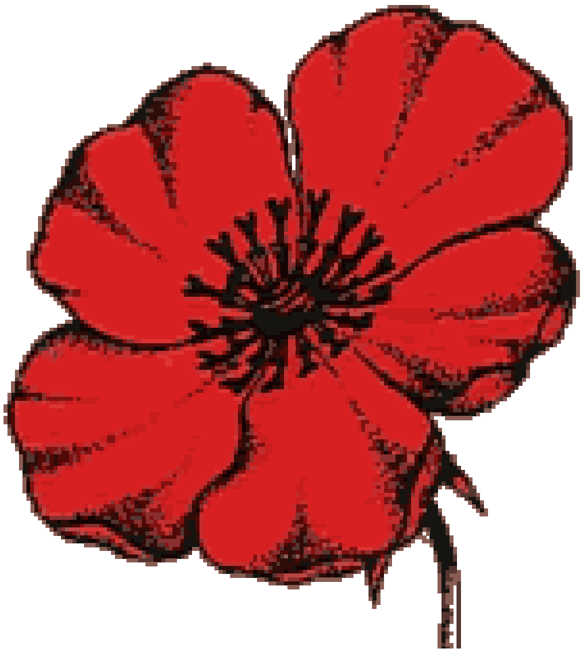
The Poppy Language Manifesto



Anthony J H Simons

Department of Computer Science
University of Sheffield
Regent Court, 211 Portobello
Sheffield S1 4DP
United Kingdom

Abstract

Poppy is a compact and expressive object-oriented language. The type system for Poppy is based on the second-order theory of F-bounds, rather than a first order theory of subtyping. Classes are treated as parametric polymorphic families of types, possessing at least a given interface. While the formal treatment of polymorphism would normally require the use of type parameters, Poppy hides this in its surface syntax, using systematic type substitutions on class identifiers to describe the propagation of new types into variables, when these are bound to more specific objects. This is most apparent during inheritance, where the self-type evolves automatically. The type system checks that each method invocation is correctly typed in the calling context, by propagating specific types into general variables. As a result, Poppy naturally supports second-order classes, whose methods are recursively closed over each new class, avoiding the problems associated with covariant argument restriction in type systems based on first-order subtyping.

Disclaimer

Readers should note that this document is a work in progress. It is being extended and modified as the design of the Poppy language is worked out. Certain sections of this manifesto may be incomplete.

Table of Contents

1. Language Syntax

This section introduces the syntax of the Poppy language. Programs are constructed out of *classes*, which are defined in separate textual units. A class defines a family of data types, which possess a common set of attributes and methods (variables and functions). Classes are organised according to their similarities and differences in a classification hierarchy.

1.1 Class Definition

The concrete syntax of the Poppy language is designed to be similar to that of other popular object-oriented languages, such as Java, C# and C++, while the encapsulation policy is influenced more by Eiffel. Poppy adopts certain standard conventions on the visibility and allocation of attributes and methods, reducing the number of explicit keywords needed to indicate private or public, abstract or concrete, and shared (static) or replicated elements. To this extent, Poppy strives for minimalism in its syntax.

However, certain kinds of declaration are made more explicitly than in other languages. The standard form of a class definition begins:

```
class (ClassName self) {
     …
}
```

in which both the `ClassName` and the `self`-referential variable standing for the current object are declared explicitly. This mimics the style of variable declaration elsewhere in the language and is a point of consistency. The `self`-reference is unbound and may be redirected onto instances of this class, or of descendant classes. Likewise, the `ClassName` is treated as the polymorphic self-type of the current class, within the class definition, and may be subsequently adapted to descendant types. In this sense, it behaves more like a type parameter than a simple type.[1] The body of a class defines a collection of attributes and methods, collectively known as features.

Poppy supports multiple and overlapping classification. The basic notion is that a class is compatible with any superclass, whose method interface it extends, after unification of the self-types and corresponding feature-types (the types of their attributes and methods). Type matching is structural, according to the shapes of the interfaces. A longer interface is compatible with a shorter one, so long as a similar relationship is satisfied recursively by the feature-types. A subclass is defined by specialising one or more superclasses and adding extra features. Poppy supports the multiple inheritance of type and implementation (with a conflict-resolution strategy) and does not distinguish the separate syntactic notion of an *interface* (as found in Java) from the regular notion of a *class*. An abstract class in Poppy is the same thing as an interface (having method signatures, but no implementations).[2]

[1] A class definition is formally treated as a double functional over two recursion variables, standing for the self-type and self-value. When an instance of the class is created, the resulting object and its type are the fixpoints of the respective functionals.

[2] This reflects the identical formal treatment of *interfaces* and *abstract classes* in the theory of F-bounds. The syntactic distinction between these notions in Java is motivated more by the single-inheritance model.

Where a class inherits from another, this is shown using the `extends` keyword (as in Java):

```
class (Child self) extends (Parent super) {
     …
}
```

in which both the `Parent` class name and `super`-reference to a parent object are declared explicitly. Technically, `super` refers to the inherited part of the current object `self`. This allows an overriding method in the `Child` to invoke the same-named method on `super`, to access the last implementation known to `Parent`, and supports a method combination strategy, in which methods may extend inherited methods. Apart from this, when `Child` inherits `Parent`'s methods, the types of these are specialised to refer to the `Child`, wherever they previously referred to the `Parent`.

The explicit naming of all the self-referential variables is most useful in cases of multiple inheritance, in which the child class is able to determine how multiple implementations of the same-named method are to be combined. For example, the following redefines the `equal` method of the child:

```
class (Child self) extends (Mother mother, Father father) {
     …
     Boolean equal (Child other) {
          mother.equal(other).and(father.equal(other))
     }
}
```

simply by combining the results of the `equal` methods inherited from the `mother` and `father` parts of itself using logical `and`. The same approach may be used to prefer the left- or right-hand implementation, or combine implementations in some other way. Other conflict resolution rules support the automatic merging of the same implementation that was inherited from the same ancestor class via multiple parents (the "fork-join" scenario). The typing issues in method combination are complex; and a full discussion of these is deferred.

1.2 Attribute Definition

The main body of a class definition consists of *attribute* and *method* definitions, sometimes collectively known as the *features* of a class (as in Eiffel). The attributes declare the variable storage allocated to instances of the class. If a class inherits from other classes, the locally declared attributes are added to those declared for the class's ancestors (and attributes inherited multiple times from the same ancestor, via different parents, are merged). Attributes cannot be redefined, but may be retyped (implicitly, by binding to more specific values). Sample attribute declaration styles are indicated by:

```
String forename;
String forename, surname;
Integer months := 12;
```

where the attribute's declared class is given first, then its name. For convenience, multiple attributes of the same class may be declared in a single expression, in which case the attributes are comma-separated. Optionally, an initialisation expression may be given as part of the declaration. Each declaration is separated from the following declaration by a semicolon. (Technically, the semicolon character is a statement *separator* in Poppy, rather than a *terminator*. It is provided to allow compilers to attempt error-recovery after a syntax fault is detected in a single statement. Compilers are also expected to be lenient in the way they process empty statements inserted accidentally at the ends of blocks).

