Handout H: Polymorphism and Abstract Datatypes

Revision 2

Christoph Reichenbach

November 30, 2021

Static type checking can help us us catch errors early. It thus contributes to the robustness of statically typed languages. Static type checking depends on the availability of static type information throughout the program: we must be able to statically assign types to all values and expressions. We usually accomplish this through a combination of automation (e.g., static typing rules) and explicit user specifications (e.g., type declarations for variables).

As helpful as they are for finding bugs, static types are imprecise in practice. Consider division: to ensure that we are not dividing by 0, we could require the right-hand side parameter of the division parameter to be of a type that excludes zeroes (e.g., a subrange type whose lower bound is greater than zero or whose upper bound is less than zero).

While this idea may be appealing, it quickly runs into limitations. Consider the following Mystery example:

```
VAR lower : [-10 TO -1];
VAR upper : [1 TO 10];
VAR combined : [-10 TO 10]
BEGIN
  lower := readA();
  upper := readB();
  IF select()
  THEN combined := lower
  ELSE combined := upper;
  PRINT divide(100, combined)
END
```

Here, the types of `upper` and `lower` guarantee that neither variable can be zero. However, during the **`IF`** statement, we may assign either of them to the variable `combined`. This means that we must give `combined` a type that can contain both the range from -10 to -1 and the range from 1 to 10. There are many types we could choose (such as `INT` or `[-100 TO 25]`), but intuitively `[-10 TO 10]` is the 'best' type that we can give `combined` because it allows all the values that `lower` and `upper` may contain, and a minimum of 'other' values (namely the value zero).

The inclusion of zero is unfortunate: if `combined` had a type that did not include zero, we could be certain that the call to `divide` will not cause a division by zero, but as it stands, the type system does not give us that guarantee. This restriction has real-life consequences: first, the compiler may not be able to optimise the code and may introduce extra code to handle a division by zero that can never happen. Second, if we use this code for a security-critical application, we may not be able to automatically show that it cannot fail and may have to perform an additional audit here.

What we see here is an example of *imprecision* or *conservativeness*: the type system cannot precisely express the possible values that `combined` may take, so it forces us to describe the properties of `combined` more conservatively.

Since at least the late 1970s, researchers have been looking for techniques that allow us to express more interesting and useful types. The focus in this research has been on catching bugs early: while type information remains valuable to compiler optimisation and thus to the task of reducing execution cost, modern compilers have additional techniques for program analysis at their disposal that we are not going to cover in this course[1]

To improve the utility of types, researchers and language designers have had to fend with tradeoffs between the following:

- **Precision**: How accurately can we make types capture the behaviour of a value, expression, subprogram, or other program concept? This concept ties into the question of *Increasing Reliability*.
- **Automation**: How much type-checking can we do while still being certain that the type-checking mechanism will eventually (and, ideally, quickly) finish? This concept ties into the question of *Reducing Compile-Time Cost* and more generally *Reducing Development Cost*.
- **User-Friendliness**: At what point does writing and reading types become too unwieldy to be practical for users? This concept ties into several of the *Readability* and *Writability* criteria.

In this handout, we describe the highlights of this research, that is, type systems that are practical enough to have been integrated into major programming languages. We will focus on *polymorphism*, the idea that part of a program can have multiple (*poly*) forms (*morph*), and related topics. The following sections cover the three key forms of polymorphism that we find in the literature:

1. Parametric Polymorphism (Section 2), which allows us to say 'this value has a type, but you don't need to know what it is'.
2. 'Ad-Hoc' Polymorphism (Section 3), which allows us to say 'this value has a type, and you don't need to know what it is, but here are a few things you can do with it'.
3. Subtype Polymorphism (Section 5), which allows us to say 'this value has a type, and while you don't know what exact type it is, that type is a more restricted (or more general) version of this other type that you *do* know, so therefore there are some things you know you can do with it'.

1 Background: Function Types and Tuple Types

Subroutines play a crucial role in the discussions of this handout, so we will introduce some auxiliary notation to discuss them.

As we have seen in class, some languages allow us to return subroutines as return values, to pass them as parameters, or to store them in variables. In most such languages, the subroutines are stored as *closures*[2]. Subprograms can thus be values in the same way that numbers, tuples, and records are, and analogously have types that comprise the types of both their return values and parameters. Different languages use different notation for the types of subprograms; in MYSTERY, we might write the following for a variable `p` that takes two integers and returns value of type `STRING`:

```
VAR p : PROC(x: INT, y: INT) : STRING
```

While MYSTERY has no support for string values, we can still define the name `STRING` as a type. Figure 1 summarises the notation for other languages. As we can see, the notation varies considerably. The early type systems literature in computer science uses the notation

$$p : \texttt{int} \times \texttt{int} \to \texttt{string}$$

which is reflected directly in Standard ML (SML), with $\texttt{int} \times \texttt{int}$ representing the type of a *tuple of two elements, both of which have type* `int`.

[1]These are discussed in EDAP15: 'Program Analysis'.

[2]As an aside, C and other low-level languages don't use closures but instead provide pointers to executable code; in other words, they support a limited form of closures in which the environment of the closures is always empty. C programmers often work around this limitation by adding an extra 'environment' parameter to these function pointers.

Language	Type	Comments
Mystery	`VAR p : PROC(x: INT, y: INT) : STRING`	`STRING` not built in
Go	`var p func(int, int)string`	
C	`char* (*p)(int, int)`	Not a closure type
Java	`java.util.functional.BiFunctor<Integer, Integer, String> p`	Convenience interface only
Scala	`var p: (Int, Int) => String`	
Haskell	`p :: (Int, Int) -> String`	
SML	`val p : int * int -> string`	
Rust	`let p : fn(i32, i32) -> &'static str`	**`static`** specifies lifetime of string

Figure 1: Comparison of how to declare a variable `p` to have the type of a subroutine from two integers to one string, in various languages

More recent languages, including the more imperative languages Rust and Scala use notation closer to Haskell's style:

$$(\texttt{int}, \texttt{int}) \to \texttt{string}$$

In the following, we will adopt their notation. That is, for the type of a tuple of an integer, a boolean, and a string we will write

$$(\texttt{int}, \texttt{bool}, \texttt{string})$$

and for the type of a function from integers to integers we will write

$$\texttt{int} \to \texttt{int}$$

For the type of functions with multiple parameters of types $\tau_1, \ldots, \tau_k$ we will combine these two notations (as in the above) and write the type of a function that takes a single parameter, where the type of the single parameter is the tuple type $(\tau_1, \ldots, \tau_k)$.

The empty tuple type `()` describes the type of tuples that contain no useful information. This type is also called the *unit type* (e.g. in Scala and SML); languages in the C family call this type *void*. We mostly find the unit type in operations that we call for their side effects. For instance, a printing function might have the type

$$\texttt{printString} : \texttt{string} \to ()$$

while a function that reads input might have the type

$$\texttt{readString} : () \to \texttt{string}$$

2 Parametric Polymorphism

When developing larger software systems, we often develop helper functions for use in more than one location. Consider the following subprogram in a statically typed language with Scala-like syntax[3]:

```
// Incomplete subprogram
def makeIntArray(len : Int, initial : Int) = {
    val result = new Array(len); // create array with 'len' entries
    for (i <- 0 to (len - 1)) {
        result.update(i, initial)
    }
    return result
}
```

[3]This subprogram isn't completely valid Scala yet, we revisit it in Section 4.3 to fix that.

This subroutine builds an `Array` of length `len` in which each element has the value `initial`. Similar operations exist 'out in the wild' e.g. in the standard libraries of languages in the ML language family. With this operation we can now quickly create arrays of any length that are initialised to 0 (useful as neutral elements for addition), 1 (useful as neutral element for multiplication), -1 (perhaps as a marker to indicate that a number is unused in an array that should otherwise only contain positive numbers) and so on, depending on our needs.

We may find this operation useful enough that we may the same operation for building an array for `Float` values:

```
def makeFloatArray(len : Int, initial : Float) = { ... }
```

I have omitted the body of this subroutine because it is exactly identical to the previous one; the only change is in the type of the parameter `initial`. The fact that we had to copy the rest of this code without modification is worrying; clearly we do not want to do the same for all other potentially interesting types. In a dynamically typed language we could have omitted the type of `initial` and reused the code for any (dynamic) type, but at the cost of losing static type checking; if we want to keep the benefits of static type checking but avoid copy-and-paste programming, we have to find a better solution.

