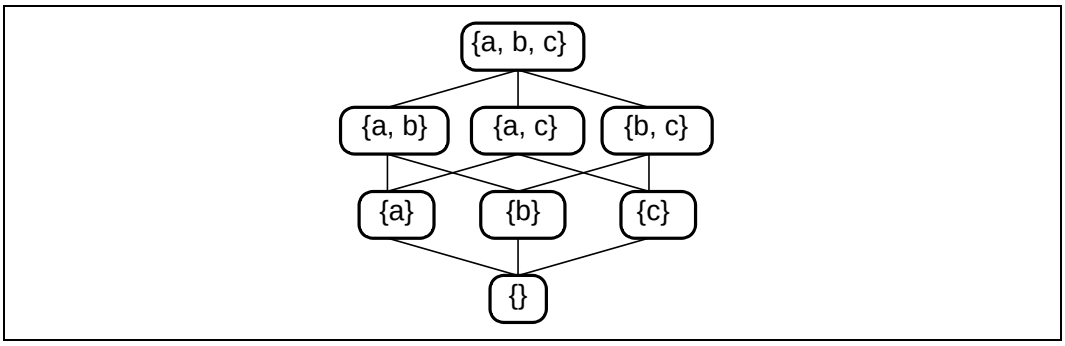

Figure 3.2: CPO for Set Types

which is appealing, in that it models a multiple specialisation type hierarchy reminiscent of the "multiple inheritance" found in [Moon86, Stro87].

3.3.2 Subtyping for Subrange Types

Given the types BOOLEAN and NATURAL, subrange types *s..t* may be constructed, where $s \in$ NATURAL; $t \in$ NATURAL; and the ordering $s \leq t$ holds. The set of all subranges has a useful partial order $\subseteq$ among its elements: if we approximate the limits of NATURAL as 1..3, this yields a partial order (PO) ordered by subrange inclusion, illustrated in figure 3.3:

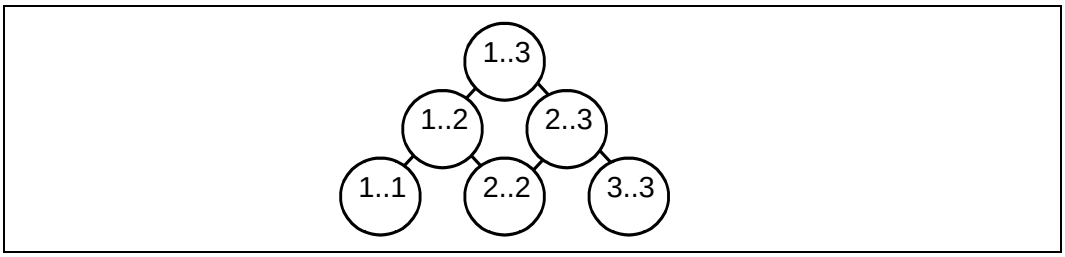

Figure 3.3: PO for Subrange Types

This has fewer arcs than the CPO for set types, due to the fact that subranges may not contain discontinuous sequences, such as the set {1, 3}. To express this constraint, the following equivalence is asserted:

$$s..t \subseteq \sigma..\tau \;\Leftrightarrow\; s \geq \sigma \wedge t \leq \tau$$

which is the axiomatic definition of a subrange. Henceforward, the (weaker) implication will be used, denoted using a sequent-calculus style of syntax:

$$\frac{s \geq \sigma,\ t \leq \tau}{s..t \subseteq \sigma..\tau}$$

[Rule 1: subtyping for subranges]

The POs permit type inference for arbitrary objects of set- or subrange-type. For example, if $x : \tau$ means x is of type τ, then we may infer increasingly more

general types for objects such as *a* or 2 by navigating upwards in the lattices, expanding the size of the type:

$a : \{a\} \subseteq \{a, b\} \subseteq \{a, b, c\}$ ...

$2 : (2..2) \subseteq (2..3) \subseteq (1..3)$ ...