By default, attributes declare variable storage that will be allocated in each individual instance. However, if an attribute is initialised as part of its definition, then it is a shared attribute, automatically allocated once for the whole class. This removes the need for any keyword such as *static* to denote the kind of allocation required. Further examples of attributes declared in the context of their owning class are indicated by:

```
class (Circle self) {
      Decimal pi := 3.1415926;
      Point centre;
      Integer radius;
      …
}
```

where `pi` is a shared class attribute (since it is intialised in the definition) and `centre` and `radius` are instance attributes, allocated in every instance of `Circle`. The values for these instance attributes will be supplied at object construction time, using constructors.

Attributes are typed using class names, referring to classes defined elsewhere. At the time of declaration, these are polymorphic class-types, until bound to values of a specific type. For example, though the `centre` is delcared to be of the `Point` class, it could be initialised to an instance of some descendant of `Point`.

By default, attributes have read-only `public` access and can be invoked exactly like simple access methods to return their value. This avoids having to write many trivial access methods and dispenses with the usual name-clashes when seeking to distinguish the names of attributes and their associated access methods. Attributes may only be updated by the owning object, which can be any object within the declaring class (including more specific objects of descendant class types). The following illustrates legal and illegal attempts to access an attribute:

```
Circle circle := Circle(Point(3, 4), 5);
Integer rad := circle.radius;    // OK, read-access allowed
circle.radius := 10;             // Error, remote assignment
```

in which it is legal to access the value of a `circle`'s radius externally, but illegal to attempt to modify this attribute through remote assignment. This is similar to the read-only policy in Eiffel, which achieves a good balance between encapsulation and succinctness.

1.3 Method Definition

The main body of a class definition also contains *method* definitions. Methods are the functions and procedures owned by a class, and implement the behaviour shared by all instances of the class. Each class may define its "own methods" for achieving similar goals, or delivering particular services. Methods are automatically allocated once for the whole class (like initialised attributes). If a class inherits from other classes, the locally declared methods are added to those declared for the class's ancestors (and methods inherited multiple times are merged). Methods can be redefined, in which case the new version overrides the inherited version; but the new version may also choose to incorporate the old version (known as *method combination*). Methods may be retyped (both implicitly, by rebinding, or explicitly, by redefinition). Sample method definition styles are indicated by:

```
Decimal area {
     pi.times(radius.times(radius).toDecimal)
}

Circle moveTo (Point position) {
     centre := position;
     self
}
```

The `area` method is in the style of a function, which computes the area of the owning `Circle` object from its (locally stored) `radius` and (shared) value of `pi`. The `moveTo` method is in the style of a procedure, which updates the `centre` of the owning `Circle` object. Methods declare the result type first, then the method name, followed by any further parameters in parentheses. The parentheses are only required if the method actually has further parameters – and are otherwise forbidden. This simplifies the defining and calling syntax for simple methods, which need not have empty parentheses. The `area` method above is invoked in exactly the same style as the `radius` attribute, to access a simple property of the `Circle`. This syntactic fusion is deliberate (similar to Eiffel).

Where methods have multiple parameters, there are two styles for supplying these. The following examples illustrate:

```
PhoneBook add (String name, Natural number) {
     contacts.addAt(name, number);
     self
}

Point moveTo (Integer newX, newY) {
     x := newX;
     y := newY;
     self
}
```

The `add` method of the `PhoneBook` class accepts two comma-separated parameters of different classes. In this case, each parameter name must be prefixed by its own class. Where a method accepts multiple parameters of the same class, they need only be prefixed once by the class name, as in the `moveTo` method of the `Point`. Mixtures of these styles may be used if methods accept some parameters of the same type, and others of different types.

The body of a method is an *expression*, enclosed in braces, which has a value, the result of the method. The body expression may either be a single statement, or a compound statement, consisting of single statements separated by semicolons. The semicolon is really provided to allow compilers to attempt error-recovery after malformed statements are encountered. Compilers are expected to be lenient in the way they process empty statements accidentally inserted at the ends of blocks.

By convention, all Poppy methods return a result, which is always the value of the last statement in the body (or, in the case of branching constructions, the value of the last statement in each branch). Poppy deliberately has no *return* keyword, to encourage clean programming styles and prevent early exit by jumping out of a method. Procedural methods usually return `self` by convention, referring to the owning object.

An alternative convention for simple update methods is to return the new updated value, which follows naturally from the result returned by an assignment expression, which is the assigned value:

```
Natural deposit (Natural amount) {
     balance := balance.plus(amount)
}
```

The choice of style is up to the programmer; the former style supports writing a sequence of updates to the same object (in the style of Smalltalk), whereas the latter supports nested actions on the assigned value.

Sometimes, the body of a method may be empty, in which case the method is *abstract*, that is, having a signature, but no implementation. The abstract `Shape` superclass of all geometric figures `Circle`, `Square` and `Rectangle` could define an abstract `area` method, to indicate that all its concrete descendants will eventually supply suitable implementations:

```
Decimal area {}
```

The difference between this syntax for abstract methods and the attribute definition syntax is in the use of empty braces. The difference between this syntax and the syntax for a default method with a trivial implementation (a *nullop*) is that a concrete, or executable method must always return some trivial value:

```
Decimal area { 0.0 }
```

Abstract methods must eventually be replaced by concrete methods in descendant classes. Alternatively, some descendants may choose to implement the abstract method by an attribute, providing access to storage, rather than computing the result. This policy aims to give designers complete freedom regarding how services will eventually be provided.

A compiler may warn if an abstract method has not been supplied with a concrete definition, in a context that expects all methods to be properly defined. A class may be partly abstract, if it has at least one abstract method, or wholly abstract, if all of its methods are abstract. A wholly abstract class is the equivalent of an *interface* in other languages.

1.4 Operator Syntax

In Poppy, all operations are fundamentally considered to be the methods of classes. Even basic operations, such as arithmetic, are styled as the methods of the numeric classes, for the sake of uniformity. So, for example, the usual `Integer` operations are officially defined in the following style (omitting the details of the method bodies):

```
class (Integer self) extends (SignedNumber number, …) {
     Integer plus (Integer other) { … }
     Integer minus (Integer other) { … }
     Integer times (Integer other) { … }
     Integer divide (Integer other) { … }
     …
}
```

For convenience's sake, the usual mathematical operators are also provided as shorthand for these, and similar methods. The operator syntax is always understood to be convertible into the canonical method invocation syntax:

```
a + b        ⇔   a.plus(b)
a – b        ⇔   a.minus(b)
a + b * c    ⇔   a.plus(b.times(c))
a * b + c    ⇔   a.times(b).plus(c)
a := b := c  ⇔   a.assign(b.assign(c))
```

The later examples also illustrate how certain rules of precedence and associativity are observed in the order of evaluation of operators. It is expected that '`/`' and '`*`' should be evaluated before '`+`' and '`-`' according to the usual mathematical order of precedence; likewise it is expected that nested assignments are evaluated from right-to-left, rather than left-to-right. This is achieved by declaring the precedence of an operator explicitly, along with its associativity and fixity. Any character symbol or symbol sequence can be declared as an operator, using the following declaration syntax:

```
operator[60] (a + b) a.plus(b);
operator[70] (a * b) a.times(b);
operator[-90] (! a) a.not;
operator[-10] (a := b) a.assign(b);
operator[50] (a == b) a.identical(b);
```

The `operator` keyword introduces an operator declaration, which includes a positive or negative integer value in square brackets. The absolute value of this number denotes the operator precedence, such that 1 is the lowest, and higher values give operators a higher precedence (they will be evaluated sooner). The sign of this number denotes the associativity, where positive stands for forward (left-to-right) and negative stands for backward (right-to-left) evaluation. Immediately following this is a *pattern expression* in parentheses, denoting the infix, prefix or suffix pattern of the operator. Pattern expressions may contain simple variables, standing for the operands. The following *translation expression* denotes the equivalent translation of the operator pattern expression into the regular method invocation syntax. The same variables are repeated in the translation expression.

