

# An Introduction to Subtyping

Type systems are to me the most interesting aspect of modern programming languages. Subtyping is an important notion that is helpful for describing and reasoning about type systems. This document describes much of what you need to know about subtyping. I've taken some of the definitions, notation, and presentation ideas in this document from Kim Bruce's text "Fundamental Concepts of Object-Oriented Languages" which is a great book and the one that I use in my graduate programming languages course. I've also borrowed ideas from Cardelli and Wegner.

## 1.1. What is Subtyping

When is an assignment,  $x = y$  legal? There are at least two possible answers:

1. When  $x$  and  $y$  are of equal types
2. When  $y$ 's type can be "converted" to  $x$ 's type

The discussion of **name** and **structural** equality (which we went into earlier in this course) takes care of the first aspect.

What about the second aspect? When can a type be safely converted to another type? Remember that a type is just *a set of values*. For example, an INTEGER type is the set of values from minint to maxint, inclusive.

Once we consider types as set of values, we realize immediately one set may be a subset of another set.

[0 TO 10] is a subset of [0 TO 100]

[0 TO 100] is a subset of INTEGER

[0 TO 10] has a non-empty intersection with [5 TO 20] but is not a subset of [5 TO 20]

When the set of values in one type,  $T$ , is a subset of the set of values in another type,  $U$ , we say that  $T$  is a **subtype** of  $U$ . Or that  $U$  is a **supertype** of  $T$ . This is often written as  $T <: U$ . What are the implications of  $T$  being a subtype of  $U$ ?

"... a *value* of type  $T$  can be used in any context in which a *value* of type  $U$  is expected"  
[Bruce 2002, pg 24] (my italics)

In other words, if a program expects a value of type  $U$ , one can substitute a value of type  $T$  in its place and the program will still be correct *with respect to types*. For example, consider the following code fragment in MYSTERY:

```

VAR x: INTEGER;
BEGIN
  x := ?;
END;

```

Disregarding any kinds of conversions, MYSTERY expects the right-hand-side of the assignment to be the same type as the left-hand-side. In the example above, the expression substituted for "?" must evaluate to INTEGER. However, since [10 TO 20] is a subtype of INTEGER, one could substitute a value in the type [10 TO 20] and still expect the above program to be safe with respect to types.

```

VAR x: INTEGER;
BEGIN
  x := An expression that evaluates to [10 TO 20] (e.g., a call to p,
where p returns [10 TO 20])
END;

```

The above program will always respect the types since any value of type [10 TO 20] is also a legal INTEGER. Or to put it another way, any value in the set [10 TO 20] is a legal value in the set [minint TO maxint]. The program above automatically converts the value of type [10 TO 20] to INTEGER and then assigns it to x. A conversion from a subtype to a supertype is called a **widening conversion**. It is called a widening conversion because it goes from a smaller type (the subtype) to a bigger type (the supertype). Note that by "smaller" or "bigger" I do not mean the size of the representation in memory; I mean the size of the sets associated with the type. Since a widening conversion goes from a subtype to a supertype it is always safe (though it may be the case that with computer representations one may lose some precision). A conversion from a supertype to a subtype is called a **narrowing conversion**. It is called a narrowing conversion because it goes from a bigger type (supertype) to a smaller type (subtype). A narrowing conversion may or may not be safe.

For example:

```

VAR y: [0 TO 10];
BEGIN
  y := some expression evaluating to an integer
END;

```

The assignment to y needs a narrowing conversion. If one is lucky and the right-hand side of the assignment happens to evaluate to a value between 0 and 10, then the program will work successfully. If on the other hand if it evaluates to a value outside the range 0 to 10, then the above program will do a "bad" assignment. Since narrowing conversions may fail but widening conversions do not fail, many programming languages, such as Java, automatically apply widening conversions but require explicit casts to apply the narrowing conversions.

If a narrowing conversion fails then one of two things happen (depends on the language): (i) The language flags it as a run-time error and the program halts, perhaps throwing an exception; (ii) The supertype value is somehow converted to a subtype value, for example by throwing away some of the bits in the supertype value's representation. Java does a

combination of the two (uses the first option for reference types and the second option for primitive types). The first option has the advantage that it is safe. The second option has the advantage that it is fast (doesn't require as much checking). Do note, however, that since the second option throws away bits, it is not strictly correct as far as types are concerned.