3.3.3 Subtyping for Function Types

A similar navigation strategy is now employed to infer increasingly more general types for functions [CW85]. Functions are constructed using $f : \sigma \rightarrow \tau$, where σ is the domain type and τ is the codomain type. Here, subranges are used to model the types σ and τ. A function f having the type $2..4 \rightarrow 3..5$ can also be given a series of increasingly more general types constructed by expanding its codomain:

$f : (2..4 \rightarrow 3..5) \subseteq (2..4 \rightarrow 2..5) \subseteq (2..4 \rightarrow 2..6)$ ...

since any function mapping NATURALs into the codomain 3..5 will also map them into 2..5 and 2..6. However, a symmetrical expansion of the domain of a function does not result in more general function types:

$g : (2..4 \rightarrow 3..5) \not\subset (2..5 \rightarrow 3..5) \not\subset (1..5 \rightarrow 3..5)$ ...

since a function accepting NATURALs in 2..4 will not accept values outside this range, such as 5 or 1. In fact, an antisymmetrical condition applies - the domain must shrink in order to obtain a more general function type:

$h : (2..4 \rightarrow 3..5) \subseteq (2..3 \rightarrow 3..5) \subseteq (3..3 \rightarrow 3..5)$ ...

Combining both conditions, a function supertype is therefore one whose domain shrinks and whose codomain expands:

$j : (2..4 \rightarrow 3..5) \subseteq (2..3 \rightarrow 2..5) \subseteq (3..3 \rightarrow 2..6)$ ...

Turning this around, a function subtype is one whose domain expands and whose codomain shrinks. We formalise this as the function subtyping rule:

$$\frac{s \supseteq \sigma,\ t \subseteq \tau}{s \rightarrow t \subseteq \sigma \rightarrow \tau}$$

[Rule 2: subtyping for functions]

This rule says that for two function types *f* and *g*, $f \subseteq g$ if *f* is *covariant* with *g* in its result type (ie the result of $f \subseteq$ the result *g*) and *f* is *contravariant* with *g* in its argument type (ie the argument of $f \supseteq$ the argument of *g*). This is an important result, whose significance for object-oriented programming we shall observe later.

3.3.4 Subtyping for Record Types

Simple record types are constructed using $\{ x_1{:}\sigma_1, ..., x_n{:}\sigma_n \}$ where the x_i are labels and the σ_i are the types of the fields indexed by the corresponding label. Cardelli first identified record subtyping [Card84, Card88a] as a way of subsuming structures of one type in more general types.

Consider the relationship between two simple, non-recursive record types INTEGER_POINT and HOT_POINT, where HOT_POINT objects have an additional field indicating whether they are currently selected:

INTEGER_POINT = { x : INTEGER; y : INTEGER }

HOT_POINT = { x : INTEGER; y : INTEGER; selected : BOOLEAN }

To determine the subtyping relationship, Cardelli appeals to the substitutability criterion. Wherever a program expects an object of type INTEGER_POINT, a HOT_POINT object may be substituted, since it has *at least* all the fields of an INTEGER_POINT. Or, put another way, it is always possible to construct an INTEGER_POINT object from a HOT_POINT object by omitting one of its fields. This means that a HOT_POINT can be coerced to an INTEGER_POINT, but not vice-versa. This suggests a subtyping relationship:

$p : \text{HOT\_POINT} \subseteq \text{INTEGER\_POINT}$

and leads to the first part of the record subtyping rule dealing with monotonic extensions to record types:

$$\{ x_1{:}\sigma_1, ... x_k{:}\sigma_k, ... x_n{:}\sigma_n \} \subseteq \{x_1{:}\sigma_1, ... x_k{:}\sigma_k \}$$

[Rule 3.1: record extension]

which says that for two record types *q* and *r*, $q \subseteq r$ if *q* has the same number, or strictly more fields than *r* and those fields that it shares with *r* are in the same types. If two records simply share a common subset of fields, neither one is in a subtype relationship with the other, but both may be subsumed by a common super type.