Any class can declare operators, so long as the translation expression refers to known methods. Operators can also be declared for construction expressions, such as the range constructor:

```
operator[50] (a .. b) Range(a, b);
```

which creates a `Range` object, whose values range from the lower bound `a` to the upper bound `b`, inclusively. Ranges are used frequently in deterministic loops.

1.5 Control Structures

The flow of control within methods may be directed using familiar control structures, offering single- and multiple branching, multiple case selection and looping using conditional or deterministic loops. Many of these are syntactically similar to control structures in other languages, like Java and C++, except that the programmer should remember that control statements, like other expressions, have return values. Poppy treats deterministic loops as a special case of deterministic iteration over collections.

The single- and binary-branching `if` statement accepts a `Boolean`-valued condition in parentheses and each branch returns a value, which should be of a common, or related type (formally, the class of the `if`-statement is the least upper bound class of the types returned by each branch):

```
Boolean withdraw (Natural amount) {
     if (amount.moreThan(balance))
          false
     else {
          balance := balance.minus(amount);
          true
     }
}
```

Each branch may contain a simple, or compound statement, and the `else`-branch may be omitted if no alternative action is required. If the `if`-statement has a single branch, the absent branch returns `null`. Multiple branching on conditions is established by using nested `if`-statements. However, where a multibranch selection is made on the basis of a scalar value, the more efficient `case` statement may be used. This behaves more like Pascal's *case* statement than like C's *switch* statement, which has unfortunate fall-through behaviour.

```
String weekDay (Natural days) {
     case (startDay.plus(days).modulo(7)) {
       0 : “Sunday”;
       1 : “Monday”;
       2 : “Tuesday”;
       3 : “Wednesday”;
       4 : “Thursday”;
       5 : “Friday”
     }
     else
          “Saturday”
}
```

The scalar expression is evaluated and if the result is equal to one of the case labels, then the following simple or compound statement is executed (without fall-through to the remaining statements). If none of the case labels matches, an optional `else` expression is evaluated. The result of the case statement is the result of one of its branches, each of which should return a common, or related type. If no case labels match and no `else`-expression is provided, the result is `null`.

Looping is provided through conditional and deterministic looping constructs. The conditional `while` loop evaluates a condition, then repeats a simple or compound statement for as long as the condition remains true. The deterministic `for` loop iterates a single variable over a predetermined range of values, repeating a simple or compound statement once for each value in the range. When treated as expressions, all looping constructs are considered to have a `null` value. The following examples illustrate:

```
Integer[] array := Integer[10];

Natural index := 0;
while (index < array.size) {
     array[index] := index.toInteger * 2;
     index := index + 1;
}

for (Natural index : array.firstIndex .. array.lastIndex)
     array[index] := array[index] / 2;

for (Integer element : array) {
     output.print("Array element: ");
     output.printLine(element)
}
```

Primitive arrays are indexed from zero to one less than the array length and have standard methods `size`, `firstIndex`, `lastIndex` and the subscript `[]` operator. The first example is a `while`-loop that initialises an `array` of 10 `Integer` values, until the increasing `index` reaches the `size` of the `array`. Normally, `while`-loops are reserved for nondeterministic conditions, unlike this example, since the bounds of the iteration are known. The second example is a deterministic `for`-loop, which introduces a local `index` variable (hiding the `index` variable in the next outer scope). The `index` automatically ranges over every value in the declared `Range`, which can be any scalar subrange, and is inclusive of the first and last value in the range. The `for`-loop can accept a variable of *any* element-type, which ranges over the collection supplied as the other argument. The third example shows an even simpler style for accessing the elements stored in the `array`, by iterating over the elements directly, rather than using an `index`. This style could not be used to update the contents of the `array`, but only to visit each element.

By default, a `Range` is assumed to be monotonically increasing (other iteration styles are discussed below). If the first and last elements are the same, a deterministic loop will be executed once. If the first element comes after the last, the loop will be skipped. A `Range` is really like a collection of values, but computes its elements on demand.

1.6 Construction and Destruction

When variables are first declared, their values are formally undefined, although the language should ensure that a standard *blank* initialisation is performed (setting bytes to zero). The following declarations:

```
Person john;
Integer age;
```

should result in `john` having the blank value corresponding to `null`, and `age` having the blank value corresponding to `0`. Variables may be initialised with values as they are declared, using a construction expression:

```
Person john := Person("John", 35, 'm');
Integer age := Integer(35);
```

Poppy deliberately blurs the distinction between initialising value-types and allocating reference-types on the heap by not having an explicit `new` operator. So, while the `Person` constructor above will typically allocate a new instance on the heap, the `Integer` constructor might store the value directly in the variable. (In fact, since `35` is the literal representation of an `Integer`, the constructor is strictly unnecessary). So long as everything appears to behave like an object, the implementation can choose to make such optimisations.

Every class in Poppy has a *single* constructor, whose name is identical to the class's name. Invoking the name of a class without arguments creates a blank object, a clean instance of the class, whose attributes are *not* initialised. So, in the following expression:

```
Person john := Person;
```

the variable `john` is initialised with a blank instance of `Person` (viz. the value of `john` is not `null`, but refers to a `Person` object, whose attributes are blank). This is the standard form of construction, which is understood to allocate the object, but not initialise it. This strategy was chosen because it simplifies object construction, for example, when restoring objects from files, where objects must be created before they can be initialised.

When a constructor is followed by parentheses surrounding further initialisation values, this is treated as shorthand for invoking a `copy` method to initialise the blank instance. So, the following expressions are equivalent, constructing a blank `Person` then initialising it:

```
Person john := Person(“John”, 35, ‘m’);
Person john := Person.copy(“John”, 35, ‘m’);
```

By default, every class has two overloaded `copy` methods, one that copies another object like itself, and the other that initialises all the declared attributes of the object from a tuple of parameters. The 3-tuple above assumes that the `Person` class was defined with three attributes in the given order:

```
class (Person self) {
      String name;
      Integer age;
      Character sex;
      …
}
```

A standard `copy`-method will be supplied that is capable of initialising all of these attributes, in the order they were declared. If it is desired to control in finer detail how objects are initialised, the programmer may write explicit overloaded `copy` methods accepting different numbers or types of argument. For example, the `Circle` class may provide a `copy` method accepting only the `radius`, which then allows construction of `Circle` objects at default `origin` locations:

```
Circle copy (Integer radius) {
      copy(Point(0, 0), radius)
}

Circle circle := Circle(5);
```

The special `copy` method calls one of the default `copy` methods, which returns `self`.

