Handout B: Type Systems

Christoph Reichenbach

November 21, 2019

Type systems are a central part of modern programming languages: they describe constraints over the possible values that an expression may evaluate to, and sometimes even more advanced properties, such as whether the evaluation may abort with exceptional behaviour, again expressed as constraints. These constraints allow programming language implementers to prevent costly bugs, or at least to make these bugs easier to catch and understand. Thus, type systems can contribute greatly to the *Robustness*, but also the *Readability* of programming languages. In addition to these improvements in error reporting, certain type systems can allow language implementers to decrease the runtime cost (and sometimes the compile-time cost) of the language.

However, there is a considerable degree of variation between the way in which different languages handle types, even among languages that we today consider modern, like Scala, Python, or Rust. Something that may be entirely legal in one programming language may violate the rules of the type system in another language.

For example, the C type system allows the following assignment, whereas the exact same assignment is illegal in Java[1]:

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

More insidiously, the following code is valid both in Scala and JavaScript:

```
var result = "foo" * 2;
```

However, in Scala the code produces a character string (`"foofoo"`), whereas in JavaScript it produces a number.

In the following, we will look into what types and type systems are, and into the main concepts underlying them.

1 Types

Very abstractly speaking, type systems are systems that relate the three concepts of *types*, *expressions*, and *values* to each other and describe rules for ensuring some form of correctness. Here, expressions e and values v are constructions that we have already encountered when discussing operational semantics:

$$e \Downarrow v$$

That is, expressions are language constructs that we can evaluate, and values are the results of these evaluations. Our earlier discussion has not delved into types, however.

Definition 1 *A* type *is a subset of the set of values.*

If τ is a type and v is a value, then we write

$$v : \tau$$

('v has the type τ') to say that $v \in \tau$.

[1] Example due to Amer Diwan

For example, the int type of Python and the BigInt type of Java represent the set of integers (also written as $\mathbb{Z}$), i.e., any whole positive or negative number, so we can formally write:

$$v : \mathsf{int} \iff v \in \mathbb{Z}$$

or, equivalently, $\mathsf{int} = \mathbb{Z}$.

Another example are *subrange types*, as found in Ada, Pascal, Modula-2, and the Mystery language:

```
VAR x : [0 TO 10];
x := 0;
```

A subrange type (written $[b_\ell$ **TO** $b_u]$ in Mystery) is the type of all integers between the lower bound b_ℓ and the upper bound b_u, including the bounds themselves. In the above example, variable x can assume the number 0, or any integer up to and including 10. However, the following code attempts to violate the constraints that the subrange type [0 **TO** 10] imposes over x:

```
VAR x : [0 TO 10];
x := 11;
```

This code contains a *type error*: $11 \notin [0$ **TO** $10]$.

Definition 2 *A* type error *is an attempt to perform an operation that requires an input value of type τ with a value v even though $v : \tau$ does not hold.*

Attempting to take a variable bound to type τ and binding it to such a value v even though $v : \tau$ does not hold is an example of such an operation.

Of course, defining our types and knowing what type errors are only helps us if we also check for them:

Definition 3 Type checking *is the process of detecting type errors.*

2 Strong and Weak Typing

Not all programming languages have a notion of type errors or type checking. Modern (and most of the not-so-modern) CPUs only know about sequences of bytes, and their assembly languages reflect this view of the world.

Recall from the textbook that integers and floating point numbers have entirely different in-memory representations; the bit string that represents the 32-bit integer number 1 is completely unlike the bit string that represents the 32-bit IEEE 754 floating point number 1.0. However, if we instruct the CPU to load the floating point number 1.0 and then perform an integer addition on that same number, the CPU will cheerfully obey — producing meaningless garbage in the process.