The second part of the record subtyping rule comes from considering what happens if the fields of *q* are not in the same types as *r*. Consider the type:

NATURAL_POINT = { x : NATURAL; y : NATURAL }

which has the same structure as INTEGER_POINT, yet the types of its fields are different. By plotting all INTEGER_POINTs in a Cartesian plane, it is easy to see that all NATURAL_POINTs form a subset of these which occupy the first quadrant, defined by:

$$\text{NATURAL\_POINT} = \{ p \in \text{INTEGER\_POINT} \mid p.x \geq 0 \wedge p.y \geq 0 \}$$

and a subset has already been identified with a subtype. Clearly, a relationship exists between two record types if the fields of one are either all super- or subtypes of the other's fields. To see what happens in the remaining cases, consider specialising just the *x* or *y* field type, constructing the two new types:

$$\text{NAT\_INT\_POINT} = \{ p \in \text{INTEGER\_POINT} \mid p.x \geq 0 \}$$

$$\text{INT\_NAT\_POINT} = \{ p \in \text{INTEGER\_POINT} \mid p.y \geq 0 \}$$

neither of whose sets forms a subset of the other's; although they intersect in the first quadrant. This intuition leads to the second part of the record subtyping rule dealing with modifications to the field types:

$$\frac{\sigma_1 \subseteq \tau_1, \ldots \sigma_n \subseteq \tau_n}{\{ x_1{:}\sigma_1, \ldots x_n{:}\sigma_n \} \subseteq \{ x_1{:}\tau_1, \ldots x_n{:}\tau_n \}}$$ [Rule 3.2: record overriding]

This rule says that for two record types *q* and *r*, $q \subseteq r$ if they have the same number of fields and the type of each field σ_i of *q* is a subtype of the corresponding field τ_i of *r*. There is no relationship between records whose fields are in a mixture of super- and subtype relationships.

The two parts of the rule are combined in the record subtyping rule:

$$\frac{\sigma_1 \subseteq \tau_1, \ldots \sigma_k \subseteq \tau_k}{\{ x_1{:}\sigma_1, \ldots x_k{:}\sigma_k, \ldots x_n{:}\sigma_n \} \subseteq \{ x_1{:}\tau_1, \ldots x_k{:}\tau_k \}}$$ [Rule 3: record subtyping]

which says that for two record types *q* and *r*, $q \subseteq r$ if q has *n-k* more fields than *r*; and the first *k* fields of *q* are subtypes of those in *r*. The general rule reduces to rule 3.1 if the first *k* fields of *q* are in fact the same types as those in *r* (allowed by the reflexivity of $\subseteq$) and reduces to rule 3.2 if *n=k*.

Strictly, this rule has not yet covered the case for recursive records. However, a simple assumption will allow us to use Rule 3 in the context of recursive record types. A subtyping rule for general recursive types is given by [Card86]:

$$\text{REC} \quad \frac{\Gamma, s \subseteq t \vdash \sigma \subseteq \tau}{\Gamma \vdash \mu s.\sigma \subseteq \mu t.\tau}$$ s free only in σ, t free only in τ.

which says that if the current type assumptions Γ extended by the assumption $s \subseteq t$ allow us to derive "$\vdash$" that $\sigma \subseteq \tau$ then the recursive type $\mu s.\sigma$ is a subtype of the recursive type $\mu t.\tau$. We can express subtyping between recursive record types on the assumption that their syntactic recursion variables enter into a subtyping relationship. This almost provides us with enough machinery for inferring subtyping relationships between object types modelled as recursive records of functions. The only area not addressed by [Card84, CW85, Card88a, Card88b] is the effect of axioms on types.

3.3.5 Axioms and Subtyping

Axioms help define the meaning of a type, by expressing invariant properties of the type in terms of relationships pertaining between executions of some of its functions. This is a time-independent view, deliberately avoiding notions such as *preconditions* and *postconditions* [Jone86], which can have a time-dependent interpretation in state-based computations.