To understand the problem, let us turn towards a smaller but very similar problem, namely the so-called *identity function*:

```
def idInt(x : Int) : Int = x
def idStr(x : Str) : Str = x
```

This function just returns its sole parameter, unchanged. The parameter's type occurs in two places: as type of the parameter itself, and as type of the subroutine's return type. This equality between input type and output type is crucial: it does not make sense for an *identity* function that takes an `Int` parameter to return a value of type `Str`.

To allow us to write these two identity functions as one single identity function while ensuring static type checking, we need a new language mechanism. The purpose of this language mechanism is then to *abstract over types* (our `Int` and `Str`) and allow us to *use these types without knowing their exact form*. As it turns out, we have already studied a very similar mechanism in our initial computer science courses, namely *parameters*. We can simply turn the type of the parameter `x` into a *type parameter*:

```
def id[T](x : T) : T = x
```

Just as with other parameters, we now have to specify this parameter when we call `id`[4]. We again use Scala notation:

```
val one = id[Int](1)
val hello = id[String](String)
```

Analogously to parameters to other subprograms, we refer to the type variable `T` as *formal type parameter*, and to `Int` in the subprogram call `id[Int](1)` as *actual type parameter* to `id`.

This feature is called *Parametric Polymorphism*; it is (to the best of my knowledge) universally supported by all nontrivial statically typed functional languages, and by many others, such as Scala, Eiffel, and Java. Java uses the following notation:

```
public class X {
    static <T> T id(T x) {
        return x;
    }
    int x = X.<Integer>id(2);
}
```

[4]Some languages (Scala, Java) are able to automatically determine type parameters statically in some situations.

C++ provides *templates*, a mechanism that is syntactically similar to the notation used by Java (in fact, C++' syntax originally motivated Java's syntax). However, templates are a more general *meta-programming feature* that we will not be able to cover within this course.

Outside of the functional world, language designers often refer to parametric polymorphism as *Generics*, and to subprograms that make use of parametric polymorphism as *generic subprograms*.

3 Typeclasses

Type parameters allow us write subprograms that operate on any type of parameter. We can now easily write e.g. a subprogram that swaps the elements in a tuple, irrespectively of what their types are:

```
def swap[T,U](x : T, y : U) : (U, T) = (y, x)
```

However, the use of type parameters also means that our subroutine no longer knows anything about its parameters. Consider the following subprogram that computes the maximum of two `Int` parameters:

```
def maxInt(x : Int, y : Int) : Int = {
  if (x > y) {
     return x
  } else {
     return y
  }
}
```

Clearly this subprogram should also work for other types, such as `Float` or `String`, so we might like to write:

```
def max[T](x : T, y : T) : T = {
  if (x > y) {
     return x
  } else {
     return y
  }
}                    // WARNING: this will not work!
```

If we ask Scala to compile this code, the Scala compiler will report the following error (with some formatting added and some irrelevant details elided):

```
> is not a member of type parameter T
```

In other words, Scala refuses to accept this generic subprogram because the type parameter `T` has no greater-than operation (>).

This may seem confounding at first — after all, we only intend to use this function with type parameters like `Int` and `Float`, for which the greater-than operation is clearly defined! However, our earlier attempt at defining a generic `max` subprogram placed no such restriction on `T`.

Recall that types effectively model the program and therefore must describe everything that might happen. They are thus contracts between the compiler and the programmer: if the compiler accepts a type for a given subprogram, then this contract requires that the subroutine must also return a result of the matching type, no matter what parameters (type or otherwise) it eventually receives[5]. In our attempt at writing the generic subprogram `max`, this contract was 'the subprogram takes in two parameters of any arbitrary type, but both parameters must have the same type, and the subprogram will return a value of that exact same type'.

What we really wanted, though, was not a subprogram that is completely generic over *every* type `T`, but only one that is generic over some types `T` for which we have an 'is-greater-than' operation.

[5]Most languages allow a small number of exceptions to these contracts, most commonly that the subroutine can 'get stuck' in an endless loop or halt the program if it runs out of memory, but these exceptions do not help us here.

In other words, we want to constrain the type `T`. By constraining `T`, we are asking for a less generous contract from the compiler. There are two main approaches for formulating this constraint; we will discuss the first one here, and the second in Section 7.2. To simplify our discussion, we switch to the language Rust.

The Rust code below is equivalent to the `max` subprogram that we wrote for Scala above:

```
fn max2<T>(x : T, y : T) -> T {
  if x > y {
    return x;
  } else {
    return y;
} }
```

Correspondingly, the Rust compiler will complain to us about this code but helpfully tell us that we can add a type constraint to bound the type variable `T` by a *Rust trait* of the name `std::cmp::PartialOrd`. If we thus change our Rust code to read

```
fn max2<T>(x : T, y : T) -> T where T: std::cmp::PartialOrd {
     ...
```

our code compiles! Moreover, we can now use the `max2` subroutine on various parameters of types for which the greater-than relation is defined, such as the numeric types. If we try to call `max2` on a type for which Rust does not know how to compute the > relation, such as the code

```
  struct MyRecord { ... } // custom record type

  let r1 : MyRecord;
  let r2 : MyRecord;
  let r3 : MyRecord = max2::<MyRecord>(r1, r2);
```

then the Rust compiler will complain at the subroutine call that we have not explained how the greater-than operation works for our `MyRecord` type. Crucially, the compiler reports an error at the site at which we try to *call* `max2`: the compiler successfully compiled `max2` with our updated contract and here reminded us that we violated the contract that we had set up for ourselves.

The general form of this particular form of constraining type parameters was first introduced into the language Haskell under the name *typeclasses*, which is also the language-agnostic term for them. Rust instead calls it *traits*.

Beware: The concept of a typeclass is fundamentally different from the concept of a *class* in object-oriented programming (which we will cover later). Similarly, Rust traits are fundamentally different from Scala traits.

Scala also allows us to use typeclasses, but doing so relies on a feature called *implicit parameters* (subroutine parameters that are passed automatically based on their type) that is specific to Scala; we will not be covering it in this course.

3.1 Defining our own Typeclass / Rust Trait

Languages that support typeclasses (Rust, Haskell, Clean, Scala) allow us to use custom user-defined operations and use them in type bounds. To see this, let us reformulate our `max2` subroutine into a subroutine that works without the built-in > operator and instead relies on the following operation that we will define ourselves:

$$\texttt{greater} : \texttt{(T, T)} \to \texttt{bool}$$

We will require that any type `T` that wishes to be part of our typeclass provides this operation. Below, we formalise this operation as a typeclass `GT`, in the form of a Rust trait:

```
trait GT {
  fn greater(Self, Self) -> bool;
}
```

Here, the keyword `Self` is a reference to whichever type wants to be part of our typeclass (or 'have the trait `GT`', if we use Rust parlance).

Note that typeclasses may ask for more than one operation. We will later see the utility of this feature, when defining abstract datatypes.

With our new typeclass we can now re-define our maximum function as `max3`:

```
fn max3<T>(x : T, y : T) -> T where T: GT + Copy {
  if GT::greater(x, y) {
    return x;
  } else {
    return y;
} }
```

This definition is almost identical to that of `max2`, except that we use the operation `GT::greater` instead of the operator >, and the type bound `GT + Copy` instead of `std::cmp::PartialOrd`.

This type bound `GT + Copy` means any types `T` must be bounded both by our trait `GT`, but also by a special built-in trait `Copy` that Rust requires for values that can be copied. This additional detail is specific to Rust (which has a unique memory and lifetime management system) and does not apply to other languages that use typeclasses.