Even languages with type systems, such as C and C++, do not reliably prevent such mis-uses of the bit strings that we store in memory. On the other hand, languages like Scala and Ocaml do:

Definition 4 Strong type checking *is a property of a programming language that states that any language implementation finds all type errors and prevents the operations that caused the type error from taking place.*

Definition 5 Weak type checking *is the absence of strong type checking.*

We say that a language with weak type checking is a *weakly-typed language*, and a language with strong type checking is a *strongly-typed language*. Examples of weakly-typed languages are C, C++, and JavaScript, while strongly-typed languages include Haskell, Python, Ruby, and Java.

Strong type checking improves the reliability of a language, at a cost to the language's expressivity, and sometimes also at a cost to the language's execution time.

For most programs the gain of increased reliability is greater than the loss in expressivity, but there are exceptions: in operating system kernel development (e.g., when working on device drivers) or language runtime environment development, developers may have to write directly to memory without regard to the type system. Some languages therefore provide backdoors that allow bypassing the type checker, like Go's unsafe package, Rust's unsafe blocks, Modula-3's LOOPHOLE operator, or 'native calls' to C or Assembly supported by most major languages.

If these backdoors are not necessary for normal use of the language, we still consider the language to be strongly typed, even though strictly speaking only the language *minus the backdoor* is strongly typed.

3 When to Type-Check

Python and Ruby are strongly typed, but they check types after the program has started, i.e., at runtime. That means that for these languages, type-checking is part of the *dynamic semantics*.

Other languages (Rust, Java, Scala) check types at compile time, i.e., before the program is run. In these languages, type checking is part of the *static semantics*.

Definition 6 *Any type checking that is performed at runtime is called* dynamic type checking.

Definition 7 *Any type checking that is performed before runtime (at compile time, as part of the static semantics) is called* static type checking.

Consider the following program, which is syntactically valid in both Scala and JavaScript:

```
var x = 0;
print("Hello, World!");
print(x[7]);   // Type Error
```

In Scala, this will not compile, due to the type error, so we cannot even execute the part of the code that is correctly typed. This demonstrates Scala's static type checking.

Note that in JavaScript, the exact same program will print[2] "Hello, World!" and then "undefined", which signals a type error that was suppressed by the JavaScript runtime. This demonstrates that JavaScript is weakly typed; a strongly typed language would have prevented us from printing the result of the invalid `x[7]`.

The corresponding Python program looke like this:

```
x = 0
print("Hello, World!")
print(x[7])   # Type Error
```

This program will print "Hello, World!", and then halt with a type error, demonstrating Python's dynamic type checking.

We often call languages that only perform dynamic type checking *dynamically typed language*. Conversely, *statically typed languages* are languages that perform *some* type checking statically.

This asymmetry in the definition comes about because it is difficult to do *all* type checking at compilation time. Consider the following rule from the Java language specification JLS-11, Section 10.4:

> *All array accesses are checked at run time; an attempt to use an index that is less than zero or greater than or equal to the length of the array causes an ArrayIndexOutOfBoundsException to be thrown (§15.10.4).*

Making sure that an array access stays within bounds is indeed a type check, as we can see more clearly in Mystery:

[2] Assuming that the function print() is bound to console.log().

```
VAR a : ARRAY [0 TO 10] OF INTEGER;
VAR x : [0 TO 10];
VAR y : INTEGER
BEGIN
    ...
    a[x] := 1;
    a[y] := 1;
    ...
END
```

Here, the array access a[x] is free of any type errors, since the type of x guarantees that x is always in the index range of the array a. However, the array access a[y] is not type safe: y may contain any integer, including the number -1, which is not of type [0 **TO** 10] and thus outside of the array's index range.

In other languages, subrange types are not an explicit part of the type system, but they still exist implicitly, as in the above-mentioned rule from the Java language specification. Array out-of-bounds checks, as well as null-pointer exceptions, are well-known examples of type checks that statically typed languages have to defer to runtime, in order to be strongly typed[3].

C and C++, meanwhile, are statically typed but do not perform such out-of-bounds checks, which is one reason for why these languages are not strongly typed.