## 1.2. **Subtyping for More Interesting Types**

So far we have used simple types such as subranges and integers to demonstrate subtyping. (BTW, MYSTERY has subrange types to allow me to demonstrate subtyping issues in the language). What about more interesting types?

Let's start with SET types (using Modula-3-like syntax):

```
TYPE Set1 = SET OF [3 TO 7];
TYPE Set2 = SET OF [1 TO 10];
```

Set1 includes all sets that have zero or more of the values in the range 3 to 7 (e.g., {3, 4, 7}). Set2 includes all sets that have zero or more of the values in the range 1 to 10 (e.g., {1, 4, 6, 7}). Thus, it must be the case that  $\text{Set1} <: \text{Set2}$ . More generally, one might say:

```
SET OF T <: SET OF U iff T <: U
```

It turns out that while the above reasoning makes perfect sense mathematically, languages that support sets (such as Modula-3) often do not support the above rule for pragmatic reasons. To see why, consider how a set is implemented. Set1 may be represented by 5 bits, with the first bit representing the value 3, the second representing the value 4, and so on. A value in Set1 will have zero or more of the bits set to 1 (e.g., 11001 would represent the set {3,4,7}). Set2 may be represented by 10 bits, with the first bit representing 1, the second representing 2, and so on. Now, if we allow the above subtyping rule, we will allow this assignment:

```
VAR s2: Set2;
BEGIN
  ...
  s2 := An expression evaluating to a value in Set1
END;
```

In order to implement the widening conversion required by this assignment, the compiler will have to generate code to convert a representation for Set1 into a representation for Set2. For example, the compiler will need to convert 11001 to 001100100. In the worst case the compiler will have to generate code to look at the value of type Set1 one bit at a time and based on its value of the bit set the corresponding bit in s2. While doable this will be slow and add complexity to the compiler. Thus, languages do not generally support subtyping of sets. (Historic note: The original definition of Modula-3 actually did support subtyping of sets but they decided to remove it from the language after a year for the above reasons).

Classes in object-oriented languages also generally support subtyping via subclassing. For example, in C++ syntax:

```
class C1 { int x; public void m() {...} };
class C2: public C1 {int y; public void n() {...}};
```

C2 is not just a subclass of C1 but is also a subtype. To see why, let's consider a set-based argument. C1 is a type whose set of values contain all objects that have an instance variable  $x$  and a method  $m$ . C2 is a type whose set of values contain all objects with instance variables  $x$  and  $y$ , and methods  $m$  and  $n$ . Considered this way, since all values in C2 also support  $x$  and  $m$ , it must be the case that the values in C2 must be a subset of the values in C1. Thus,  $C2 <: C1$ . Since we will see a lot more of this as the class progresses, I won't go into any further detail of subtyping of classes.

For languages that treat functions as first-class values, one may want to talk about subtyping of function types. For example:

```
TYPE P1 = PROCEDURE (x: P1A): P1R;
TYPE P2 = PROCEDURE (x: P2A): P2R;
```

Where P1A and P1R are defined elsewhere and are the argument and return types for P1. Similarly P2A and P2R are the argument and return types for P2. For P1 to be a subtype of P2, it must be legal (with respect to types) to call P1 instead of P2. There are two parts to this:

1. Since P1 is being called instead of P2, it must be the case that any argument for P2 must be a legal argument for P1. i.e.,  $P2A <: P1A$
2. Since P1 is being called instead of P2, it must be the case that any return value for P1 must be a legal return value for P2. i.e.,  $P1R <: P2R$ .

The above two requirements define the correct subtyping rule for procedures. The above rule is also sometimes called the "arrow rule" (since in formal notation, function types are written using  $\rightarrow$ , with the argument type on the left-hand-side of the arrow and the return type on the right-hand side of the arrow).

I do not know of any language that uses the full rule above. However, languages, such as C++, often use half of the rule. In particular they use part (2) of the rule. The reason why languages do not use (1) is that it doesn't seem to be very useful while (2) is very useful. We will see an example of this when we study object-oriented languages later in the course.

### **1.3. Subtyping of Updatable Entities**

So far in our discussion of subtyping we have ignored things that can be updated (such as memory locations or arrays). It turns out that subtyping of updatable entities is much more restrictive than subtyping of non-updatable entities.