Before we can now call our function to compute the maximum of two values, we have to make the type of these values a member of our typeclass. For example, to be able to compute the maximum of `i32` values (Rust's signed 32 bit numbers), as in the call

```
  max3::<i32>(7, 13)
```

we have to provide an `implementation` of the operations in the typeclass `GT` for `i32`. The Rust syntax is as follows:

```
impl GT for i32 {
  fn greater(x : i32, y : i32) -> bool {
    return x > y;
} }
```

In other words, we specify a concrete instance of the abstract operation

$$\texttt{greater} : \texttt{(T, T)} \to \texttt{bool}$$

where our concrete instance has the type

$$\texttt{greater} : \texttt{(i32, i32)} \to \texttt{bool}$$

Each such *typeclass instance* (Haskell terminology) or *trait implementation* (Rust terminology) will allow the instantiating / implementing type to participate in any operations that require our type bound `GT`.

The typeclass mechanism in the languages Haskell and Clean is analogous.

3.2 Connection to Operator Overloading

Another way to look at typeclasses / traits is that they give us a means for user-defined overloading [6]. Typeclasses define the operators and operations that we may want to overload, whereas typeclass instances simultaneously declare that a certain type overloads a given operator and explain the semantics of the overloading for that type.

Typeclasses are not the only mechanism for user-defined operator overloading — Sebesta discusses two alternative mechanisms in Section 9.11, one for C++ and one for Python. All of these approaches have their individual strengths and weaknesses, though we wait with a discussion of the differences until we have explored object-oriented programming (which is the foundation to Python's approach).

[6]Cf. 'How to make *ad-hoc* polymorphism less *ad hoc*' Wadler, Blott in POPL'89

Operator overloading is one of the three main forms of polymorphism, under the somewhat derisive name *Ad-Hoc Polymorphism.* This name predates the introduction of typeclasses as a more systematic mechanism for managing overloading, so it is possible that we will in the future refer to this form of polymorphism as *trait polymorphism* or *typeclass polymorphism.*

4 Abstract Datatypes

To see what parametric polymorphism and typeclasses allow us to do, let us take a brief trip back in history. In the late 1960s, software development had reached a crisis: companies like IBM were trying to maintain pieces of software, such as the operating system OS/360, with its ten million lines of assembly code (considered a very large software systems by the standards of the day), but struggled to keep the rate of bug fixes at the same pace as the rate of bug reports. This crisis ultimately led to many innovations, including the invention of the field of Software Engineering. One of these innovations was the direct result of some key observations of the time:

Developers were writing code modules that directly referenced implementation details in other modules. As a result, changes in these other modules risked unforeseen disruptions elsewhere in the code — in other words, once a developer had committed to a particular code or data structure, they could not change it any more, since unknown external modules might depend on those implementation details.

Programming language designers took this insight as a basis for developing mechanisms that would allow software module designers to hide internal (and often accidental) implementation details so that other modules could only access a more restricted (and more intentional) view of each module. They adopted the term *information hiding* for the process of hiding such details.

4.1 Information Hiding and Encapsulation

The researcher David Parnas, who was central in the early work on information hiding, summarised his perspective on information hiding as follows[7]:

> "The major advancement in the area of modular programming has been the development of coding techniques and assemblers which (1) allow one module to be written with little knowledge of the code in another module, and (2) allow modules to be reassembled and replaced without reassembly of the whole system. This facility is extremely valuable for the production of large pieces of code [...]."

He continues to discuss how in a case study, applying the idea of information hiding led to a design where '[a] data structure, its internal linkings, accessing procedures and modifying procedures are part of a single module' instead of split across many modules as was then common. This insight gives us a concrete strategy for how one can hide such implementation details, and led to the idea of *Encapsulation.*

Encapsulation in turn is one strategy for information hiding. Alan Snyder defines it as follows[8]:

> "Encapsulation is a technique for minimizing interdependencies among separately-written modules by defining strict external interfaces. The external interface of a module serves as a contract between the module and its clients, and thus between the designer of the module and other designers. If clients depend only on the external interface, the module can be reimplemented without affecting any clients, so long as the new implementation supports the same (or an upward compatible) external interface. Thus, the effects of compatible changes can be confined.
>
> "A module is encapsulated if clients are restricted by the definition of the programming language to access the module only via its defined external interface. Encapsulation thus assures designers that compatible changes can be made safely, which facilitates program evolution and maintenance. These benefits are especially important for large systems and long-lived data."

[7]David L. Parnas, 'On the criteria to be used in decomposing systems into modules.' Comm. of the ACM, 15(12):1053–1058, 1972.
[8]Alan Snyder: 'Encapsulation and Inheritance in Object-Oriented Programming Languages', Proceedings of OOPSLA'86.

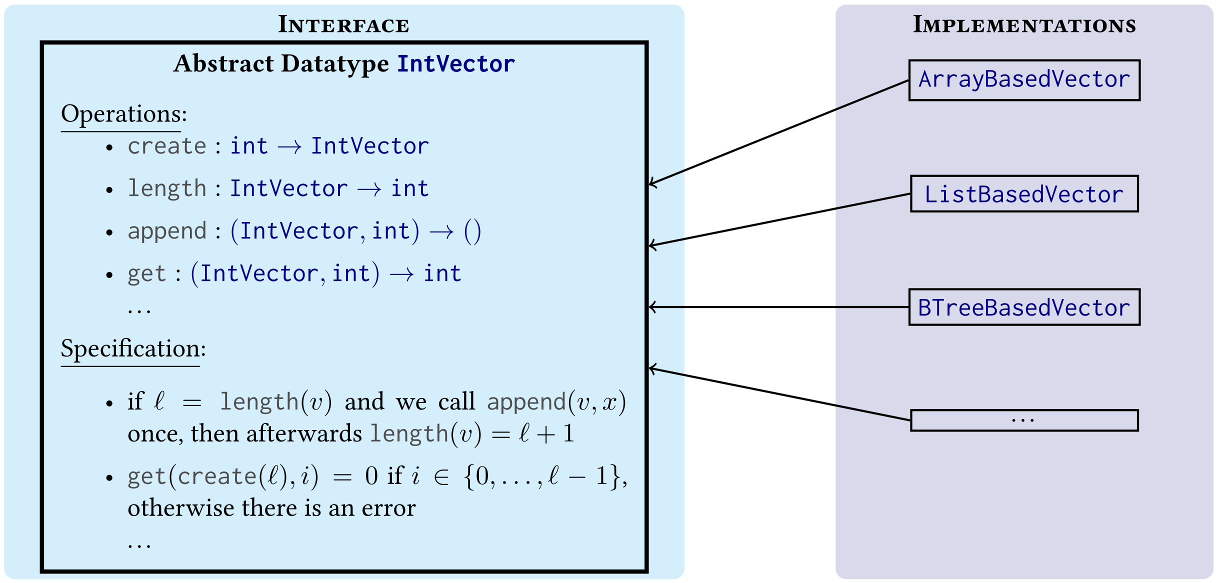
INTERFACE
Abstract Datatype IntVector
Operations:
• create : int → IntVector
• length : IntVector → int
• append : (IntVector, int) → ()
• get : (IntVector, int) → int
...
Specification:
• if ℓ = length(v) and we call append(v, x) once, then afterwards length(v) = ℓ + 1
• get(create(ℓ), i) = 0 if i ∈ {0, ..., ℓ − 1}, otherwise there is an error
...
IMPLEMENTATIONS
ArrayBasedVector
ListBasedVector
BTreeBasedVector
...

Figure 2: Abstract Datatype (ADT) as interface and datastructures that implement the ADT

Encapsulation in Snyder's view then requires explicit programming language support to separate the *interface* to a module from its *implementation*. This means that when we define a piece of functionality, we describe it at least twice: once in the interface (for public consumption) and once in the implementation.

We can also use encapsulation to define new datatypes. This leads us to datatypes whose abstract interfaces can be re-used by multiple implementations. The term for these datatypes is *Abstract Datatypes* or ADTs.

Figure 2 illustrates this idea: here (on the left-hand side) we define a single abstract datatype, *IntVector*, in the style of a heap-dynamic array that allows both appending elements (append) and random access to each element (get). Operations like append, length, and get make up the *interface* of the datatype.

On the right-hand side we list several possible implementations of this datatype: we could implement it using arrays, lists, BTrees, but also in other ways (e.g., storing it on the harddisk or in a database). That is, each abstract datatype can be implemented by potentially many different datastructures. Ideally, this could allow us to switch out one of the datastructures for another without substantially altering the behaviour of the program, e.g., to replace a datastructure with a bug by one without that bug, or to replace a fast but memory-intensive datastructure by a slower but more memory-efficient datastructure when our program must handle more data.