Unused objects are reclaimed by automatic garbage collection. Any suitable policy, such as mark-and-sweep, reference counting or generation scavenging may be used. In general, a memory management system can be relied upon to reclaim all unused objects eventually. However, there are circumstances in which objects obtain other resources, which they must release explicitly. For this reason, a class may define a `tidy` method, so its instances may perform clean-up actions automatically as they are deleted. By default, the `Object` class defines a trivial method:

```
Object tidy { self }
```

which returns the reference to be reclaimed. Other classes may redefine this, so that their clean-up actions are always performed, before they are deleted. For example:

```
Reader tidy {
      input.close;
      super.tidy      // inherited clean-up
}
```

ensures that the `input` file stream owned by a `Reader` object is always closed, if the `Reader` object is deleted in normal, or abnormal circumstances. Unlike Java's *finalize* method, which is not called reliably for all objects, `tidy` is automatically called for every reclaimed object.

1.7 Encapsulation and Visibility

Poppy supports the encapsulation of attributes and methods in a different way from most other object-oriented languages, enforcing encapsulation at an object-level rather than a class-level. The chief distinction is between features that are `public`, if they are intended for use by external clients of an object, and those that are `private`, if they are intended for use internally by the object itself. Public visibility (the default) grants *read-only* access to the feature (method, or attribute) in question; attributes are therefore *always* protected from external modification by remote assignment. Private visibility indicates that the feature can only be accessed by the owning *object*, rather than by any instance of the same *class*.

Poppy maintains a strong theoretical position that inheritance is merely shorthand for defining a class by extension from another. Subclasses defined incrementally using inheritance should be exactly equivalent to classes that were defined from scratch. This rules out treating inheritance as a modularisation mechanism, by which features declared in the parent are kept secret from the child. Poppy therefore makes no distinction between *private* and *protected* visibility; and Poppy's `private` is more like the *protected* visibility of Java, or the *secret* export status of Eiffel. Poppy's `private` features can be accessed in instances of the declaring class, and in instances of all descendant classes. This is more liberal than the *private* visibility of Java and C++. On the other hand, encapsulating at an object-level is also more restrictive. Whereas declaring a feature *private* in C++ allows one instance (of the declaring class) to access this feature in another instance of the same class, this is prohibited in Poppy.

Attribute and method declarations may be prefixed by the `private` or `public` keywords, to indicate the visibility granted to that feature. However, since the *default* is `public` visibility granting *read-only* access, most attributes and methods may be declared without further qualification. Methods are only ever accessed (for invocation) and are never reassigned, so read-only access is operationally equivalent to public method access in other object-oriented languages. For attributes, the default visibility grants efficient *read-only* access to stored data and also prevents external modification by remote assignment. This removes the tedious burden of writing of many simple public access methods to return the values of private attributes in other object-oriented languages. To update the value of an attribute, a class must provide a method that explicitly allows this. Internally, the method may refer to the attribute by its short name and assign the new value.

The other advantage of permitting read-only access to attributes is that there is no distinction, in the Poppy syntax, between invoking a simple method that computes a result, and accessing the value of an attribute directly. Software designers may choose to replace one strategy by the other one, as the class hierarchy evolves.

1.8 Exception Handling

Poppy supports exception handling through the unwinding of the execution stack. Any method may raise an exception using the `throw` keyword, typically followed by an expression constructing an `Exception` object of some kind, which encapsulates the data relating to the reported failure. Handlers are indicated by special `catch` blocks, which may appear after *any* standard block. There is no need to wrap checked portions of code with a special `try{…}` construct, nor is there any need to declare in advance which exceptions can be thrown by a method (as in Java). It is assumed that the compiler may detect statically which exceptions are thrown, or caught, within a given block scope, such that the usual binding mechanism may be relied upon to propagate uncaught exceptions to the next outer scope. A compiler may warn that certain exceptions are raised.

For example, a `Stack` class may raise exceptions when the preconditions of its operations are violated, expecting these to be handled by the caller (or not at all):

```
Stack pop {
      if (count == 0) throw EmptyCollection(“pop”);
      count := count – 1;
      self
}

Element top {
      if (count == 0) throw EmptyCollection(“top”);
      array[count – 1]
}
```

Any uncaught exceptions propagate up through the program execution stack. As this unwinds, suitable handlers may be found at any level. If so, they get to deal with the exception; otherwise, it may eventually propagate up to the top level, where a backtrace should be reported.

Exception handlers are sensitive to the kind exception raised. A series of handlers may be placed after a block, each seeking to handle a specific class of exceptions. Poppy’s run-time type analysis mechanism is used to select the first handler matching the type of an exception. Typically, handlers are ordered from specific to general, to allow the more specific handlers to deal with the exception first, up to the most general `Exception` handler, which will catch all remaining exceptions. For example:

```
        {
                // … block in which exceptions are raised
        }
        catch (IOFailure ex) {
                ex.file.close;          // ex wrapped the open file
                output.println(ex)
        }
        catch (Exception any)
                output.println(“Some other failure”);
```

Here, the first handler to be tried is `IOFailure`. If the thrown exception is at least of the `IOFailure` class, then `ex` will be bound to this exception and that `catch`-block will execute, after which control returns to the next outer scope. Otherwise, the exception will match the `Exception` class, whose `catch`-block simply prints a general message. A handler may partly clean up the failure and `throw` another exception. In this case, control returns to the next outer scope (the other handlers in the current scope do not deal with a re-thrown exception).

2. Type System

One of the more original features of Poppy is its type system and associated type-checking algorithm. Poppy is quite unlike other object-oriented languages in the way it treats the notions of *class* and *type*. In most other object-oriented languages, a class is treated as though it were a simple type. In Poppy, a class is a *family of types*. These two notions are formally distinct.

2.1 Class and Type

Poppy supports a second-order notion of *class*, which is distinct from the usual first-order notion of *type*. However, this aspect is hidden in the surface syntax of the language, to keep Poppy as simple and elegant as possible. Class names are used both in type declarations and as constructors. These two uses are disambiguated by their context.

When a class name is used in a type expression, to declare the class of some variable, this is *not* understood as the exact (simple, first-order) type of this variable, but as an upper bound on a polymorphic family of types. For example:

```
Number num;
```

declares `num` to be a polymorphic variable of the `Number` class[3]. A value of any numerical type within this class may be assigned to this variable. For example:

```
num := Integer(25);

num := Complex(3.0, 2.5i);
```

are both valid assignments, assuming that the constructed `Integer` and `Complex` objects belong to types within the family of the `Number` class. When a simply typed value is bound to a polymorphic variable, a type check verifies that the assignment is type correct[4]. The binding forces the types to unify, such that afterwards, `num` is understood to have the same type as the bound value, for the duration of the enclosing scope.

This means that a polymorphic variable may not be bound to two differently typed values within the same scope. The two assignments above must happen in separate scopes, or upon re-entry to the same scope on another occasion. This binding of types causes a kind of *type propagation* to occur, whereby the original `Number` type is replaced by the substituted type (`Integer`, or `Complex`) for the duration of the enclosing scope. This is the heart of Poppy's type substitution mechanism.

[3] In the theory of F-bounds, `num` has the polymorphic type $\forall t <: F\text{-}Number[t]$, where *F-Number* is a type function, parameterised by the self-type, describing the interface of the `Number` class.

[4] In the theory of F-bounds, the constraints: *Integer <: F-Number[Integer]* and *Complex <: F-Number[Complex]* will both hold, if `Integer` and `Complex` extend the interface of `Number`.

