Handout E: Type Equality

courtesy of Amer Diwan (University of Colorado at Boulder, now Google)

November 12, 2021

Type equality

When are two types equal? Based on the notion of types we have discussed above, one answer is:

> *Two types, T1 and T2, are equal if the set of values in T1 is the same as the set of values in T2*

While the above definition makes good sense, it is not constructive: in other words, since the set of values in many types is unbounded (e.g., **`String`**) one cannot simply enumerate the sets of values and compare them. We need a definition that can be applied (as an algorithm) to determine when two types are equal.

Modula-3's definition of when two types are equal gives us a hint for what an algorithm should look like:

"Two types are the same if their definitions become the same when expanded; that is, when all constant expressions are replaced by their values and all type names are replaced by their definitions. In the case of recursive types, the expansion is the infinite limit of the partial expansions" [Systems Programming in Modula-3, Nelson 1991]

Let's see what this definition means.

Types **`INTEGER`** and **`INTEGER`** are equal because they are the same when expanded (there is nothing to expand here since **`INTEGER`** is a primitive type).

Types **`BOOLEAN`** and **`BOOLEAN`** are equal because they are the same when expanded (once again there is nothing to expand).

Types **`BOOLEAN`** and **`INTEGER`** are not equal because they are not the same when expanded.

Now consider these more interesting types:

```
TYPE A = ARRAY [1 TO 10] OF INTEGER;
TYPE I = INTEGER;
TYPE B = ARRAY [1 TO 10] OF INTEGER;
TYPE C = ARRAY [1 TO 10] OF I;
```

(The **`TYPE`** declarations declare a new name for a type, much like what **`typedef`** accomplishes in C and C++).

Types **`A`** and **`B`** are equal because their expanded definitions are the same (both are **`ARRAY [1 TO 10] OF INTEGER`**). Types **`A`** and **`C`** are also equal because their expanded definitions are the same. Note that when expanding **`C`**'s definition, we have to expand **`I`** with **`INTEGER`**.

How about these types:

```
TYPE T1 = RECORD val: INTEGER; next: REF T1; END;
TYPE T2 = RECORD val: INTEGER; next: REF T2; END;
```

(**REF T1** means pointer to **T1**, which in C/C++ notation you would write as **T1***). These two types are infinite so you cannot expand out their definitions and compare. The key insight when comparing such types is that one can assume that two types are equal unless one finds a counter example. The way to implement this as follows: whenever the type equality mechanism encounters the same pair of types that it has encountered in the past, it assumes that they are the same. For example, the compiler will go through these steps in order to answer **is-type-equals(T1,T2):**

1. If both **T1** and **T2** are both records, examine their fields. Since both have the same number of fields, the types might be equal.

2. Compare the first field of **T1** to the first field of **T2**. Since both have the same name and same type, we have not yet found a reason why **T1** and **T2** must be not equal.

3. Since the second field of **T1** and the second field of **T2** have the same name, compare their type (i.e., **is-type-equals(REF T1, REF T2)**).

4. Since both types are **REF** types (i.e., pointer types) compare their referent types (i.e., **is-type-equal(T1,T2)**). Recall that we are assuming at this point that **T1** and **T2** are equal so there is nothing to do here.

Since there is nothing else left to check and we have not found any reason to believe that **T1** and **T2** are different, they must be equal.

Here is a pair of types that are not equal:

```
TYPE T11 = RECORD val: INTEGER; next: REF T11; c: CHAR; END;
TYPE T21 = RECORD val: INTEGER; next: REF T21; c: INTEGER; END;
```

We will go through the same steps as the previous example. However, after the last step in the previous example, we will compare the third fields of the two types and find them to be not equal (**CHAR** is not equal to **INTEGER**). Thus, the we have found a reason why **T11** and **T21** should not be considered equal.