However, to do so we must place stronger requirements on the datastructures than just what operations they should support. After all, the following Python program implements the above interface perfectly well:

```python
IntVector = bool

def create(len: int) -> IntVector:
  return False

def length(v : IntVector) -> int:
  return 0

def append(v : IntVector, value: int):
  pass   # return nothing

def get(v : IntVector, offset: int) -> int:
  return 0
```

Yet this implementation will be entirely useless, as it does not capture the *behaviour* that we want out of an *IntVector* datatype. For that reason, abstract datatypes are accompanied by descriptions of this desired behaviour. Figure 2 lists some in its *Specification* section, requiring (among other things) that calling `append` will result in later calls to `length` returning one more than before the call to `append`.

How to describe these specifications is a field of research in its own right. The main formal strategies revolve around axiomatic specifications (similar to the ones from Figure 2) and operational semantics (such as our natural semantics).

However, there is some disagreement on *how much* we should specify. The more precise the specification is, the more the *user* of an abstract datatype can rely on common behaviour, but the more constrained the *implementer* of a datastructure for such a datatype is. For instance, if we do not specify what happens when we call `get` with a negative index, then this may enable some datastructure implementers to avoid checking the parameter and thus produce more efficient implementations. On the other hand, it also means that for some datastructures, bugs in code that call `get(v, -1)` might go undetected, whereas for others they might not.

Another example of this debate revolves around whether the specifications of ADTs should only include the return types of operations, or also memory and time usage. For example, the C++ Standard Template Library (STL) and the Java standard library specify a number of ADTs along with their asymptotic complexity. In our example, we might require that our `get` operation should take $O(1)$ time over the size of the `IntVector`, which would rule out a list-based implementation of the vector.

As it turns out, Parnas already formulated these ideas in 1971[9], while discussing the idea of combining information hiding in modules with a specification:

1. "The specification must provide to the intended user all the information that [they] will need to use the program correctly, and nothing more."
2. "The specification must provide to the implementer, all the information that [they need] to complete the program, and no additional information [...]."
3. Parnas also asks that the specification should ideally be sufficiently formal to be machine-testable, and
4. that the specification should use language that is usually used by implementers and users to describe concepts in the problem domain into which the ADT best fits.

4.2 Building Abstract Datatypes

ADTs are a close fit to our typeclasses. For instance, we can compactly describe our `IntVector` datatype as a Rust trait:

```
trait IntVector {
  fn create(i32)-> Self;
  fn length(Self)-> i32;
  fn append(Self, i32); // no return value
  fn get(Self, i32)-> i32;
}
```

Of course, this description does not include a behavioural specification, though we could write generic unit tests to automatically check for at least partial adherence to the specification. We can now provide implementations of this ADT as we did in Section 3.

4.3 Generic Abstract Datatypes

While we have now built an ADT for storing integer values, we find ourselves in a similar situation as at the beginning of Section 2: wouldn't it be better if we could use this '`Vector`' ADT to store types other than `int32` values? In other words, can we *parameterise* the `Vector` ADT to abstract out the type of its elements? The literature calls such type-parametric ADTs *Generic ADTs*.

Indeed Rust supports what we want and allows us to introduce a type variable for the elements of our vector:

[9]David L. Parnas, 'A Paradigm for Software Module Specification with Examples', Technical Report, CMU, 1971

```
trait Vector<T> {
     fn create()-> Self;   // changed to always start out empty
     fn length(Self)-> i32;
     fn append(Self, T);
     fn get(Self, i32)-> T;
}
```

The scope of this type variable `T` is now not just the type of an individual subroutine. It is the type of the entire trait (and, consequently, the type of any datastructure that wants to implement this trait).

As before, our datastructures are not allowed to do anything with values of type `T`, since we told the compiler that we want *any* type to fit in. As before, we may want to restrict ourselves to types that provide certain functionality.

As an example, consider a generic ADT priority queue to which we can `push` an arbitrary number of elements of some type. This queue will order the elements that it contains such that the "greatest" element is always the next one that we will retrieve (e.g., the largest number, the most urgent network request, the book with the most positive reviews).

Thus, a priority queue must be able to tell which of its elements is the "greatest". Here, we encode this requirement by asking that elements are comparable via the greater-than operator (>). The solution is the same as before: we restrict our type parameter `T` by requiring it to be an element in a pre-existing typeclass (in Rust, `std::cmp::PartialOrd`). This ensures that the actual type parameter for `T` provides support for comparison:

```
trait PriorityQueue<T> where T: std::cmp::PartialOrd {
     fn create()-> Self;
     fn push(Self, T);
     fn getTop(Self)-> T;
}
```

While Haskell and Clean have a different notion of state and updates (which is why we did not use them for these examples), they support the same features as far as the type system is concerned.

Generic ADTs are widely supported in modern languages. For example, we can define a type `Vector[T]` in Scala that is analogous to the above (cf. Section 7). The code below is then a variation of our earlier 'create a pre-initialised array' code for Scala, with a custom type `Vector[T]` instead of `Array[T]`[10]:

```
  def makeVector[T](len : Int, initial : T): Vector[T] = {
    // create heap-dynamic array with 'len' entries:
    val result : Vector[T] = new Vector[T](len)
    for (i <- 0 to (len - 1)) {
      result.update(i, initial)
    }
    return result
  }
```

5 Subtype Polymorphism

The third and final form of polymorphism that we discuss here is *subtype polymorphism*.

Consider the following variable declarations in MYSTERY:

```
  VAR a : ARRAY[0 TO 10] OF INT;
  VAR x : [0 TO 10];
  VAR y : [2 TO 5]
```

Here, we see that it is statically safe to index the array `a` with the variable `x`, i.e., to write `a[x]`, without any dynamic checking, since the valid indices for `a` are in `[0 TO 10]` and we also have `x : [0 TO 10]`.

[10]Implementing the same example with type `Array[T]` in Scala would require us to provide a *class tag* for `T`, an object that represents the dynamic type of `T`, due to technical complications at the level of the Java Virtual Machine.

However, it is *also* statically safe to write `a[y]`, even though `y` has a type that is different from the array's index range. The underlying reason is that the set that represents the type `[2 TO 5]` is a subset of the set that represents `[0 TO 10]`, i.e.,

$$\texttt{[2 TO 5]} \subseteq \texttt{[0 TO 10]}$$

We say that '`[2 TO 5]`' is a *subtype of* '`[0 TO 10]`', or conversely that '`[0 TO 10]`' is a *supertype of* '`[2 TO 5]`, notation

$$\texttt{[2 TO 5]} <: \texttt{[0 TO 10]}$$

Definition 1 *A type* `T` *is a* subtype *of type* `U`*, notation* `T` $<:$ `U` *or* `U` $:>$ `T`*, if* any *value* v : `T` *can be used in* any *context that requires a value of type* `U`[11]

For example, we have:

- `[2 TO 5]` $<:$ `[2 TO 6]`
- `[2 TO 5]` $<:$ `[1 TO 5]`
- `[2 TO 5]` $<:$ `[1 TO 6]`
- `[2 TO 5]` $<:$ **`INT`**

Moreover, since supertyping and subtyping are based on the subset relation, they inherit two few useful properties: first, they are reflexive i.e., each type `T` is a subtype (and supertype) of itself, `T` $<:$ `T`, and second, they are transitive, meaning that whenever we know that `T` $<:$ `U` and `U` $<:$ `V`, we also know that `T` $<:$ `V`.

Whenever we use a value of a subtype in a place that expects a supertype, the language must perform a *widening* conversion. In most languages that support subtyping, this conversion is implicit, since it is safe. Some languages also allow us to translate a supertype to a subtype, using a *narrowing* conversion. This conversion may fail, so it is usually an explicit operation.

For example, let us assume that `SmallInt` $<:$ `BigInt`. In Java we can write

```
SmallInt small = readSmallInt();
BigInt big = small; // implicit widening conversion

big = readBigInt();
small = (SmallInt) big; // explicit narrowing conversion required
```

where the narrowing conversion may fail at runtime.