When a class name is used as a constructor, to create an object, this is understood to return an object of the exact, simple type requested[5]. So, the constructed `Integer` and `Complex` objects above each have an exact *type*, but may be bound to a variable of the polymorphic `Number` *class*. This is the distinction between the notions of *class* and *type* in Poppy. Where these terms are used in a formal sense, *type* denotes a simple, first-order construct and *class* denotes a polymorphic, second-order construction.

Poppy naturally supports the nesting, by inclusion, of recursively-closed classes, that is, nested classes and subclasses, whose methods accept arguments, or return results of the same kind (viz. of the same *class*). This follows intuitive notions of classification, but is something that cannot be expressed in other object-oriented languages, where the notion of classification is approximated by first-order types and subtyping. For example, in Poppy the `Integer` class is a natural subclass of the `Number` class:

```
class (Number self) … {
      Number plus (Number other) {}
      Number minus (Number other) {}
      Number times (Number other) {}
      Number divide (Number other) {}
      …
}

class (Integer self) extends (Number number, …) {
      Integer plus (Integer other) { … }
      Integer minus (Integer other) { … }
      Integer times (Integer other) { … }
      Integer divide (Integer other) { … }
      …
}
```

This would be impossible to express in C++ or Java, where it would be illegal to redefine the types of the methods[6]. The usual rules of subtyping allow a redefined method to have a more restricted result type, but do not allow a more restricted argument type. The reason why this is possible in Poppy is due to the way in which classes are second-order constructs, rather than first-order types. Subclassing is determined only after the self-types have been unified[7] (by type substitution – see below). The uniform specialisation would not be type-safe in a first-order type system. However, since Poppy supports a second-order type system, the nesting of the classes `Number` and `Integer` is sound. Type errors are trapped statically by the type checking mechanism, which is based on type substitution.

---

[5] Construction takes a double fixpoint, binding the self-type and the `self`-reference recursively. This fixes the self-type to refer to a simple type, and fixes the `self`-reference to refer recursively to an instance of this type.

[6] In first-order subtyping, these redefined methods would violate the rule of *contravariance*, according to which redefined function arguments must be of the same, or a larger type.

[7] The notion of subclassing in Poppy is formally based on *pointwise inclusion*. In the theory of F-bounds, a class S is a subclass of T if $\forall t \,.\, F_S[t] <: F_T[t]$, where F_S, F_T are the functionals (type functions) describing the respective class interfaces.

2.2 Unification and Type Substitution

It is as though a class definition were treated like a parameterised signature. The class identifier (e.g. `Number`, above) is treated more like a *type parameter* than a simple type. A subclass is therefore like an extended parameterised signature, with different variables standing for `self` and the self-type (e.g. `Integer`, above). The two signatures can only be compared after the respective occurrences of `self` and the self-type have been unified. Then, occurrences of `self` in `Integer` and `Number` refer to the same object; and the parametric identifiers `Integer` and `Number` are also unified, referring to the same class.

Unification on two class identifiers always preserves the greatest lower bound of the two classes[8]. Whereas in parametric schemes, the unification would be achieved by a common substitution for the two type variables, in Poppy this is handled by a superficially simpler syntactic mechanism called *type substitution*. Type substitutions occur implicitly when a subclass inherits from a superclass, or when values of more specific types are bound to variables of general class-types. Type substitutions may also be requested explicitly by the programmer, in a style analogous to template instantiation in generic programming.

Let us first consider inheritance. During the resolution of single inheritance, it is always the case that unifying a subclass with a superclass will preserve the bound of the subclass. So, when unifying the self-types `Number` and `Integer`, the resulting class is `Integer`, since this is the greatest lower bound (viz. all types in the `Integer` class are also in the `Number` class, but not vice-versa). It is as though all occurrences of the identifier `Number` inherited from the superclass were replaced by `Integer` (after the merger of features) in the subclass.

A slightly more complicated case occurs during the resolution of multiple inheritance, where a child class may inherit from several parents. Merging the parent classes computes a *type intersection*, since it preserves the greatest lower bound of all the parents. The self-type of the child unifies with the self-type of each of its parents, merging all the classes. Occurrences of the self-type and `self` in methods inherited from a parent class will eventually refer to the child subclass and objects of this kind, since this is more specific than any parent class.

Let us now consider binding. Whenever a variable binding occurs, either as a result of parameter passing, or as a result of assignment, then a type substitution occurs for the duration of the scope in which the binding lasts. So, if a method accepting arguments of the `Number` class receives an object of the exact `Integer` type, then for the duration of the method invocation, the substitution of `Integer` for `Number`, which we write in the usual logical style: `{Integer` / `Number}`, is considered to apply in the method body. Likewise, if a polymorphic variable receives a value of a specific type by assignment (e.g. `num` receiving a `Complex` value), then for the duration of that scope, the substitution `{Complex` / `Number}` is considered to apply. Type substitution therefore requires a kind of *type propagation* throughout the enclosing scope, to ensure that no other incompatible type bindings are requested.

Type substitution must always be done uniformly. If, during the resolution of inheritance, the compiler detects a non-uniform substitution, this will be reported as a static type error. For

[8] Pierce has referred to this as a *type intersection*, because the set of types belonging to the resultant class is the intersection of the sets of types belonging to the two classes individually, before unification.

example, it should *not* be possible to specialise an `Imaginary` subclass out of the `Number` class, having the following method signatures:

```
class (Imaginary self) extends (Number number, …) {
      Imaginary plus (Imaginary other) { … }
      Imaginary minus (Imaginary other) { … }
      Decimal times (Imaginary other) { … } // type error!
      Decimal divide (Imaginary other) { … }// type error!
      …
}
```

This is because the `times` and `divide` methods have a `Decimal` return type (as you would expect); yet this would require the conflicting substitutions: {`Imaginary` / `Number`, `Decimal` / `Number`} which will not be accepted by the type checker. (If `Decimal` and `Imaginary` numbers are treated as distinct classes, we need several overloaded methods for multiplication and division, to handle the different classes of argument and result). Similarly, it is illegal to bind the same polymorphic variable to two different types, within the same scope.

Finally, since merging classes computes a greatest lower bound, type substitutions can also propagate in both directions. For example, if a `Pair` class containing `first : Number, second : Complex` unifies with another class containing `first : Integer, second : Number` then the resultant must be the merged class containing `first : Integer, second : Complex`, in which the following right-to-left and left-to-right substitutions {`Integer` / `Number`, `Complex` / `Number`} have taken place. This is the same algorithm as used in logic programming, when computing a most general unifier (MGU) for two expressions.

2.3 Generic Programming

In Poppy, every class definition is potentially generic, since all variables are declared to be of some polymorphic *class*, which can later be bound to an exact *type*, or restricted to some more specific *class* by unification with another class. Therefore, it was considered superfluous to introduce a wholly *distinct* parametric mechanism for generic programming, as found in other object-oriented languages. Instead, generic programming is potentially available for every polymorphic variable. Consider the familiar circumstance of rebinding the `Object` element-class of a `List` explicitly to the more restricted `Integer` class:

```
class (List self) extends (Sequence super) {
      List add (Object item) { … }
      List remove (Object item) { … }
      Object first { … }
}

List<Object := Integer> intList;
…
```

The `List` type-expression uses the angle bracket notation <…> to enclose a set of more specific *type bindings*. In general, several comma-separated bindings are allowed. Here, the type declaration explicitly rebinds the element-class {`Integer/Object`} throughout the scope of the `List` class declaration. All elements declared of the `Object` class within the scope of the `List` class are thereby restricted to the `Integer` class. Note that it was not necessary to specify in advance that `List` was parameterised by its element-type, as in other languages; rather this rebinding mechanism is generally available, so long as substitutions are performed uniformly and systematically. Note also that explicitly rebinding any element-type of a class causes an implicit rebinding of the self-type at the same time, such that the methods `add` and `remove` return the specialised class `List<Object:=Integer>`, rather than the original `List`.