If a base type is further qualified by an axiom, then the effect is always to generate a type whose objects form a subset of the base type. Any set *defined by comprehension* has the property:

$$\{ x \in S \mid p(x) \} \subseteq S$$

since p(x) is either already an axiom of S or restricts S to a proper subset. Conventionally, sets defined by comprehension must indicate the base type for which the restricting predicate is well defined. Axioms are expressed in terms of operations that a given base type must possess; otherwise their meaning is undefined. Here, we examine the effect of adding or substituting axioms over a single base type. Further examples are given in [Simo94b].

A certain primitive collection of objects might have the partial specification:

$$\begin{aligned}
&\text{COLLECTION} = \exists\ \text{col}.\{ \\
&\quad \text{new} : \rightarrow \text{col}; \\
&\quad \text{add} : \text{col} \times \text{ELEMENT} \rightarrow \text{col}; \\
&\quad \text{rem} : \text{col} \times \text{ELEMENT} \rightarrow \text{col}; \\
&\quad \text{has} : \text{col} \times \text{ELEMENT} \rightarrow \text{BOOLEAN}\ \}
\end{aligned}$$

$$\begin{aligned}
&\forall \text{d, e} : \text{ELEMENT}, \forall \text{c} : \text{COLLECTION} \\
&\quad \neg\ \text{has}(\text{new}(), \text{e}); \\
&\quad \text{has}(\text{add}(\text{c}, \text{e}), \text{e}); \\
&\quad \text{add}(\text{add}(\text{c}, \text{d}), \text{e}) = \text{add}(\text{add}(\text{c}, \text{e}), \text{d}); \\
&\quad \text{rem}(\text{new}(), \text{e}) = \text{new}(); \\
&\quad \text{rem}(\text{add}(\text{new}(), \text{e}), \text{e}) = \text{new}();
\end{aligned}$$

This type of COLLECTION is empty when created, contains an element that has been added, is unordered and removes an initial element if one is present. The latter axiom deliberately underspecifies the behaviour of *rem()*. It is an open issue whether COLLECTIONs contain single, or multiple occurrences of each element; or whether *rem()* removes one, or all occurrences of an element.

Consider now the type obtained by providing the additional axioms:

$$\begin{aligned}
&\forall \text{e} : \text{ELEMENT} . \{ \text{c} \in \text{COLLECTION} \mid \\
&\quad \text{add}(\text{add}(\text{c}, \text{e}), \text{e}) = \text{add}(\text{c}, \text{e}); \\
&\quad \text{rem}(\text{add}(\text{c}, \text{e}), \text{e}) = \text{rem}(\text{c}, \text{e})\ \}
\end{aligned}$$

This definition comprehends only those COLLECTIONs which behave like SETs. It rules out those for which a double application of *add()* results in a semantically different COLLECTION; and rules out those for which an application of *rem()* leaves some occurrence of the element in the COLLECTION.

This leads to the first part of the axiom subtyping rule dealing with monotonic additions to the axioms of a type:

$$\{ x \in S \mid \alpha_1, \ldots \alpha_k, \ldots \alpha_n \} \subseteq \{ y \in S \mid \alpha_1, \ldots \alpha_k \}$$

[Rule 4.1: axiom addition]

which says that for two types *s* and *t* defined by comprehension on S, $s \subseteq t$ if *s* has the same number, or strictly more, distinct axiomatic properties than *t*. The axioms are implicitly conjoined with $\wedge$. Distinctness means that the properties are judged primary and cannot be derived from other properties.

Consider now the type obtained by adding a different axiom:

$$\forall e : \text{ELEMENT} . \{ c \in \text{COLLECTION} \mid \text{rem}(\text{add}(c, e), e) = c \}$$

which comprehends all those COLLECTIONs for which applications of *add()* and *rem()* are symmetrical, ie those which behave like BAGs. It turns out that this new axiom for BAGs actually subsumes a previous axiom of COLLECTION:

$$\text{rem}(\text{add}(c, e), e) = c \Rightarrow \text{rem}(\text{add}(\text{new}(), e), e) = \text{new}()$$