Many languages that support subtyping allow us to explicitly check at runtime whether a narrowing conversion will succeed or fail (e.g., using the **`instanceof`** operator in Java). The language Swift even supports a combined check and conversion operation:

```
if let small = big as? SmallInt {
   ... // only executed if big <: SmallInt
}
```

[11]Based on the definition by Kim Bruce, 'Foundations of Object Oriented Languages: Types and Semantics', MIT Press, 2002

5.1 Subtyping and Records

As we defined, subtyping allows us to talk about when values of one type can be used in a place that expects a value of another type. For example, consider the following two record types in Modula-3:

```
TYPE TR1 = RECORD
  VAR x : INT;
END

TYPE TR2 = RECORD
  VAR x : INT;
  VAR y : STRING;
END
```

Any operation that expects a record of type `TR1` (e.g., a subroutine that prints the number from field `x` to the screen) can in principle also work with a record of type `TR2`, so we have

$$\texttt{TR2} <: \texttt{TR1}$$

This may be counter-intuitive at first glance, since `TR2` appears to be visually 'bigger' than `TR1`. To understand why the above makes sense, consider the question of what we can do with values of this type: For `TR1` all we can do is to read/write integers from/to a field `x`, or store them there. For `TR2` we can do all that we can do for `TR1`, *and also* read/write strings from/to a field `y`. In other words, `TR2` satisfies stronger constraints, just like the subrange type `[2 TO 5]` satisfies stronger constraints than the subrange type `[0 TO 10]`.

As we have seen, adding new fields to a record creates a subtype. Changing the types of the fields may also an option, but this depends on the semantics of records in the underlying language, so we will return to this question in Section 5.3.

5.2 Subtyping Subprograms

As we have seen, deciding whether we can assign a value of some type to a variable of another type (or pass it as a parameter) already raises some challenges. These challenges increase when the value that we assign is a *closure*, e.g., when we are passing a function as a parameter to another function.

Consider the following MYSTERY subprogram `sub`:

```
PROC sub(p : PROC(x: [0 TO 1]) : [0 TO 1]) =
  VAR v : [0 TO 1];
  VAR w : [0 TO 1]
  BEGIN
    v := p(0);
    w := p(1);
    ...
  END
```

This subprogram takes a subprogram parameter of function type

$$\texttt{p} : \texttt{[0 TO 1]} \rightarrow \texttt{[0 TO 1]}$$

If we have a subprogram `q` of a different type, we may or may not be able to pass `q` as a parameter to `sub`. Let us carefully consider the cases, assuming in-mode parameter passing (e.g., Pass-by-Value):

1. **Return type**: If our subroutine `q` returns something that is a subtype of `[0 TO 1]` (such as `[0 TO 0]`, which must always be 0), then `sub(q)` will work just fine: the only change is that `v` and `w` will always be assigned 0.

   However, if our subroutine returns a strict supertype such as `INT`, then we could at runtime attempt to assign any integer value to `v` and `w`. Thus, **the return type of `q` must be a *subtype* of the return type of `p`**.

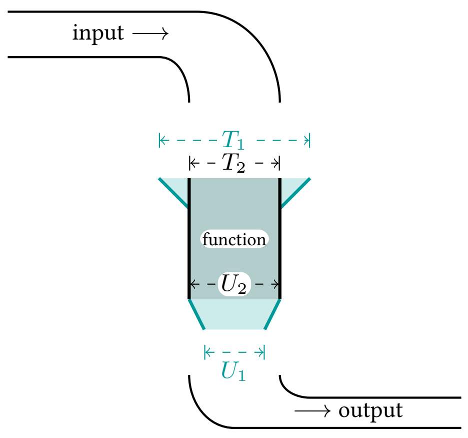

Figure 3: Subroutine subtyping: if we need a subroutine with function type $T_2 \to U_2$, we can instead provide a subroutine of type $T_1 \to U_1$ if T_1 is less constrained than T_2 and if U_1 is more constrained than U_2, in other words, if we think of the function in the middle as a funnel whose opening we can widen and whose end we can narrow.

2. **Parameter type**: If our subroutine `q` only accepts something that is a subtype of `[0 TO 1]` (such as `[0 TO 0]`, then it will not be able to process the call `p(0)` in line 5 or the call `p(1)` in lines 6.

   However, `q` can be more generous about accepting parameters: if it accepts any supertype, such as `INT`, `sub` will still be able to run without a risk of failure. Thus, **the type of the parameter of `q` must be a *supertype* of the type of the parameter `p`**.

Figure 3 visualises these considerations. We can formalise them compactly in the so-called *arrow rule*:

$$\frac{\mathtt{T}_1 :> \mathtt{T}_2 \quad \mathtt{U}_1 <: \mathtt{U}_2}{\mathtt{T}_1 \to \mathtt{U}_1 <: \mathtt{T}_2 \to \mathtt{U}_2}$$

As we will see later, the rule for parameter passing flips around for out-mode parameter passing, since we (in essence) add additional return values.

As with other forms of subtyping, some languages may be less generous than what the most general form of subtyping might allow. For example, early versions of Java (before version 5.0) only permitted parameter types of methods (analogous to subroutines) to change, but not return types.

5.3 Subtyping and Updates

Most languages allow us to pass parameters in a way that persists changes to the parameter, e.g., through Pass by Reference or through some other in-out parameter passing mechanism. For example, many languages from the C family pass arrays by reference. Subtyping for mutable values raises some questions that we didn't see for e.g. our subrange values.

Let us consider the following MYSTERY program, and assume that we pass all parameters by reference or through another in-out parameter passing mode:

```
1  PROC upd(x : [1 TO 5]) =
2    VAR y : [1 TO 5]
3    BEGIN
4      y := x;          ← read from reference cell
5      IF y == 1
6      THEN x := 1
7      ELSE x := 5
8    END
```

As we did with subroutines, we can now carefully consider all cases:

1. **Supertypes**: If we pass a reference of a supertype (such as `INT`), then the read operation in line 4 may produce a result that is not in `[1 TO 5]`. Thus, **We cannot allow a reference to a supertype as a parameter**.

2. **Subtypes**: If we pass a reference of a subtype (such as `[2 TO 4]`), then one of the updates in line 5 or line 6 may fail. Thus, **We cannot allow a reference to a subtype as a parameter**.

In other words, *there is no interesting subtyping possible when passing mutable data.*

This insight helps us understand subtyping of other mutable types. While some languages such as C and Haskell allow passing read-only records (modifiable copies of such records in the case of C), other languages such as Java or Eiffel only allow passing mutable *objects*. In these languages, we cannot safely vary the types of the objects' fields, no matter whether the new type would be a subtype or a supertype.

For the same reason, we cannot safely pass mutable arrays whose element types vary. Despite this observation, the Java language allows arrays of subtypes to be passed in place of arrays of supertypes:

```
void f(BigInt[] bigArray) {
   bigArray[0] = readBigInt();
}

SmallInt[] array;
f(array);  // assume SmallInt <: BigInt
```

The price that Java pays for this feature is that it must perform dynamic checks on any update to arrays of subtypes that are used in place of arrays of supertypes.

5.4 Subtyping and Inheritance

As we have seen, what subtyping a language *can* permit while remaining statically checkable and strongly typed is not necessarily the same as what subtyping the language *does* permit; this is left to the language designer. To see another example of this, let us revisit our record subtyping example and translate it to Java, using Java's classes (which for this discussion we can think of as an enhanced version of records):

```
class TC1 {
  int x;
}

class TC2 {
  int x;
  int y;
}

void initX(TC1 v) {
  v.x = 23;
}
```

Here, Java will *not* allow us to supply a `TC2` in a place that expects a `TC1`, such as in the call `initX(new TC2())`; from the perspective of Java's type system, `TC2` $\not<:$ `TC1`. The reason for this difference is that Java does not automatically consider two types to be subtypes simply because they are structurally suitable to be subtypes; instead the language requires developers to explicitly *declare* that two types are subtypes of each other.