4 Type Checking

To express the type system of a language, we can use the same tools as for describing their natural semantics. Consider the language AddEqIf, whose syntax we have in Figure 1, and whose semantics we see in Figure 2. This is again a language that consists purely of expressions, but this time we have two types of values, the natural numbers (the elements of *nat*) and the booleans (true and false).

This language has three operators: Addition (+), which is only defined if both parameters are in *nat*, equality (=), which allows us to compare two natural numbers or two booleans, and the conditional expression `if` e_c `then` e_t `else` e_f, which checks if e_c evaluates to true (and in that case gives us the result of evaluating e_t) or if $c \Downarrow$ false (in which case it gives us the result of evaluating e_f).

To formalise these constraints, we introduce two separate types into the language: Bool, the type of booleans, and Nat, the type of natural numbers. Thus, we can say e.g.

$$\texttt{true} : \mathsf{Bool}$$

and

$$\texttt{14} : \mathsf{Nat}$$

We can now use these *typing judgements* to formalise the typing constraints that we sketched informally earlier. First, we want to say that e_1+e_2 is allowed in our language if $e_1 : \mathsf{Nat}$ and $e_2 : \mathsf{Nat}$. Let's call this rule 'rule A'. If we also know that $\texttt{14} : \mathsf{Nat}$ (let's call this 'rule B'), then we can now say that `14 + 14` is allowed according to rules A and B. However, rules A and B together are not enough to let us see that `(14 + 14) + 14` is allowed! The reason for this is that the rules don't tell us what the type of `14 + 14` is, only that it is 'allowed'.

Programming language designers have come up with an elegant solution that allows us to simultaneously say 'this expression has that type' and 'the type system allows this expression': the type system allows an expression *if and only if it can assign a type to the expression.*

Definition 8 *An expression e is* well-typed *if and only if we can show $e : \tau$ for some type τ.*

Let us apply this idea to our + operator. If both arguments to + have the type Nat, then the entire expression has the type Nat.

[3] Particularly clever compilers may be able to eliminate some of these checks, but the problem of eliminating them entirely is undecidable.

$$
\begin{array}{rcl}
\langle expr \rangle & \longrightarrow & nat \\
& | & \texttt{true} \mid \texttt{false} \\
& | & \langle expr \rangle \texttt{ + } \langle expr \rangle \\
& | & \langle expr \rangle \texttt{ = } \langle expr \rangle \\
& | & \texttt{if } \langle expr \rangle \texttt{ then } \langle expr \rangle \texttt{ else } \langle expr \rangle
\end{array}
$$

Figure 1: Syntax for the language AddEqIf.

$$
\boxed{\begin{array}{rcl} e, e_i & \in & \langle expr \rangle \\ v, v_i & \in & \{\text{true}, \text{false}\} \cup nat \end{array}}
\qquad
\frac{v \in nat}{v \Downarrow v}\ (nat)
\qquad
\frac{}{\texttt{true} \Downarrow \text{true}}\ (true)
\qquad
\frac{}{\texttt{false} \Downarrow \text{false}}\ (false)
$$

$$
\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad v = v_1 + v_2}{e_1 \texttt{ + } e_2 \Downarrow v}\ (add)
\qquad
\frac{e_1 \Downarrow v \quad e_2 \Downarrow v}{e_1 \texttt{ = } e_2 \Downarrow \text{true}}\ (eq\text{-}true)
\qquad
\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad v_1 \neq v_2}{e_1 \texttt{ = } e_2 \Downarrow \text{false}}\ (eq\text{-}false)
$$

$$
\frac{e_1 \Downarrow \text{true} \quad e_2 \Downarrow v_2}{\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \Downarrow v_2}\ (if\text{-}true)
\qquad
\frac{e_1 \Downarrow \text{false} \quad e_3 \Downarrow v_3}{\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \Downarrow v_3}\ (if\text{-}false)
$$