The previous axiom is in fact a ground instance of the new BAG axiom. Looking closer, it is apparent that one of the SET axioms introduced above also entails this formula by a two step proof involving another of COLLECTION's axioms:

$$\frac{\dfrac{\text{rem}(\text{add}(c, e), e) = \text{rem}(c, e)}{\text{rem}(\text{add}(\text{new}(), e), e) = \text{rem}(\text{new}(), e)} \quad \wedge \quad \text{rem}(\text{new}(), e) = \text{new}()}{\text{rem}(\text{add}(\text{new}(), e), e) = \text{new}()}$$

Typically, axioms are selected for economy, with the aim of capturing precisely the semantics of a type. It is undesirable to overspecify or to underspecify the semantics of types. It would be redundant to include, in the specification of SETs or BAGs, any axiom from COLLECTION which was automatically entailed by other SET or BAG axioms. Although subtyping always involves adding to the logical properties of a type, in some cases we may achieve this by modifying the syntactical form of axioms in order to subsume the axioms of the supertype. This motivates the second part of the axiom subtyping rule governing substitution:

$$\frac{\{ \alpha_1, \ldots \alpha_m \} \Rightarrow \{ \beta_1, \ldots \beta_n \}}{\{ x \in S \mid \alpha_1, \ldots \alpha_m \} \subseteq \{ y \in S \mid \beta_1, \ldots \beta_n \}}$$ [Rule 4.2: axiom substitution]

which says that for two types *s* and *t* defined by comprehension on S, $s \subseteq t$ if the *m* syntactically modified axioms α_i of *s* necessarily entail the *n* original axioms β_i of *t*. Axiom substitution is not one-to-one, but on the basis that the new set of axioms entails the original set. The size *m* of the substituted set may therefore arbitrarily grow or shrink with respect to the size *n* of the original set, so long as the entailment "$\Rightarrow$" obtains.

The two parts of the rule are combined in the axiom subtyping rule:

$$\frac{\{ \alpha_1, \ldots \alpha_k \} \Rightarrow \{ \beta_1, \ldots \beta_n \}}{\{ x \in S \mid \alpha_1, \ldots \alpha_k, \ldots \alpha_m \} \subseteq \{ y \in S \mid \beta_1, \ldots \beta_n \}}$$ [Rule 4: axiom subtyping]

which says that for two types *s* and *t* defined by comprehension on S, $s \subseteq t$ if *s* has *m-k* more distinct axiomatic properties than *t* and the first *k* axioms of *s* necessarily entail all *n* axioms of *t*. The general rule reduces to rule 4.1 if the first *k* axioms of *s* are in fact identical to the *n* axioms of *t* (allowed by the reflexivity of $\Rightarrow$) and reduces to rule 4.2 if *m=k*.

The axiom subtyping rule permits the derivation of syntactically similar, but semantically disjoint subtypes of a common, partially specified type. Familiar examples of these include STACKs and QUEUEs [Amer90, Simo94b]. However, it is more usual for axioms to be introduced at the same time as new operations. In this case, the task at hand is to relate two sets defined by comprehension over two *syntactically different* base types:

$$\{ x \in S \mid \alpha_1, \ldots \alpha_m \} \subseteq \{ y \in T \mid \beta_1, \ldots \beta_n \}, \quad S \subseteq T$$

but provided these are also in the same relationship, subtyping is preserved. Consider that, in general, we may interleave the syntactic and semantic subtyping stages:

$$\{ x \in S \mid \alpha_1, \ldots \alpha_m \} \subseteq S \subseteq \{ y \in T \mid \beta_1, \ldots \beta_n \} \subseteq T$$

T is a syntactic type whose semantics is then given by $\tau = \{ y \in T \mid \beta_1, \ldots \beta_n \}$. The syntactic subtype $S \subseteq \tau$ is defined by adding one more operation *f* to τ's interface. So far, *f* has no semantics. If a set $\sigma = \{ x \in S \mid \alpha_1, \ldots \alpha_m \}$ is now defined to give *f* a meaning in relation to other operations, this in turn creates a subtype of the partially specified type S.