Let's consider the following scenario. Given that  $[1 \text{ TO } 10] <: [1 \text{ TO } 100]$  we can substitute any value in the set of  $[1 \text{ TO } 10]$  where a value in the set of  $[1 \text{ TO } 100]$  is expected. Now the question is: can I substitute a *variable* of type  $[1 \text{ TO } 10]$  where a variable of type  $[1 \text{ TO } 100]$  is expected?

```
VAR x: [1 TO 10];
VAR y: [1 TO 100];
VAR z: [1 TO 100];
BEGIN
  y := z;
  z := y;
END;
```

The above assignment is legal because  $y$  and  $z$  are of the same type (assuming structural equality). Since the type of  $x$  is a subtype of the type of  $z$ , can I substitute the variable  $x$  where the variable  $z$  is used?

```
VAR x: [1 TO 10];
VAR y: [1 TO 100];
VAR z: [1 TO 100];
BEGIN
  y := x;
  x := y;
END;
```

It is obvious that the first assignment ( $y := x$ ) will always succeed. However, the second assignment ( $x := y$ ) may or may not succeed if, for example,  $y$  holds the value 20. Bottom line: if  $S <: T$ , it does not mean that  $\text{REF } S <: \text{REF } T$  (I use "REF" to mean a location, i.e.,  $\text{REF } T$  is an updatable location that can hold values of type  $T$ ).

Let's consider the above situation more abstractly and derive a rule:

```
VAR x: T;
VAR y: S;
```

Imagine that we are trying to determine if one can use  $y$  in a place where the program expects  $x$ . Since  $x$  may be read to obtain a value, it must be the case that  $y$ 's value must be a subtype of  $x$ 's value:

$S <: T$

In addition, since  $x$  may be modified, it must be the case that any value that  $x$  can be modified with must be a legal value for  $y$ . Since  $x$  may be modified with any value in  $T$ , it must be the case that

$T <: S$

The only reasonable solution to these two constraints is that  $T = U$ ; i.e., a variable may be used in place of another variable only if they have the same type.

Let's now apply what we have learned so far to arrays. We would like to be able to say:

`ARRAY I OF S <: ARRAY I OF T, if S <: T`

In other words, one array is a subtype of another array if they have the same index type (e.g., [10 TO 20]) and the element type of the first array is a subtype of the element type of the second array. Is this actually true? If we think about it for a moment, we realize that like variables, arrays are updatable in most languages. In other words, one can not only read the values in an array but one can modify values in the array.

In other words, let's suppose I have a context that expects an `ARRAY I OF T`. This context may read from the array to get a value of type `T` and may write values of type `T` to the array. Now let's suppose I want to substitute an `ARRAY I OF S` in this context. When I read from the array, I'll expect to get back a value of type `T` since the context expects an `ARRAY I OF T`. Thus, I have the requirement:

`S <: T`

When I write into the array, I'll expect to be able to write a value of type `T` since the context expects an `ARRAY I OF T`. Thus, I have the requirement:

`T <: S`

In other words, the above array subtyping rule is incorrect. For one updatable array type to be a subtype of another array type, it must be the case that they have the same element type. Java actually uses the "wrong" rule for subtyping arrays:

The following conversions are called the *narrowing reference conversions*:

...

From any array type `SC[]` to any array type `TC[]`, provided that `SC` and `TC` are reference types and there is a narrowing conversion from `SC` to `TC`. [Section 5.1.5, Java Language Specification, 2nd Ed]

The price that Java has to pay for this flexibility is lots of extra run time checks. Every assignment to an element of an array of references in Java needs to be checked to make sure that the value being stored is legal for the array.

## **1.4. So what good is subtyping?**

We now discuss the implications of subtyping for type checking and for *polymorphism*.