While polymorphic variables may be restricted, as above, it is more typical to apply the restriction to constructor expressions, and then allow the usual binding rule of assignment to propagate the specific type into the general variable:

```
List intList = List<Object := Integer>;
intList.add(3).add(5).add(7);
Integer elem := intList.first;
```

Because this is a creation expression, the result has an exact type. The result of construction is assigned to `intList`, which has the effect of propagating the exact type into the polymorphic class. Therefore, when an element is later extracted using `first`, it is known to be of the exact `Integer` type.

Type substitution offers a much simpler surface syntax than the usual generic programming style, in which explicit type parameters must be declared up-front for all the polymorphic types. Since every class-declaration in Poppy is polymorphic, this would require an abundance of type parameters, if the standard approach were adopted. Type substitution largely removes the need for explicit type parameters. However, there is a cost. Type substitution is strictly less expressive than full generic programming, because it fails to distinguish different occurrences of the substituted class-identifier, within a given scope. In the above example, *all* occurrences of `Object` were substituted by `Integer`, within the scope of the `List` declaration.

While this is sufficient for most practical uses of explicit rebinding and restriction, there are circumstances in which it would be desirable to distinguish different occurrences of the same class identifier, where different parallel substitutions were anticipated. For example, in the `Pair` class, you might specialise the first and second projections separately, even though these are of the same `Object` class. In this case, Poppy permits new class names to be declared internally:

```
class (Pair self) extends … {
      class First renames Object;
      class Second renames Object;
      First first;
      Second second;
}

Pair<First := String, Second := Integer> strIntPair;
```

Here, the local class names `First` and `Second` are declared as distinct *synonyms* for the `Object` class. Internally, the identifiers `First` and `Second` may be used to prefix any variable declarations, and are understood to have the same class-constraint as `Object`. Technically, `First` and `Second` refer to trivial extensions of `Object`, whose local names are accessible in `Pair`, according to the usual rules of visibility. This allows us to treat the introduction of new names in the same formal framework as subclasses. Practically, `First` and `Second` can be considered as distinct synonyms for `Object`, within the scope of `Pair`, but with the advantage that they are distinguished as far as type substitution is concerned. This restores the full expressive power of parametric polymorphism.

Local class names are subject to the same bound as the class they rename. So, any type instantiating `First` or `Second` must be at least of the type `Object`, and is likely to be of some more specific type. Local class names may be restricted further by redefinition in subclasses. For example, it would be possible to declare:

```
      class First renames PartialOrder;
      class Second renames Number;
```

in some subclass of the `Pair` class. In this case, any type instantiating `First` must be at least some kind of `PartialOrder`; likewise any type instantiating `Second` must at least be some kind of `Number`. Local class names that are re-declared unify with their predecessors; and as before, the greatest lower bound is always preserved.

2.4 Run-time Type Analysis

All of the above generic styles support the *homogeneous* instantiation (or specialisation) of each type parameter at compile time. Type information is propagated into the structures that were specialised, such that any extracted elements are known at compile-time to have more specialised types. Sometimes it is desirable to defer the binding of types until run-time. This supports a *heterogeneous* style of programming. For example, one usage of the `List` class above might define a list containing elements of mixed type:

```
      List<Object := Object?> objList;
```

Appending a question mark “?” to a class name denotes deferring the exact type binding until run-time. This tells the compiler to expect elements of different types, unknown at compile time, but all of which belong to the `Object` class. The element-type may also be restricted at the same time as being declared heterogeneous, using the syntax:

```
      List<Object := Shape?> shapeList;
```

This substitutes the heterogeneous `Shape?` class for the `Object` element class, restricting the `List` to contain elements of mixed types, which are all at least some kind of `Shape`. Where heterogeneous styles are adopted, it is also usually necessary to recapture the most specific class of objects that were inserted into a heterogeneous collection. In most languages, this would require some kind of *type cast*, checked at run-time to ensure that the dynamic type of the object was compatible with the static type of the target variable.

In Poppy, no separate syntax is required for a type cast. In the context of run-time recapture, the following expression is considered to be an assignment *attempt*, rather than a regular assignment, because the right-hand expression is known to be heterogeneous:

```
      Square square := shapeList.first;
```

While it is known that the target variable is of the more specific `Square` class than the expression’s upper bound, the compiler also knows that the expression is a heterogeneous

`Shape?` and could return a value satisfying `Square`. The attempt either succeeds, in which case `square` acquires a valid object reference, or it fails, in which case `square` is set to `null`. The program may check for `null` and continue accordingly. However, repeatedly assigning and checking variables could be slightly heavy-handed. So, the following `type`-selection mechanism may be used:

```
type (shapeList.first) {
  (Square square) :
     … // do something with square
  (Circle circle) :
     … // do something with circle
}
else
     … // report failure
```

This works in a similar way to the `case` statement, but selects on the `type` of the expression, rather than the value. A hetereogeneous expression is evaluated, and if the resulting object has a type that matches one of the class-specifiers, which are tried in sequential order, then a variable of that type is bound to the object and may be referred to in the following simple, or compound expression. Only one branch may be executed. If none of the class-specifiers matches, an optional `else` expression is evaluated. The result of the `type` statement is the result of one of its branches, each of which should return a common, or related type. If no case labels match and no `else`-expression is provided, the result is `null`.

Generally speaking, it is not considered good programming style to have to perform lots of runtime type recapture, since this kind of code is fragile, subject to modification every time a new subclass is invented. A better approach is to define a general interface supported by all of the types in the heterogeneous class and rely on the usual dynamic binding mechanism.

2.5 Type Binding Rules

The Poppy type-checker works by propagating exact types into class type parameters, and by unifying type parameters and calculating their greatest lower bound. All typing judgements are made within a particular *scope*. A new scope is introduced by each class definition or method body. Control constructs that introduce new code blocks (delimited by braces `{}`) also introduce new scopes. Within a given scope, *type bindings* may occur, in which more specific types propagate into parameters through assignment and argument passing (including the returning of the result from a method).

Consider first an example of assignment to a polymorphic variable, which binds an exact type to a class parameter:

```
{     Shape shape;     // Variable shape has the class Shape
      …
      shape := Square(Point(3,5), 5);
      …                // Shape now has the exact type Square
      shape := Circle(Point(3,4), 5);
                       // Type error – rebinding to type Circle
}
```

Here, the shape variable is initially declared to have the polymorphic class Shape (which behaves like a parameter). It is bound, in the given scope, to an exact instance of Square. This is only valid if the type Square belongs to the class Shape. The type checker can determine this by a static analysis of the class hierarchy. Because of the assignment, the exact type Square is propagated into the variable's class parameter, resulting in the local type substitution {Square / Shape} in the current scope. Because of this, the subsequent assignment of a new Circle instance fails, because this would require the conflicting substitution {Circle / Shape} in the same scope.