The correspondence between first-order logic and the typed λ-calculus has been known for a long time [CF58, Tait65, Howa80]. For example, the type elimination rule for $\rightarrow$ mirrors the *modus ponens* rule in logic:

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

As a result, it is not surprising that relationships can be found between axiomatic and syntactic ways of describing type. Most of the existing literature, however, disregards axioms in its formulation of type rules. What has been achieved here is to introduce such rules for the sake of those languages [Meye88, Meye92, Omoh94] which reason incrementally with axioms in order to determine type compatibility relationships.

3.4 Object Subtyping

Chapter 2 described how object-oriented languages open up their type systems to allow systematic sets of relationships between types. A key focus of interest is to discover under what conditions an object having one type may safely be bound to a variable expecting a different type - the so-called *substitutability criterion* [CW85], which offers a rudimentary kind of polymorphism. More is at stake than this, however. The theory that classes can be treated as types and subclassing can be modelled as subtyping [SCBK86, Stro86, Meye88] must be properly tested. By integrating all the above subtyping rules, it is possible to formulate the conditions under which one object type is compatible with another.

3.4.1 Harmonising Existential Types and Closures

As a precursor to exposing the rules for object subtyping, we need to fine-tune some of the syntax, since we have used a mixture of data abstraction (existential types) and procedural abstraction (functional closure) techniques to motivate models of object behaviour and subtyping. We presented the type effects of axioms using existential types for the sake of having a freestanding initial constructor *new()* for each type. Clearly, such a constructor is vital for specifying boundary conditions on types. A potential problem exists with the *new()* constructor in functional closure models.

Firstly, it is arguable whether such a method occurs in the interface of objects, or whether it is external, used to create the closures which are the objects. Secondly, incorporating the *new()* method is technically difficult. This is because the constructor for closures would have to contain an embedded call to itself in the *new()* method, which in turn would have to be fixed over a different *self* object. While there are abstraction techniques to solve this [Harr91a, CHC90] which will also handle different numbers of instantiation parameters, we prefer for the moment to consider the object constructor as an external function.

This done, algebraic specifications may be converted into object-oriented specifications by changing the style of invocation to reflect the selection of methods. For example, the unique element axioms for SETs would appear as:

```
∀d, e : ELEMENT, ∀s : SET
    s.add(e).add(e) = s.add(e);          -- unique elements
    s.add(e).rem(e) = s.rem(e)
```

The axioms requiring an empty set would refer to the *newSet* constructor, an external function used to create closures representing sets.

Problems to do with recursion and binding occur again and again in object-oriented programming. This is the theme of later chapters.

3.4.2 Subtyping Rules for Objects

The subtyping Rules 1-4 given above are now combined and presented in a more practical and accessible format. Any two related object types σ and τ, modelled as records containing functions, are in a subtype relation $\sigma \subseteq \tau$ if:

- extension: σ adds monotonically to the functions inherited from τ (Rule 3); and
- overriding: σ replaces some of τ's functions with subtype functions (Rule 3); and
- restriction: σ has a stronger data type invariant than τ (Rule 4) or is a subrange (Rule 1) or subset (Rule 0) of τ.

A function σ.f is a legal subtype replacement for another τ.g only if:

- contravariance: the arguments of σ.f are more general supertypes than those of τ.g (Rule 2); and therefore preconditions are weaker (Rule 4);
- covariance: the result of σ.f is a more specific subtype than that of τ.g (Rule 2); and therefore postconditions are stronger (Rule 4).

The ability to *extend* type structure and obtain a subtype is clearly a boon for object-oriented programming. Intuitively, a variable may always safely receive a longer record than it expects, in which case it can only access some of the fields. This kind of assignment is typically handled by copying a subset of the record's fields into the target variable [Stro91] or by copying a pointer to the subtype record [GR93, Meye88]. In either case, functions statically bound over the variable can only access supertype fields, which are guaranteed to be present in the subtype record. The requirement always to extend structure and behaviour monotonically means that selective inheritance is illegal from a type-theoretic viewpoint.

