

Type Checking and Type Equality

Type systems are the biggest point of variation across programming languages. Even languages that look similar are often greatly different when it comes to their type systems.

Definition: A *type system* is a set of types and *type constructors* (arrays, classes, etc.) along with the rules that govern whether or not a program is legal with respect to types (i.e., type checking).

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.

C++ and Java have similar syntax and control structures. They even have a similar set of types (classes, arrays, etc.). However, they differ greatly with respect to the rules that determine whether or not a program is legal with respect to types. As an extreme example, one can do this in C++ but not in Java:

```
int x = (int) "Hello";
```

In other words, Java's type rules do not allow the above statement but C++'s type rules do allow it.

Why do different languages use different type systems? The reason for this is that there is no one *perfect* type system. Each type system has its strengths and weaknesses. Thus, different languages use different type systems because they have different priorities. A language designed for writing operating systems is not appropriate for programming the web; thus they will use different type systems. When designing a type system for a language, the language designer needs to balance the tradeoffs between execution efficiency, expressiveness, safety, simplicity, etc.

Thus, a good understanding of type systems is crucial for understanding how to best exploit programming languages. I've seen many programmers who do not understand the type system of the language they are using and thus end up writing code that fights with the type system rather than works with the type system. This chapter introduces some of the basic issues with types and type checking. Later in the course we will delve even deeper into these issues and some of the points of variations across languages.

1.1. *The need for a type system*

Some older languages and all machine assembly languages do not have a notion of type. A program simply operates on memory, which consists of bytes or words. An assembly language may distinguish between "integers" and "floats", but that is primarily so that the

machine knows whether it should use the floating point unit or the integer unit to perform a calculation. In this type-less world, there is no type checking. There is nothing to prevent a program from reading a random word in memory and treating it as an address or a string or an integer or a float or an instruction.

The ability to grab a random word from memory and interpret it as if it were a specific kind of value (e.g., integer) is not useful most of the time even though it may be useful sometimes (e.g., when writing low-level code such as device drivers). Thus even assembly language programmers use conventions and practices to organize their program's memory. For example, a particular block of memory may (by convention) be treated as an array of integers and another block a sequence of instructions. In other words, even in a type-less world, programmers try to impose some type structure upon their data. However, in a type-less world, there is no automatic mechanism for enforcing the types. In other words, a programmer could mistakenly write and run code that makes no sense with respect to types (e.g., adding 1 to an instruction). This mistake may manifest itself as a crash when the program is run or may silently corrupt values and thus the program produces an incorrect answer. To avoid these kinds of problems, most languages today have a rich type system. Such a type system has types and rules that try to prevent such problems. Some researchers have also proposed typed assembly languages recently; these would impose some kinds of type checking on assembly language programs!

The rules in a given type system allows some programs and disallows other programs. For example, unlike assembly language, one cannot write a program in Java that grabs a random word from memory and treats it as if it were an integer. Thus, every type system represents a *tradeoff between expressiveness and safety*. A type system whose rules reject all programs is completely safe but inexpressive: the compiler will immediately reject all programs and thus also all the "unsafe" programs. A type system whose rules accept all programs (e.g., assembly language) is very expressive but unsafe (someone may write a program that writes random words all over memory).

1.2. ***What exactly are types and type checking?***

We have been talking about types and type checking but have not defined it formally. We will now use a notion of types from theory that is very helpful in understanding how types work.

Definition: A *type* is a set of values.

A **boolean** type is the set that contains **True** and **False**. An **integer** type is the set that contains all integers from **minint** to **maxint** (where **minint** and **maxint** may be defined by the type system itself, as in Java, or by the underlying hardware, as in C). A **float** type contains all floating point values. An **integer** array of length 2 is the set that contains all length-2 sequences of integers. A **string** type is the set that contains all strings. For some types the set may be enumerable (i.e., we can write down all the members of the set, such as **True** and **False**). For other types type set may not be enumerable or at least be very hard to enumerate (e.g., **String**).