### **1.1.1. Subtyping and type checking**

Recall that when we considered the legality of the following code we only considered type equality:

```

VAR l: Tl;
VAR r: Tr;
...
l := r;

```

In other words, we said that the assignment was legal only if **Tl** and **Tr** were equivalent types. Real languages consider this to be overly restrictive and allow some kinds of conversions. For example, many languages (e.g., C and C++) allow an expression of type **short** to be assigned to an expression of type **int**. More specifically, these languages automatically convert the right-hand side to the type of the left-hand side in some cases. From a type safety point of view, *an assignment is always legal if the type of the right-hand side can be widened to the type of the left-hand side of the assignment.*

Sometimes, however, programmers need the flexibility to do things that may not always be legal with respect to types. For example, consider the code fragment:

```

VAR l: SHORT;
l := 10;

```

(MYSTERY does not have a **SHORT** type; I made it up for this example). Is this assignment legal if we allow only widening conversions? It depends on the type of '10'. The most intuitive type of '10' is **INTEGER** but if this is the case, then the assignment is not legal because it needs a narrowing conversion (from **INTEGER** to **SHORT**), not a widening conversion.

This is obviously a very serious issue: if we don't have a solution for this then it renders shorts useless since we can not initialize them! Programming languages can deal with this situation in one of three ways:

1. Define the type of '10' to be a subrange type [**10 TO 10**]. Since [**10 TO 10**] is a subtype of **SHORT** it enables us to initialize shorts. However this is not a general solution. There are many other situations when one may want to assign a supertype to a subtype. Thus, the next two solutions are more general.
2. Allow not just widening conversions but narrowing conversions. In other words, the language automatically does a narrowing conversion when the type of the right-hand side is a supertype of the type of the left-hand side of the assignment. Modula-3 takes this approach.
3. The problem with the previous approach is that it automatically does unsafe things because narrowing conversions are potentially unsafe. Worse yet, the assignments that use the potentially unsafe narrowing conversions are not explicitly marked: a programmer reading the code would have to examine the types involved in an assignment to determine whether or not the assignment is potentially unsafe. Thus, some languages do automatic widening conversions but require the programmer to explicitly request a narrowing conversion if it is needed. In other words, the programmer would have to write

```

VAR l: SHORT;
l := (SHORT) 10;

```

(Using C style syntax for conversions). As is often the case with programming languages, things are never so cut and dried: many languages use a combination of the last two options. For example, Java does automatic narrowing conversions for primitive types (e.g., `int`) in some cases and requires explicit narrowing conversions for non-primitive types (e.g., arrays).

So far we have talked primarily about type checking assignments but the issues in type checking other operations are similar. For example, parameter passing using pass-by-value is similar to assignments where the actual arguments are assigned to the formal arguments. Thus, pass-by-value parameters use similar type checking to assignments.

### 1.1.2. Subtyping and polymorphism

Besides allowing language designers to cleanly express the conditions for type safety, subtyping also provides an incredibly powerful feature: *polymorphism*. Polymorphism means “having many forms”. For example, a value of type `[10 TO 20]` has the type `[10 TO 20]` but can also be considered to have the type `[10 TO 30]` and `[5 TO 20]` (and really any supertype of `[10 TO 20]`). This is just a reformulation of our earlier statement that “... a *value* of type T can be used in any context in which a *value* of type U is expected” [Bruce 2002].

Consider the following function in MYSTERY syntax:

```
PROCEDURE gt(i: INTEGER; j: INTEGER): BOOLEAN =
BEGIN RETURN i>j; END;
```

This function is polymorphic because it can take an argument of many different types: not only `INTEGER` but any subtype of `INTEGER`. Note how polymorphism gives us *code reuse*: we can use this function to compare values of any type as long as that type is a subtype of `INTEGER`. If we did not have subtyping then we may need many different functions, for example:

```
PROCEDURE gt(i: INTEGER; j: INTEGER): BOOLEAN =
BEGIN RETURN i>j; END;
PROCEDURE gt(i: SHORT; j: SHORT): BOOLEAN =
BEGIN RETURN i>j; END;
PROCEDURE gt(i: LONG; j: LONG): BOOLEAN =
BEGIN RETURN i>j; END;
```

(MYSTERY does not really have a `LONG` or a `SHORT` type: I made it up just for this example).

The polymorphism described above is often called *inclusion* polymorphism because it comes about due to inclusion of values (recall that a supertype’s value set includes all the values in the subtype’s value set). There are other kinds of polymorphism. Later in the course we will explore *parametric* polymorphism.

To summarize, polymorphism enables us to write something once and use it for many different types. This polymorphism is the foundation of object-oriented languages. We will get back to it later in the course!