The ability to *restrict* type membership to obtain a subtype is well-known. In conventional languages, it is common to declare integer subranges, whose elements are then correctly handled by the functions of the base integer type [Wirt82]. In this case, the result of such expressions has the base type and type information about the subrange is lost. Type restriction is handled in another way in Eiffel [Meye88, Meye92], which introduces class axioms called

data type invariants [Hoar72], restricting the semantics of a syntactic type. Subtypes may strengthen this invariant, by the addition of axioms. A function expecting its argument to have certain semantics will receive objects having at least the required semantics, if not stronger.

The ability to *override* parts of a type's structure is peculiar to object-oriented languages. Chiefly, this is accomplished by replacing methods with new versions having modified type signatures. The requirement to replace functions with subtype functions often strikes programmers as strange, especially the *contravariant* rule for function arguments. This is a counter-intuitive result [Cook89b] for object-oriented programming, which wants to be able to replace a function $f : \tau \rightarrow \tau$ defined over one type τ with another function $g : \sigma \rightarrow \sigma$ closed over the subtype $\sigma \subseteq \tau$. In a statically-bound language, this would present no problems, since a variable typed in τ would always safely execute *f()*, even if it contained an object of type σ. However, because of dynamic selection in object-oriented languages, a variable typed in τ may sometimes execute *g()*, when it contains an object of type σ. Such a function should not restrict its arguments any more than the function it replaces. Dynamic dispatch has been compared with higher-order functional programming [Harr91b] in this respect. The *covariant* rule presents less of a problem; it allows function results and record field-types generally to be retyped with subtypes.

Whereas type axioms were introduced in the wholistic, algebraic way above, here they are unpacked into the more familiar *pre-* and *postconditions* and *invariants* encountered in specification languages [Jone86, Spiv88, Spiv89] and Eiffel [Meye88, Meye92]. It should be obvious that attaching a precondition to a function is like restricting the set of arguments it may accept; and that attaching a postcondition is like restricting the valid results it may yield. In practice, programming languages have the choice of maintaining a smaller set of types, at the cost of admitting *partial functions*, which are not applicable to every element of the type - consider *pop()* and the empty stack - or introducing a much larger set of types having total functions, at the cost of admitting some dynamic typing. OBJ specifies stacks using the sorts *Stack* and *NeStack* (*ie* non-empty stack) for the sake of defining a total *pop()* function [FGJM85]. Based on this insight, certain experimental languages have adopted the idea of *dynamic reclassification*, whereby state changes affecting the boundary conditions of an object result in it changing type.

3.4.3 A Language Survey

Immediately it is clear that a large group of languages violate the requirements for subtyping. It is impossible, in these languages, for *classes* to be considered *types* and *subclassing* as *subtyping*. We have reviewed some of the more popular languages in this respect [Simo94a]:

Smalltalk [GR83, Digi92]: Despite its informal commitment to class protocols, *Smalltalk* allows inheriting classes to provide replacement methods with arbitrarily changed argument lists - the number and types of argument need not have anything in common with the superclass's method. *Smalltalk*'s protocols are only checked for name-equivalence.

Furthermore, a form of selective inheritance is practised whereby general methods are later derailed in subclasses for which they do not legitimately apply. An example [Digi92] is *Collection*'s *add:* method which is replaced by an error message in *FixedSizeCollection.* This is a consequence of having only a single dimension in which to factor out behaviours. With care, the need for method derailment can be avoided in languages with multiple inheritance.

Smalltalk only supports *implementation inheritance* in which a *Dictionary* class, a keyed lookup table that is implemented as a hash table of <key, value> pairs, can inherit the hash table implementation of a *Set*; and yet applications using *Sets* would behave quite differently if given *Dictionaries* instead [GJ90] .