To declare their relationship, we can rewrite the above class definitions as follows:

```
class TC1 {
  int x;
}

class TC2 extends TC1 {
  // Automatically inherit field x
  int y;
}
```

Here, `TC2` states that it **`extends`** the type `TC1`. The **`extends`** declaration is is one of the two mechanisms through which user-defined Java types can declare themselves to be a subtype of another type[12]. A second effect of the **`extends`** declaration is that `TC2` automatically obtains all fields declared for `TC1`; this mechanism (part of the object-oriented principle of *inheritance*) conveniently ensures that we cannot forget to add any fields that `TC2` would need to be a subtype of `TC1`.

5.5 Nominal and Structural Subtyping

As we saw, Modula-3 and Java have two different mechanisms for subtyping of records (Modula-3) and classes (Java). Specifically, Modula-3 uses *structural subtyping* of records:

Definition 2 *A type constructor in a language uses* structural subtyping *if two different types* `T`, `U` *constructed from this type constructor can be in a subtyping relation without being explicitly declared to be in a subtyping relation.*

Meanwhile, Java uses *nominal subtyping* of classes:

Definition 3 *A language uses* nominal subtyping *for a type constructor if two different types* `T`, `U` *constructed from this type constructor can be in a subtyping relation, but only if they are explicitly declared to be in this relation.*

The advantages and disadvantages of structural and nominal subtyping are analogous to those of structural and nominal type equivalence.

[12]The other form of declaration is the **`implements`** declaration for Java interfaces, cf. Section 7.

5.6 Formalising Subtyping

If we try to type-check a program that uses subtyping, we must now consider the language's subtyping rules when performing type-checking. For example, consider the following rule:

$$\frac{x : \texttt{int} \quad y : \texttt{int}}{x + y : \texttt{int}} \ (\textit{add})$$

If we have a variable `z`: `[0 TO 5]`, we still want to be able to type-check the addition `z + 10` and give it a meaningful type such as `int`, even if we now require a widening conversion on `z`.

This means that we need a formal rule that allows us to widen a value to have the type of a supertype. This rule is the so-called *subsumption rule*, which languages with subtyping use as part of their type system:

$$\frac{v : \texttt{T} \quad \texttt{T} <: \texttt{U}}{v : \texttt{U}} \ (\textit{subsumption})$$

This allows us to check the type of `z+10` as follows:

$$\frac{\dfrac{z : \texttt{[0 TO 5]} \quad \texttt{[0 TO 5]} <: \texttt{int}}{z : \texttt{int}} \ (\textit{subsumption}) \quad \dfrac{10 : \texttt{[10 TO 10]} \quad \texttt{[10 TO 10]} <: \texttt{int}}{10 : \texttt{int}} \ (\textit{subsumption})}{z + 10 : \texttt{int}} \ (\textit{add})$$

A direct consequence of the subsumption rule is that in languages with subtyping, most values have more than one type.

6 Subtyping in Object-Oriented Programming

As Sebesta describes, the necessary components of object-oriented programming languages are

1. Dynamic method binding or *dynamic dispatch*
2. Inheritance, and
3. Support for Abstract Datatypes

Statically typed object-oriented programming languages complement these three features with *subtyping* (Section 5.3).

The subtyping rules in major object-oriented languages follow the rules that we discussed above. The main complication here comes about due to *method overriding*. Consider the following classes in Java-like notation (though the concepts translate directly to Scala, C++ with virtual member functions, Eiffel etc.):

```
class A {
  String name() { return "A"; }
  String f() { return "from " + this.name(); }
}

class B extends A {
  @Override
  String name() { return "B"; }
}
```

After these declarations, we have `B` $<:$ `A`.

We can thus pass any B instance into a context that expects a A type:

```
A v = new B();
```

If we now call v.name(), then dynamic dispatch dictates that Java will return the String "B". Thus, the language's run-time system must be able to know that we want to call B.name(), rather than A.name().

The same is true for indirect calls: if we call v.f(), then we will still be calling the method A.f() (since B has inherited this method from A without overriding it), but the call to this.name() within A.f() must now call B.name() to produce the expected result (the String "from B").

This means that the object that we store in variable v must be able to remember what type it was created with originally; otherwise dynamic dispatch cannot work. Thus, variable v has two type bindings: a *static type binding* (which here is to A) and a *dynamic type binding* (which here is to B).

6.1 Subtyping and Methods

In our previous example, we overrode the method

$$\texttt{name} : () \rightarrow \texttt{String}$$

from class A in class B with a method of the same type. The subtyping rules for methods follow the subtyping rules for subroutines, so the following overriding between classes C and D is perfectly safe:

```
class C {
  A g(B b) { ... } // type B -> A
}
class D extends C {
  @Override
  B g(A a) { ... } // type A -> B
}
```

since

$$\texttt{A} \rightarrow \texttt{B} <: \texttt{B} \rightarrow \texttt{A}$$

One complication here is the implicit **self** or **this** reference (or pointer, in C++). This self-reference allows methods to access their own state, so its type is always fixed to be a subtype of the type of the declaring class.

This self-reference is technically a 'hidden parameter' to each method, but since a method can only be called on an object whose dynamic type is a subtype of the class in which that method was defined, it is safe for us to use this more precise type information.

6.2 Method Overriding

Many object-oriented languages permit both method overriding and method overloading (analogous to subroutine overloading) at the same time. That is, they allow the definition of two methods of the same name but with different types in the same class. Consider:

```
class E {
  int f(B b)  { ... } // defines method:  int E.f(B)
  int f(A a)  { ... } // defines method:  int E.f(A)
  int g(A a)  { ... } // defines method:  int E.g(A)
}
class F extends E {
  @Override
  int g(A a)  { ... } // defines method:  int F.g(A)
}
```

The example above illustrates both overloading and overriding: `E.f(B)` and `E.f(A)` are two methods that have the same name but different parameter types, they are thus *overloaded.* Meanwhile, `F.g(A)` *overrides* the method `E.g(A)`.

The difference here is that we know *statically* which of the two possible `f` methods we will call, whereas we cannot (in general) know until run-time which of the possible `g` calls we will make.

```
A aa = new A(); // static type: A, dynamic type: A
A ab = new B(); // static type: A, dynamic type: B
E ee = new E(); // static type: E, dynamic type: E
E ef = new F(); // static type: E, dynamic type: F

ee.f(aa)    // => E.f(A)
ee.f(ab)    // => E.f(A)  (!)
ee.g(aa)    // => E.g(A)
ef.g(aa)    // => F.g(A)
```

That is, when trying to resolve overloading, the language uses the *static* type binding; even though `ab` has the dynamic type `B`, `ee.f(ab)` will invoke `E.f(A)`.

When trying to resolve overriding, the language instead uses the dynamic type, performing dynamic dispatch.

Note that typeclasses (Section 3) as a technique for systematic overloading are also statically resolved, though the *implementation* of typeclasses (at the compiler level, not visible to programmers) is similar to the implementation of dynamic dispatch.

6.3 Multiple Dispatch

The combination of (static) overloading and dynamic dispatch can be confusing to use in practice. Some languages (the most mainstream of which are Common LISP and Julia) have thus adopted an enhanced form of dynamic dispatch called 'multiple dispatch'. The idea here is to allow an alternative to overloading in which the *dynamic types of actual parameters* determine which method gets called.

In our previous example, a language with dynamic dispatch would execute the call `ee.f(ab)` by dispatching to method `E.f(B)`.

One challenge that languages with dynamic dispatch face is what to do in case of ambiguity. Consider the following situation (here presented in Java syntax):

```
void f(A a, B b) { ... }
void f(B a, A b) { ... }
```

If we now try to call `f(new B(), new B())`, it is not clear which of the two calls to make. That means that at run time, this code run into a situation in which there is more than one way to continue execution. The Julia language handles this situation by aborting with an error.

While we can have an analogous issue with overloading, the compiler can detect ambiguities in overloading (since overload resolution happens statically), so the compiler can ask the user to resolve the ambiguity (e.g., by providing an explicit widening conversion to one of the parameters).

6.4 Classes and Types

In class-based object-oriented languages, each class gives us one type (or a type constructor, as we will see later). However, there may be types that do not correspond to classes. One example of types without classes are Java's *primitive type* `int`; C++ similarly has a large number of primitive types (mostly inherited from C) that are not class-based. User-defined enumeration types in C++ are another example; In Java, a user-defined enumeration will also create a class that corresponds to that enumeration, though Java does not support inheritance over enumerations.