Consider now an example of method invocation, in which the polymorphic class of self and the method argument other are both rebound to more specific types:

```
class (TotalOrder self) extends (PartialOrder super) {
      …
      class Other renames TotalOrder;
      …
      TotalOrder minimum (TotalOrder other) {
            if (self.lessThan(other)) self else other
      }

      Boolean lessThan (Other other) {}
}
```

Here, the method minimum expects self and other both to be of the same TotalOrder class (since the result, also a TotalOrder, is either one or the other). The general algorithm for minimum is expressed in terms of the abstract method lessThan, which is defined in subclasses. Now, this is interesting, because lessThan allows its argument to be of a different class, Other, than the self-class TotalOrder. In general, lessThan could compare two things of different types[9].

However, within the context of the minimum method, lessThan is invoked on two objects of the same type. This results in a type unification of the two class parameters: Other := TotalOrder for the duration of the scope of lessThan. The type checker can verify that minimum needs a version of lessThan accepting two objects of the same type, within the TotalOrder family.

Now, when the minimum method is applied to some Integer values:

```
Integer target := 5;
Integer argument := 9;
Integer result := target.minimum(argument);
                                  // type substitution
```

the exact Integer type of the target 5 and the argument 9 is propagated into the parameter TotalOrder. Likewise, the same Integer type is propagated into the lessThan method, resulting in the type substitution: {Integer / TotalOrder} for the duration of the scope of minimum. It is as if minimum temporarily had the exact type:

[9] For example, when testing the subset relationship between two different set-types having different implementation policies, but a common abstract algorithm for determining subsets.

```
Integer minimum (Integer other) {
     if (self.lessThan(other)) self else other
}
```

and the body of this method is only well-typed if there exists a `lessThan` method which accepts two `Integers` for its target and argument. The variable `c` receiving the result of `minimum` expects this to be of the `Integer` class. This could only be type safe, if the method returned a result within the `Integer` class, which is more restricted than the declared result type: `TotalOrder`. However, as a consequence of type substitution, we know that the result is of the exact `Integer` type, which is suitable. So, we see that a number of different type unifications and checks are performed during method invocation.

2.6 Type Checking Rules

Poppy overcomes the lack of expressiveness in other object-oriented languages, by allowing the definition of recursively-closed classes and subclasses, whose methods accept arguments and return results of the same type as the self-type. This traditionally causes problems for type checkers based on simple types and subtyping, which forbid the retyping of any methods closed over the self-type.

Consider an example of generic addition. The base class `Number` defines addition with the signature: `Number plus (Number other)`. This is retyped in the `Integer` class to have the signature: `Integer plus (Integer other)`, which would conventionally violate the rule of *contravariance*[10] in a subtyping scheme. However, in Poppy, we can safely invoke the `Integer` version of addition through a polymorphic variable of the `Number` class, without risk of type failure, because of the constraint that type substitution brings:

```
Number target := 5;
Number argument := 7;
Number result := target.plus(argument);
```

In the above, a type substitution `{Integer` / `Number}` occurs during the assignment of the first `target` variable. Likewise, a similar substitution occurs during the assignment of the second `argument` variable. By the time `target.plus(argument)` is evaluated, we know that we are invoking the version of `plus` that is closed over the exact `Integer` type (this is determined by static type analysis, and need not appeal to dynamic binding). The result of this type-adapted version of `plus` has the `Integer` type, which is also the type expected at the call-site.

If we try to break the type system, by performing addition over two different numerical types, we can show how this is prevented by the type checker, even though the numerical types are both kinds of `Number`:

```
Number target := 5;
Number argument := Complex(3.71, 2.94);
Number result := target.plus(argument);     // type error!
```

[10] Contravariance: the type rule stating that, in a subtype method, the argument type should be the same, or of a more general type, than the corresponding argument in the supertype method.

The first assignment causes the type substitution {Integer / Number}. The second assignment causes the substitution {Complex / Number}. By the time plus is invoked on the target, the type checker can determine what overloaded method signatures exist for plus in the class Integer. If we assume that there is only one suitable method, with the type signature: Integer plus (Integer other), then attempting to pass an argument of the Complex type is immediately detected as a type error.

The difference between this and the usual typechecking algorithm applied in schemes based on simple types and subtyping is that, in Poppy, type substitutions are carried out before the types of method invocations are checked. In a standard type-checking scheme, the invocation of plus on a variable of the type Number would be checked statically, but only up to the base type Number. The above call would be considered statically valid (assuming Integer and Complex were subtypes of Number), but would give rise to a dynamic type failure, when Integer's plus method was invoked dynamically through the target variable and received an argument of the Complex type[11].

2.7 Type Conversions

On the whole, Poppy does not favour *automatic* type conversions between the different Number subclasses. In languages like C++, a complicated set of precedence rules apply to arithmetical expressions of mixed type, resulting in automatic promotions to the common base types, or secret conversions using constructors, as the compiler tries to find a route from one type to the other. We believe this gives rise to faults, particularly in the lossy conversions (truncation, or loss of precision).

However, Poppy allows explicit type conversion, where this is desired. For example, the Integer and Natural number classes provide the explicit conversion method toDecimal to convert an integral number to its corresponding double-precision floating point representation.

It would also be permissible to define multiple versions of the same method, overloaded on disjoint classes of argument. So, as well as the closed version of plus described above, it would be possible for Integer to support mixed-type versions of plus, which invoked the explicit conversion methods internally:

```
class (Integer self) extends … {
      Decimal plus (Decimal other) {
            other.plus(self.toDecimal)
      }
      …
}
```

This would use a technique called double-dispatch for determining the type conversion most acceptable to both classes of argument. In practice, the library Integer class does not provide this and programmers are expected to use explicit type conversions.

[11] This kind of example was used by William Cook in his famous paper on type failure in object-oriented languages, which allowed covariant specialisation of method arguments as well as results: W R Cook, "A proposal for making Eiffel type safe", *Proc. European Conf. Obj.-Oriented Progr.*, ed. S Cook (Cambridge : CUP, 1989), 57-70.

3. Programming Idioms

Poppy strives to provide a small and efficient kernel of classes, while supporting a rich lattice of overlapping reusable concepts. The overall root of the class hierarchy is called `Top`, which is a vacuous abstract class with no behaviour. The only subclass of `Top` is called `Object`, which defines the basic notion of identity. The notion of equality is first introduced in `PartialOrder`, specialised by `TotalOrder`. `Number` is the abstract ancestor of numerical types, which may be either totally-ordered (such as `Natural`) or partially-ordered (such as `Complex`). A hierarchy of `Collection` classes also appear under `PartialOrder`, for which the notion of being less than another collection translates smoothly into set-inclusion, or bag-inclusion. Certain collections are `Sequences` and the usual `String` class is in fact a sequence of `Character` elements. One totally-ordered subclass is the `Constant` class, ancestor of all enumerated constants.

In this way, Poppy supports a greater natural generalisation among data type concepts and abstract operations than found in many other object-oriented languages[12]. This means that common programming goals should be achieved by adhering to the natural architectural styles supported by Poppy, rather than by reimplementing from scratch. Some of the distinctive programming idioms are described in this section.