Modula-3's type equality mechanism is called *structural type equality* since it looks at the structure of types to figure out if two types are the same. While structural equality makes a lot of sense, some language designers do not like it. For example, consider the two records:

```
TYPE Student = RECORD id: INTEGER; name: String; END;
TYPE Course = RECORD id: INTEGER; name: String; END;
```

The programmer has declared two types, **`Student`** and **`Course`**, and they happen to have the same fields with the same types. As far as structural equality is concerned, these two types are equal. Consider the following code:

```
PROCEDURE registerForCourse(s: Student; c: Course) = …
VAR aStudent: Student;
VAR aCourse: Course;
…
registerForCourse(aCourse, aStudent)
```

The compiler will not flag the error (i.e., the programmer has mistakenly swapped the arguments to **`registerForCourse`**). In other words, structural type equality may be too liberal in some cases.

Thus, many languages use an alternative to structural type equality: *name type equality*. To understand name type equality, we need this definition:

> **Definition**: A *type constructor* is a mechanism for creating a new type.

For example, **`ARRAY`** is a type constructor since when we apply it to an index type and an element type it creates a new type: an array type. **`class`** in C++ is another example of a type constructor: it takes the name of the superclass, the names and types of the fields, and creates a new class type.

The idea behind name type equality is as follows: every time a program uses a *type constructor* the language automatically generates a unique *name* for the type. Another way to think of it is: every time a program creates a new type, the language automatically gives it a unique name. Two types are equal if they have the same *name*. Note that this *name* has nothing to do with the programmer-given name of the type (e.g., the **`Student`** and **`Course`** type names in the example above). Thus the term *name type equality* is often confusing to students of programming languages.

Let's now see some examples of name type equality:

```
TYPE Student = RECORD id: INTEGER; name: String; END;
TYPE Course = RECORD id: INTEGER; name: String; END;
```

These two types are different because each use of "**`RECORD`**" creates a new type name: the **`Student`** record and **`Course`** record thus have different names. However, if Student and Course were defined as follows, they would be equal types:

```
TYPE T = RECORD id: INTEGER; name: String; END;
TYPE Student = T;
TYPE Course = T;
```

How about **`INTEGER`** and **`INTEGER`**? **`INTEGER`** is not a type constructor; thus, all uses of **`INTEGER`** have the same name. Thus all uses of the **`INTEGER`** type refer to the same type.

While structural type equality can be too liberal, name type equality can be too restrictive. For example, consider the following code fragment:

```
VAR a: ARRAY [1 TO 10] OF INTEGER;
PROCEDURE p(f: ARRAY[1 TO 10] OF INTEGER) = …
BEGIN
  p(a);
END;
```

The above call to **p** will not succeed with name type equality since the two uses of **ARRAY** (for the declarations of **a** and **f** respectively) yield different types. In a language that uses name type equality, one would have to rewrite the above as:

```
TYPE T = ARRAY [1 TO 10] OF INTEGER;
VAR a: T;
PROCEDURE p(f: T) = …
BEGIN
  p(a);
END;
```

The above succeeds since there is just one use of the type constructor **ARRAY**.

In reality, most languages use a combination of name and structural equality. For example, Modula-3 uses structural equality for all types except when a programmer requests otherwise (using a special keyword **BRANDED**). C, C++, and Java use name type equality for all types other than arrays, which use structural type equality. Here is what Java's language definition says about type equality for its reference types (i.e., arrays, classes, and interfaces):

> *Two reference types are the same run-time type if:*
>
> *They are both class or both interface types, are loaded by the same class loader, and have the same binary name (§13.1), in which case they are sometimes said to be the same run-time class or the same run-time interface.*
>
> *They are both array types, and their component types are the same run-time type(§10).* [The Java Language Specification, 2nd Edition]

In other words, class and interface types must have the same name (a "binary name" is a fully qualified name such that each class/interface in a program has a unique name). Arrays, on the other hand, use structural equality because one looks at the element type of the array to determine equality.

Type equality and distributed computing

The issue of what type equality a language uses can have an impact on the kinds of programs that one can write in a language. In this section, I'll describe an example situation that is problematic for name equality but not for structural equality. While this example situation is described in terms of distributed computing, it represents a more