It is thus important to distinguish these two concepts. While a class always contains a type that describes the class' method types (and possibly other information, such as the types of visible fields and constructors), the class also contains the *implementations* of its various methods. When one class inherits from another class, it obtains ‘copies’ of these fields and methods from its parent class[13].

The type, meanwhile, is only concerned with the operations that we can perform on class instances (method calls, and possibly reads from/writes to fields).

Both Java and Scala provide a mechanism for defining new types that describe families of operations that are not classes. In Java, these types are called *interfaces*, and *traits* in Scala. Interfaces and traits may be subtypes of other interfaces and traits, and classes may be subtypes of interfaces and traits, but since there are no implementations to inherit from an interface or trait, these two do not act as classes.

6.5 Multiple Subtyping and Inheritance

So far, we have only considered cases where a class has exactly one superclass. However, it is not necessary for a class to have a superclass at all. Recall that if `A` $<:$ `B`, then we cannot also have `B` $<:$ `A` unless `A` $=$ `B`, so with a finite number of classes there must be some class ‘on top’ that has no superclass. In Java, that is the class `java.lang.Object`, in Scala it is `Any`. Other languages may or may not have a single ‘root’ class; C++ for example permits any number of top-level classes without superclasses.

Classes in Java and Scala can have at most one direct superclass. They can however have any number of super*types*. Consider the following class `B`:

```
class A { ... }
interface I { ... }
interface J { ... }
class B extends A implements I, J {}
```

This class has five supertypes: `B` (itself), `A` (its immediate parent class, which is also a type), the interfaces `I` and `J`, and `java.lang.Object`, which is the direct parent class and parent type of `A`.

C++ instead permits multiple superclasses (and multiple inheritance). Section 12.3.3 in Sebesta covers the key concerns about multiple inheritance.

6.6 Implementing Objects

Compilers and interpreters for object-oriented languages use different strategies for making subtype polymorphism and inheritance work. The most common of these (used in e.g. JVM and .NET-based languages, as well as (with minor tweaks) in JavaScript and Python) is to place all objects on the heap[14] in a data structure that (simplified) looks like this:

```
TYPE Object = RECORD
  VAR typeinfo : INT;
  VAR methods : REF ARRAY[0 TO *] OF PROC(*):*;
  VAR fields : UNION
    VAR type_0_fields : RECORD ... END;
    VAR type_1_fields : RECORD ... END;
    VAR type_2_fields : RECORD ... END;
    ...
  END
END
```

The data structure consists of three parts:

- A `typeinfo` tag that tells us the concrete dynamic type of the object
- A reference to a `methods` array that contains all methods for the dynamic type. This array is also called the “virtual method table” (or `vtbl` in C++).

[13]The technical implementation is usually more efficient than copying and instead shares the common information

[14]By default — the Java virtual machine can often optimise this case, but other than through reduced execution time and memory usage, this optimisation is invisible to the programmer.

```
class A {
  int unused = 3;
  String f() {return "A.f";}
  String g() {return "A.g";}
}
```

```
class B extends A {
  int count = 1;

  @Override
  String g() {return "B.g():"
            + (this.count++);}
  String h() {return "B.h";}
}
```

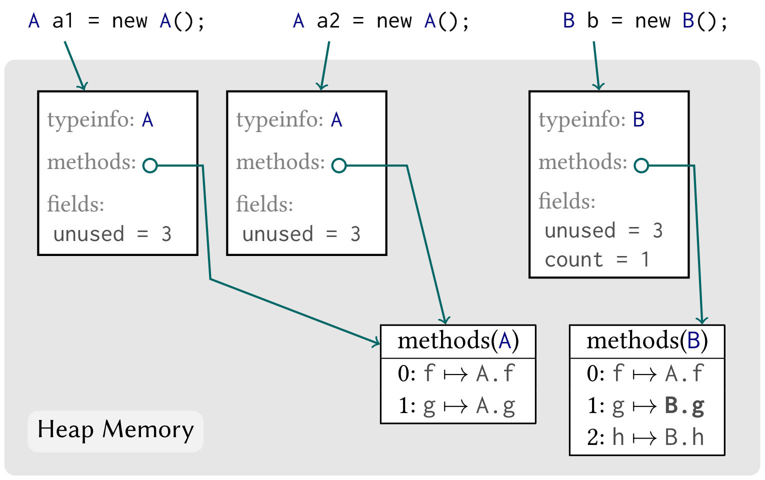

Figure 4: Implementing Subtyping. Class B <: A, and the language (here: Java) uses dynamic dispatch. To speed up dynamic dispatch, the language aligns the virtual method table for classes A and B (and any other subclasses) so that any indices into the method table for e.g. A are also valid for any subclass, e.g., B, and automatically resolve overriding. Fields are aligned similarly.

- A union of `fields`, with one variant for each of the types in the program. This is a *tagged union*, and `typeinfo` acts as its tag.

Concretely, consider the classes A and B in Figure 4: the memory layout for the method tables is aligned. If a piece of code wants to call the `g` method on an object of static type A, the language can call the method at offset `1` in the array that `methods` points to, i.e., `methods[1]()`. When we call `a1.g()`, then `a1.methods[1]` gives us `A.g` (i.e., the machine code for the method `g` in class A), and for `b.g()`, loading `b.methods[1]` gives us `B.g`. This allows the language to implement overriding relatively efficiently.

Since `typeinfo` uniquely determines the `methods` pointer, some language implementations combine these two fields (e.g., by reserving `methods[0]` to to store `typeinfo`) to save space.

Java interfaces or multiple inheritance require more complex techniques but also exploit the above.

6.7 Objects and Primitive Types: Boxing and Unboxing

The previous subsection described how modern language implementations lay out objects on the heap. However, variables of primitive type do not normally follow this layout, and it would be inefficient to force all of them into this format. (E.g., a `bool` takes up a single bit of memory, whereas a Java object takes up at least `12` bytes in the current generation of virtual machines).

C++ and early versions of Java therefore treat primitive types as strictly separate from object types. Consider the following Java method that finds the array index of the first occurrence of some object `key` in some array `objects`:

```
public static void findFirstIndex(Object key, Object[] objects) { ... }
```

Since the method takes `Object` parameters, it will work on any `Object`, e.g., `findFirstIndex("foo", array)`. However, it would not work on primitive types, e.g., `findFirstIndex(100, array)`. To address this problem, Java introduced, for each primitive type, an alternative type that represents that underlying primitive type:

```
  int i = ...;
  Integer i_box = new Integer(i);
```

`Integer` is the **boxed** representation of `int` values: it is nothing more than an object that stores a single `int`. We refer to the process of moving from a primitive type to a boxed representation as **boxing**, and for the process of moving from a boxed representation back to a primitive value as **unboxing**.

In newer versions of Java (and in most other languages that use boxing), the conversion is implicit. Java will automatically insert the box and unbox coercions in the code below:

```
Object o = box(7);
Integer i_box = new Integer(i);
Integer sum =  box( i_box.unbox() + 3 );
```

7 Abstract Datatypes in Object-Oriented Languages

Subtyping gives us a second mechanism for implementing abstract datatypes. Consider the following Java type:

```
interface IntVector {
  int length();
  void append(int value);
  int get(int offset);
}
```

Or the analogous declaration in Scala:

```
trait IntVector {
  def length : Int
  def append(v : Int)
  def get(v : Int) : Int
}
```

This declaration specifies a variant of our abstract datatype `IntVector` from Figure 2 as a Java interface or Scala trait. Any subclass of either of these types must now provide concrete implementations of the three methods declared in `IntVector`, so any subclass of `IntVector` is an *implementation* of the abstract datatype whose interface we have specified here, and due to subtyping we can now write code that relies on `IntVector` (here in Scala):

```
def add12(v : IntVector) : Int = {
  v.append(12)
  return v.length
}
```

Our `add12` method can use the `IntVector` `v` without knowing which implementation we are providing. Unlike with typeclasses, we do not even need to keep a type parameter for the `IntVector`; instead, the necessary type information is hidden in the dynamic type of `v`.