3.1 Symbolic Constants

Poppy supports the definition of classes that have a finite set of symbolic instances. Such classes inherit from the kernel class `Constant` and declare *shared attributes* naming all of the symbolic instances of the class. Such constant-classes have no further dynamically-created instances, but may define methods that act upon their constant instances.

Poppy has no separate syntax for enumerations. Instead, all classes, which declare *shared attributes* having the *same self-type as the owning class*, are treated as constant-classes. The names of the shared attributes are treated as the enumerated constants. The programmer may refer to constant names in an unqualified way, where the binding context is unambiguous. For example, the `Boolean` class enumerates two distinguished instances, `false` and `true`:

```
class (Boolean self) extends (Logic logic, Constant const) {
      Boolean false := 0;
      Boolean true := 1;
      …
}
```

and it is possible to refer to the simple constant names without further qualification, for example, when assigning `false` or `true` values to `Boolean` variables, in which binding context the compiler can resolve from which class the constants are selected:

```
Boolean propositionA := true;
Boolean propositionB := false;
```

[12] For example, in Java the `String` class is not a `Collection`; there is no partially-ordered counterpart to the totally-ordered `Comparable` class; and there is no relationship between inclusion among the collections and the usual inequality ordering relationships.

In this way, symbolic constants are *implicitly* selected from their class, rather than explicitly, like other shared attributes (there is no object from which the constants could reasonably be selected). The Poppy compiler is sensitive to constant declarations, as described above, and so will support this style of implicit selection.

In other ways, symbolic constants are exactly like shared attributes: they are initialised as part of their declaration to a given value. However, the initial value must always be a unique `Natural` number; and this is expected by the `Constant` parent class, which provides support for translating between symbolic constants, unique natural numbers and printable strings. For example, the constant class `Choice` declares three symbolic options `cancel`, `no` and `yes`:

```
class (Choice self ) extends (Constant super) {
      Choice cancel := 0;
      Choice no := 1;
      Choice yes := 2;
}
```

Since these shared attributes are of the same `Choice` class, they are recognised as symbolic constants. Each `Choice` option is initialised to a unique `Natural` number. This actually invokes the `copy` constructor inherited from `Constant` to initialise an attribute called the `index`, which is used to identify each constant uniquely, within their class.

All `Constants` are totally ordered (within their class) and must be efficiently coercible into small `Natural` numbers, so that they may be used as the labels in `case`-statements, or as the logical values {`0`, `1`} in `if`-statements. All `Constants` also have a printed form, which is a `String` representation of their symbolic name. To support this, the `Constant` class defines the methods `toNatural` and `toString`. Conversely, it must be possible to construct a constant from its `Natural` index number and from its `String` representation. The `Constant` class defines two `copy` constructors for this purpose.

The conversions to and from strings depend on a private `toString(Natural)` method, which is abstract in `Constant` and defined by each constant-subclass. An example of this for the `Boolean` class is:

```
String toString (Natural index) {
      case (index) {
            0 : “false”;
            1 : “true”
      }
      else throw UnknownConstant(index)
}
```

In this way, the programmer may control the printed representation of constants. As shown here, the method must always raise an `UnknownConstant` exception for out-of-range constant indices.

3.2 Object Comparison

Poppy supports object comparison on the basis of *identity* and *equality*. The `Object` class provides the methods `identity`, which returns a `Natural` number that is unique for each object in the current runtime, `identical`, which compares two objects according to their `identity`, and the complementary `notIdentical`. Subclasses of `Object` may redefine `identity` to take into account other concerns, such as object persistence across multiple runtimes, or duplicate occurrences of the same logical object.

A feature of all comparison methods is that they compare *mixed* types of object. The `Object` class declares a local class `Other` (also a kind of `Object`) for the `other` argument of the `identical` method, which is independent of the self-type:

```
class (Object self) extends (Top super) {
      class Other renames Object;
      …
      Boolean identical (Other other) {
            self.identity.equal(other.identity)
      }
      operator[50] (a == b) a.identical(b);
}
```

This allows *any* two objects to be compared by their identity, not just two objects of the same type. A similar policy is followed for the other comparison methods.

While `Object` is the ancestor of all objects, most classes are either descendants of `PartialOrder`, or of its subclass `TotalOrder`, which define the comparison methods `equal`, `notEqual`, `lessThan`, `moreThan`, `lessEqual` and `moreEqual`[13], all of which compare the states of two objects. (Two objects may be `equal`, without being `identical`).

All of the binary inequalities are initially defined out of a fundamental `compare` method, which compares two `PartialOrder` objects and returns a symbolic `Ranking` expressing the ordered relationship (if any) between the two objects. For example, the `equal` method is first defined in `PartialOrder` as a comparison yielding the symbolic value `equal`:

```
class Other renames PartialOrder;
…
Boolean equal (Other other) {
      self.compare(other) == equal
}
```

This is one possible `Ranking` from the set: `{less, equal, more, meet, none}`. The first three values stand for the ordered relationships *less than*, *equal to,* or *greater than* (the other). The last two values stand for the incomparable relationships *overlapping with*, and *disjoint with* (the other). Partially ordered classes are expected to define their own implementation of `compare`, on the basis of which all the other inherited inequalities will work.

The generic `compare` method is useful in certain contexts, where it is not known whether an ordered relationship exists. Partially ordered comparisons are not often supported in other object-oriented languages, but Poppy may `compare` two `Complex` numbers:

[13] Not a reference to Orwell, but rather a succinct name, chosen in preference to some verbose style mimicking Java: `greaterThanOrEquals`. Similarly, we chose `equal` by symmetry with `identical`, rather than `equals` as used in Java.

```
      Complex cx1 := Complex(3.2, 0.7i);
      Complex cx2 := Complex(2.6, 4.3i);
      Ranking rank := cx1.compare(cx2);     // rank == meet
```

Here, neither number is lesser or greater (in the Cartesian sense) because the real and imaginary parts vary in opposite directions. So, the incomparable result is `meet`, denoting an overlap, or an intersection (in the ordering).

Many commonly used types, such as numbers, characters or logical values, belong to the subclass `TotalOrder`, whose elements are strictly comparable. The implementation policy is reversed in `TotalOrder`, which defines all other comparison methods (including `compare`) out of the `lessThan` operation. This is because all basic types provide efficient primitive implementations of `lessThan` (and also `equal`). All other inequalities are obtained by reversing the operands, or negating the result, of `lessThan` (similar to the strategy of the C++ standard library).

When applied to the `Collection` classes, comparison methods take on a new significance, describing element inclusion. For two `Set` objects, `lessThan` denotes the strict subset relationship; while `lessEqual` denotes the inclusive subset relationship (the opposite relationships denote strict and inclusive supersets). Furthermore, by inspecting the result of `compare`, we may determine whether one set subsumes (`more`), or is subsumed by (`less`) or only intersects with (`meet`), or is completely disjoint with (`none`) another. So, the notion of comparison is truly general; and this obviates the need for any special operations with names like “`subset`” or “`superset`”, which are not provided.

Implementations of `compare` vary in efficiency, ranging from versions that perform two-sided element subtraction (in the class `Bag`, where the cardinality of each element must be respected), to one-sided inclusion tests and fast-failing versions (in the class `OrderedSet`, which respects the element-order in both sets).

3.3 The Collections

Poppy provides a hierarchy of multiobjects, rooted in the ancestor class `Collection`.

[More to follow here]