When we declare a variable to be of a particular type, we are effectively restricting the set of values that the variable can hold. For example, if we declare a variable to be of type **Boolean**, (i) we can only put **True** or **False** into the variable; and (ii) when we read the contents of the variable it must be either **True** or **False**.

Now that we have defined what a type is, we can define what it means to do type checking.

Definition: *Type checking* checks and enforces the rules of the type system to prevent type errors from happening.

Definition: A *type error* happens when an expression produces a value outside the set of values it is supposed to have.

For example, consider the following assignment in MYSTERY syntax:

```
VAR s:String;  
s := 1234;
```

Since **s** has type **String**, it can only hold values of type **String**. Thus, the expression on the right-hand-side of the assignment must be of type **String**. However, the expression evaluates to the value **1234** which does not belong to the set of values in **String**. Thus the above is a type error.

Consider another example:

```
VAR b:Byte;  
b := 12345;
```

Since **b** has type **Byte**, it can only hold values from -128 to 127. Since the expression on the right-hand-side evaluates to a value that does not belong to the set of values in **Byte**, this is also a type error.

Definition: *Strong type checking* prevents all type errors from happening. The checking may happen at compile time or at run time or partly at compile time and partly at run time.

Definition: A *strongly-typed language* is one that uses strong type checking.

Definition: *Weak type checking* does not prevent type errors from happening.

Definition: A *weakly-typed language* is one that uses weak type checking.

If you use a weakly-typed language, it is up to your discipline to make sure that you do not have type errors in your program. Many languages use something in the middle: they catch many type errors but let some through.

1.3. ***Dimensions of type checking***

As we discussed above, the type checking may be:

- Strong, weak, or something in between
- At compile time, run time, or something in between

We now discuss the tradeoffs involved in the above two dimensions.

As discussed above, strong type checking catches all type errors. Thus, it provides the greatest amount of safety: programmers that write code in a strongly-typed language do not have to worry about many kinds of bugs. For example, when a program in a strongly-typed language dereferences an integer pointer (`int *`), we know for sure that if the dereference completes successfully it will produce an integer. On the other hand, strong type checking can be overly restrictive in some cases. Let's suppose you need to write a device driver and that the device's registers are mapped to some fixed known location in memory. When writing this driver you will need to grab data at the fixed locations in memory and treat them as if they were integers or whatever (depending on what the device specification says). A strongly typed language will not allow you to do this. Thus, for some applications, strong typing may be just right whereas for others, strong typing may be too restrictive. To get most of the benefits of strong typing while avoiding most of its disadvantages, some languages use a combination of the two.

If a language performs any type checking, it may perform it at compile time or at run time or partly at compile time and partly at run time. Consider the following code fragment:

```
VAR b: Byte;
VAR x: INTEGER;
b := x;
```

The assignment to `b` may or may not be a type error. If `x` happens to have a value between -128 and 127 (which is the set of values in the set for byte), the assignment won't cause a type error; if `x` has a value outside the range -128 to 127 then the assignment will cause a type error. If the language is strongly typed and does type checking only at compile time, then it will reject the above program (because it *may* cause a type error when the program runs). If the language is strongly typed and does type checking at run time, then it will all allow the program to run but it will check at run time to make sure that a type error does not happen. In other words, if the language does type checking at run time, the compiler may transform the assignment above into:

```
IF (x < -128 OR x > 127) abort();
ELSE b := x;
```

Effectively, the compiler wraps a run-time check around the assignment to make sure that it does not cause a type error.

The above example also demonstrates the relative strengths and weaknesses of compile-time and run-time type checking:

- Compile-time type checking finds bugs sooner than run-time type checking.

- Run-time type checking is more expressive since it rejects fewer programs than compile-time type checking.
- Run-time type checking may slow down the program since it introduces checks that must be performed at run time.

Since there is no one clear winner between compile-time checking and run-time checking most modern languages do a bit of both. Java, for example, checks for many errors at compile times and for others at run time. We will see many examples of this when we discuss *subtyping* later in the course.