7.1 Generics

We can combine subtyping with type parameters to make our vector able to handle any element type:

```
trait Vector[T] {
  def length : Int
  def append(v : T)
  def get(v : Int) : Int
}
```

Both Scala and Java refer to this availability of type parameters in the presence of subtyping as *Generics*, since they permit the implementation of generic abstract datatypes (Section 4.3).

7.2 Bounded Parametric Polymorphism

Let us now define the interface of a priority queue, as in Section 4.3. Recall that we want our priority queue to be generic, but at the same time we only want to allow element types that allow us to compare objects against each other so that we can keep the priority queue sorted and always find the 'top' element.

We can express this requirement in another Scala trait:

```
trait Cmp {
  def isGreaterThan(v : Any) : Boolean
}
```

This trait requires any subtype to provide an operation that allows comparing against *any* object. We can now define the Scala trait for our priority queue:

```
trait PriorityQueue[T <: Cmp] {
  def push(v : T)
  def getTop() : T
}
```

Here, the type constraint `T <: Cmp` ensures that `T` must be a subtype of `Cmp`.

7.3 F-Bounded Parametric Polymorphism

While our type bound in the previous example gives implementers enough information to build a priority queue, it is also slightly too strong. `Cmp` requires all subclasses to write code that can compare their own state against *any* other object. However, in a priority queue of integers, we only need to be able to compare integers with other integers. Thus, we can relax the requirements on our element types somewhat. We might instead of `Cmp` use the generic type `GT`:

```
trait GT[T] {
  def isGreaterThan(v : T) : Boolean
}
```

This generic trait specifies that its subtypes must provide a method for comparing against objects of some type `T`. If we now set this type `T` to be the type of our elements, we can obtain a more precise bound on `T`:

```
trait PriorityQueue[T <: GT[T]] {
  def push(v : T)
  def getTop() : T
}
```

In other words, the type `T` must promise us to be able to compare values of `T` against other values of type `T`. If we now want to implement a class of `UserRequest` objects that we want to be able to handle in a `PriorityQueue[MyRecord]`, we can define it as follows:

```
class UserRequest extends GT[UserRequest] {
  ...
  def isGreaterThan(v : UserRequest) = ...
}
```

This facility of bounding a type parameter by a type that contains the type parameter itself is called *F-bounded polymorphism* in the literature. It is supported e.g. by Java, Scala, and C#.

7.4 Variance on Type Parameters

As we have seen, generics are a valuable type system feature for the specification of abstract datatypes. However, the interaction of type variables and subtyping is not always trivial. Consider the following abstract datatype:

```
trait Box[T] {
  def put(v : T)
  def get() : T
}
```

This ADT simply allows us to store a value and retrieve it again. If we now use it in our code, we may run into situations where we may need to deal with subtypes and subtypes of *element* types. This raises the question: When is `Box[A]` <: `Box[B]`?

To help us understand this, we will write a small helper function that uses the `Box[T]`:

```
  def rebox(box : Box[B], b : B) {
    val v : B = box.get()
    box.put(b)
  }
```

Let us now consider the possibilities for passing a `Box[A]` to `rebox`:

- **A** :> **B**: For example, consider A = INT and B = [1 TO 10]. In this case, `box.put(b)` is safe, as `Box[A]` can store any number. However, `box.get()` might now return the number 99, which does not fit into the variable `b`. Thus, **this option is not statically type-safe**.
- **A** <: **B**: For example, consider A = [1 TO 10]. and B = INT In this case, `box.get()` works, but `box.put(b)` does not: `b` might be 99, which we cannot pass to an operation that only accepts `A` as parameter. Thus, **this option is not statically type-safe either**.

In other words, we cannot vary the type parameter `T` of `Box[T]`; we thus say that `Box[T]` *is* ***invariant*** *in* `T`.

As we saw, the operations that prevented us from varying `T` to supertypes and subtypes were different for our two cases. If we were to drop one of these operations, we might actually be more flexible. Let us thus consider the following types:

```
trait ReadBox[T] {
  def get() : T
}

trait WriteBox[T] {
  def put(v : T)
}
```

As we can see, we can safely replace a `ReadBox` with a `ReadBox` of a subtype; e.g.,

`ReadBox[[1 TO 10]]` <: `ReadBox[INT]`

since the only operation we have on a `ReadBox` is to read out an element. That means that when we vary the type parameter `T` towards a subtype, the type of `ReadBox[T]` also varies towards that of a subtype; we say that `ReadBox[T] is` **`covariant`** `in T`.

Conversely, we can replace `WriteBox`es by `WriteBox`es of supertypes; e.g.:

`WriteBox[INT]` <: `WriteBox[[1 TO 10]]`

since the only operation we have on a `ReadBox` is to write something. That means that when we vary the type parameter `T` towards a subtype, the type of `WriteBox[T]` varies in the opposite direction, towards that of a supertype; we say that `WriteBox[T] is` **`contravariant`** `in T`.

Many type constructors take type parameters, and for each of these type parameters we can wonder whether these are *invariant*, *covariant* or *contravariant*. A fourth option is for the type parameters to be *bivariant*, which means that they can be varied both to subtypes and supertypes; in practice, this means that they aren't used at all in the body of the type's definition. For example, recall our function types: We can think of the function type arrow (→) as a type constructor $\rightarrow [P, R]$ with two type parameters, parameter type P and return type R. As we have seen in the Arrow Rule (Section 5.2), the function type constructor is contravariant in P and covariant in R.

More formally:

Definition 4 *Let τ be a type constructor with formal type parameters $\tau_1, \ldots, \tau_k$, such that $T = \tau[\tau_1, \ldots, \tau_k]$ is a type.*

Let $i \in \{1, \ldots, k\}$.

If for all $\tau_i' <: \tau_i$ we can always substitute a value of type $\tau[\tau_1, \ldots, \tau_i', \ldots, \tau_k]$ in a context that expects a value of type $\tau[\tau_1, \ldots, \tau_i, \ldots, \tau_k]$ without violating type preservation then τ_i is covariant *in T.*

If for all $\tau_i' :> \tau_i$ we can always substitute a value of type $\tau[\tau_1, \ldots, \tau_i', \ldots, \tau_k]$ in a context that expects a value of type $\tau[\tau_1, \ldots, \tau_i, \ldots, \tau_k]$ without violating type preservation then τ_i is contravariant *in T.*

If τ_i is neither covariant nor contravariant in T, then τ_i is invariant *in T.*

7.5 Definition-Site and Use-Site Variance

Variance is a very explicit concept in Scala, Java, and C#. In Scala and C#, type parameters are always invariant unless they are declared to be covariant or contravariant. Scala and C# use the prefix operators + and - to declare co- and contravariance, respectively:

```
trait ReadBox[+T] { // covariant
  def get() : T
}

trait WriteBox[-T] { // contravariant
  def put(v : T)
}

trait Box[T] { // invariant
  def get() : T
  def put(v : T)
}
```

This *declaration-site variance* means that we decide about the type variable variance when we define our abstract datatypes and classes.

Java takes an alternative approach: here, we specify whether we want a type to be covariant or contravariant when we use it. Assuming that B $<:$ A :, we can write:

```
  class Box<T> {
    public T      get()      { ... };
    public void   put(T v)   { ... };
    public String toString() { ... };
  }

  Box<? extends A> covariantBox = new Box<B>();
  Box<? super B> contravariantBox = new Box<A>();
  Box<?> bivariantBox = new Box<AnyWeirdType>();
  Box<A> invariantBox = new Box<A>();
```

The literature calls this approach *use-site variance.*

Java will now prevent us from calling any method in `covariantBox` that has the type parameter in a contravariant position, so we cannot call `covariantBox.put()`. Analogously, for `contravariantBox`, we cannot call the `get()` operation. The `bivariantBox` prohibits calls to either `put` or `get`, but it won't stop us from calling `toString()`. Finally, the `invariantBox` permits all calls, at the cost of requiring an exact match of the type parameter T.

Use-site variance requires us to write more complex types, but allows us to re-use the same type definition (and the same implementations) for covariant, contravariant, bivariant, and invariant uses.

A Changes and Errata

A.1 Revision 2

- Fixes, minor clarifications, and improved phrasing.