C++ [Stro91, ES90]: Notwithstanding its unprepossessing parent language, *C++* is fairly consistent in its support for subtyping. The types of data declarations may not be changed. Derived classes may supply replacement functions only if their type signatures match those of their forerunners exactly. This rule does not include the type of the implicit argument (the owning class, whose type status is actually determined by application of the rule) and is in fact stricter than required. However, the ability to provide overloaded functions (within the same class) can sometimes have the effect of hiding an inherited function by accident.

C++ provides alternative *private* and *public* inheritance mechanisms. The former permits the sharing of implementation only - privately derived classes are barred from being treated as subtypes. The latter provides both type and implementation inheritance. This is upheld by the exporting mechanism, which is linked to public inheritance - any function declared public in a base class is also public in any publicly derived class. *Friend declarations*, intended to give certain client classes privileged access to *private* information, have the effect of opening up the external interface on an *ad hoc* basis. However, these may not be inherited.

In some other respects, *C++* is less secure, in that type coercions can fairly easily be constructed in arbitrary directions and these can be invoked, without the programmer being aware, in expressions of mixed type.

Eiffel [Meye88, Meye92]: A dedicated exponent of strong, axiomatised types, *Eiffel* nonetheless retains features which mitigate against subtyping. For function replacement, *Eiffel* follows the *covariant* rule for function results, but curiously ignores the *contravariant* requirement for function arguments [Cook89b]. Meyer has vigorously defended this decision, on the basis of arguments explored later. While he has provided a global patch to fix type aliasing problems arising from the violation of contravariance [Meye89, Meye95], this is not the correct mathematical solution. *Eiffel* is unique in allowing object attributes to be retyped with a subtype [Meye88], which is allowed by the *covariant* rule, since it considers access to attributes as the invocation of a function, delivering a result. In [Cook89b] Cook points out that attributes must also be initialised and gives an example where this fails. We have subsequently analysed this case as another violation of *contravariance*: assignment is a function that is redefined for every new subtype τ and has the signature := : $\tau \rightarrow$ UNIT. Elsewhere, *Eiffel*'s generic parameter mechanism is

theoretically neutral with respect to the type rules given here, but may interact unhelpfully with *contravariance* [Coo89b, Simo95]. Parametric polymorphism will be considered in much more detail later.

Version 2 [Meye88] provided orthogonal inheritance and export mechanisms, leading to a form of selective inheritance in which a feature exported in a parent class could be subsequently hidden in a child. Version 3 [Meye92] added a finer-grained visibility mechanism: each feature may list the client classes to which it is exported. This means that a class has different public interfaces, depending on the client's perspective! Fortunately, visibility declarations are observed in subclasses by default, although the option exists to change them. Version 3 provides a *feature undefinition* mechanism, which removes its effective implementation, but not its signature, from the class interface. This is still equivalent to selective inheritance, since it is an error to invoke deferred features.

Eiffel is best known for its *assertions*, executable axiomatic statements defining the semantics of its routines. Coincidentally, *Eiffel* obeys the rules for axioms, although this is justified in terms of Meyer's *programming by contract* metaphor, rather than from subtyping considerations [Meye88, p256-7]. All services that are guaranteed by one class must also be guaranteed by its descendants, therefore the postcondition on which the success of the service depends must not be weaker, but may be stronger. On the other hand, all messages which one class understands must also be understood by its descendants, therefore the precondition on which acceptance of a message is contingent must not be stronger, but may be weaker. Inconsistencies between *Eiffel*'s syntactic and semantic type rules have been discussed on the Internet newsgroup *comp.lang.eiffel*.

Trellis [SCBK86] and *Sather* [Omoh94, SOM93] are two among the very few languages which correctly observe both the covariant rule for function results and the contravariant rule for arguments. For *Trellis*, *class* is *type* and *classification* is *subtyping*; *Sather* correctly recognises cases when inheritance supports subtyping and disallows polymorphic aliasing when subtyping is contravened. The independent type hierarchies of *POOL-I* [Amer90] and of *Emerald* [BHJL86] also respect the subtyping rules as we have given them. However, as the following chapter demonstrates, a strict adherence to subtyping eventually paralyses object-oriented languages. The problem is that classes are not types at all, but something else altogether.