1.4. **What to check during type checking?**

Assume that we have the following assignment and need to check whether or not the assignment is type correct (i.e., does not have a type error):

```
VAR l: T1;
VAR r: Tr;
...
l := r;
```

When type checking happens at run time, we just need to check that the actual value of **r** is in the set of values that **l** can hold. When type checking happens exclusively at compile time, there are two possible rules that one could use:

1. **T1 = Tr**. In other words, the types of the left and right hand side must be *equal*
2. Set for **Tr** is a subset of the set for **T1**. If all the values in the set for **Tr** belong to the set for **T1**, then the assignment will always succeed. For example, consider the code fragment:

```
VAR b: Byte;
VAR x: INTEGER;
x := b;
```

Since the set of values in **Byte** is a subset of the set of values in **INTEGER**, the above assignment will never result in a type error.

We will discuss option (1) in the rest of this chapter. In other words, we will assume the type checking rule: *the left and right hand sides of an assignment must be of equal types and the actual parameter passed to a procedure must have the same type as the corresponding formal parameter*. Later in the course we will discuss option (2). Option (2) actually forms the foundation of object-oriented languages!

1.5. **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, assume that the types are equal. The remaining steps try to find a contradiction to this.
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, 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 code first declares a procedure (**registerForCourse**) that takes a student and a course and then calls the procedure. Unfortunately, the code passes the arguments in the wrong order. The compiler will not flag the error since the types still match up according to structural equality. In other words, structural type equality may be too liberal in some cases. It is worth noting here that the above is a contrived example; in practice it is unlikely that two unrelated types will end up with exactly the same structure.

Nonetheless, many languages use an alternative to structural type equality: *name type equality*.

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;
```

(**TYPE** is not considered to be a type constructor here since it does not create a new type; it just gives a new name for an existing type).

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.

1.6. Implications of type equality mechanisms

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 general phenomenon that occurs whenever two programs communicate or if a program needs to read or write typed data to a file (e.g., a database application).

Consider two programs, P1 and P2, which communicate with each other. P2 exports the following routine that P1 can call:

```
PROCEDURE calledByP1(x: INTEGER) = ...
```

In a type-safe environment, we would like the communication to avoid type errors. In this case, we need to make sure that P1 passes an integer to `calledByP1`. This situation

is different from a normal procedure call in that the caller and callee are in separate programs.

As long as **P1** and **P2** communicate only with primitive types, it is not a problem. Now let's suppose we want a richer form of communication. In that case **P2** might contain:

```
TYPE R = RECORD x: INTEGER; y: INTEGER; END;  
PROCEDURE calledByP1b(a: R) = ...
```

P1 can try to call this as follows:

```
TYPE P1R = RECORD x: INTEGER; y: INTEGER; END;  
VAR r: P1R;  
BEGIN  
  (* Assume that r is initialized with some data *)  
  calledByP1b(r);  
END;
```

Does this call work? Well, it depends on whether the language uses structural type equality or name type equality. If the language uses structural type equality, **P1R** and **R** are equal and thus **P1** can call **calledByP1b** with **r**. If the language uses name equality, **P1R** and **R** are not equal types: they are created using two distinct applications of a type constructor. With name equality there is no clean way of fixing this problem: since **P1** and **P2** are different programs, they cannot share the same **RECORD**. In other words, with name equality, there is no clean way of making the above solution work.

A language that has name type equality can try to fix the above in one of two ways: it can relax name type equality for the types that are likely to be used in inter-program communication (perhaps this related to the reason why C and C++ use structural equality for array types). Or it can incorporate some other mechanisms involving run-time checking; this is what Java does.

1.7. Summary of Chapter

Type systems are a large and fundamental part of modern programming languages. A type system has two parts: a set of types and a set of rules for checking consistent uses of the types. In this chapter we focused on a simple set of rules that use type equality only. Later on we will look at rules that look at other forms of type compatibility. We identified two kinds of type equality mechanisms: name type equality and structural type equality. Since there are strengths and weaknesses of both mechanisms, there is no one clear winner: different languages use different mechanisms and even within a language some types may use one mechanism and other types the other mechanism. Understanding these issues will give you a clear sense of how to work with the type system of a language rather than against the type system of the language (which is never productive!).