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. Some of the definitions, notation, and presentation ideas in this document have been taken from Kim Bruce's text "Fundamental Concepts of Object-Oriented Languages" which is a great book and the one that I use in my CSCI 5535 course. Some of the presentational ideas have also been taken from Cardelli and Wegner.

The Basic Concept of 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 the semester) 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 *as 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.

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 `Set1 <: 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 →, 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 semester.

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 TO 10] <: [1 TO 100] we can substitute any value in the set of [1 TO 10] where a value in the set of [1 TO 100] is expected. Now the question is: can I substitute a *variable* of type [1 TO 10] where a variable of type [1 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 REF S <: REF T (I use "REF" to mean a location, i.e., 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 look at a concrete example that illustrates the issues here:

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 array element in Java needs to be checked to make sure that the value being stored is legal for the array.