Figure 2: Natural Semantics for the language AddEqIf. Here, *nat* is the set of natural numbers, $\mathbb{N}$, as part of the input language

We can use inference rule notation, as for operational semantics, to compactly write down this rule:

$$
\frac{e_1 : \mathsf{Nat} \quad e_2 : \mathsf{Nat}}{e_1 \texttt{ + } e_2 : \mathsf{Nat}}\ (t\text{-}add)
$$

As with operational semantics, we can use this rule recursively, to allow us to see not only that the type system allows `(14 + 14) + 14`, but also that the type of `(14 + 14) + 14` is Nat:

$$
\dfrac{\dfrac{\dfrac{}{\texttt{14} : \mathsf{Nat}}\ rule\ B \quad \dfrac{}{\texttt{14} : \mathsf{Nat}}\ rule\ B}{\texttt{14 + 14} : \mathsf{Nat}}\ (t\text{-}add) \quad \dfrac{}{\texttt{14} : \mathsf{Nat}}\ rule\ B}{\texttt{(14 + 14) + 14} : \mathsf{Nat}}\ (t\text{-}add)
$$

If we now generalise over 'rule B' to allow any elements of *nat* to have the type Nat, we can allow all natural numbers and all additions involving natural numbers to be well-typed (and, hence, for our type system to allow them):

$$
\frac{v \in nat}{v : \mathsf{Nat}}\ (t\text{-}nat)
$$

Now that we have formalised the type system as far as we need it for + operator. This will allow the type system to catch the error in the expression `true + 2`: even if we give the booleans their respective, as in

$$
\texttt{true} : \mathsf{Bool} \qquad \texttt{false} : \mathsf{Bool}
$$

we will not be able to use our typing rules to type `true + 2`, since `true` : Bool, while the + operator requires Natparameters.

Let us move on to formalise the = operator, which should test for equality:

$$
\frac{e_1 : \mathsf{Nat} \quad e_2 : \mathsf{Nat}}{e_1 \texttt{ = } e_2 : \mathsf{Bool}}\ (t\text{-}eq\text{-}nat)
$$

This time, the resultant type is Bool, the type of booleans, so `1 = 2` : Bool.

$$\texttt{true} : \mathsf{Bool} \qquad \texttt{false} : \mathsf{Bool} \qquad \frac{v \in nat}{v : \mathsf{Nat}}\ (t\text{-}nat) \qquad \frac{e_1 : \mathsf{Nat} \quad e_2 : \mathsf{Nat}}{e_1 \texttt{ + } e_2 : \mathsf{Nat}}\ (t\text{-}add)$$

$$\frac{e_1 : \mathsf{Nat} \quad e_2 : \mathsf{Nat}}{e_1 \texttt{ = } e_2 : \mathsf{Bool}}\ (t\text{-}eq\text{-}nat) \qquad \frac{e_1 : \mathsf{Bool} \quad e_2 : \mathsf{Bool}}{e_1 \texttt{ = } e_2 : \mathsf{Bool}}\ (t\text{-}eq\text{-}bool) \qquad \frac{e_1 : \mathsf{Bool} \quad e_2 : \alpha \quad e_3 : \alpha}{\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : \alpha}\ (t\text{-}if)$$

Figure 3: Type system for the language AddEqIf. This type system can be checked completely statically.

This type system will refuse to assign a type to `1 = true`; an implementation of our language would print a type error for such an expression. This is working as intended: we can't accidentally compare booleans with natural numbers.

However, the type system also stops us from comparing booleans with each other, so it will refuse to type the expressions `false = false` or `false = (1 = 2)`. There is no reason for us to be so restrictive — our operational semantics rules *eq-true* and *eq-false* can handle the booleans perfectly well. We can address this either by making the rule *t-eq-nat* more general, or by introducing another rule:

$$\frac{e_1 : \mathsf{Bool} \quad e_2 : \mathsf{Bool}}{e_1 \texttt{ = } e_2 : \mathsf{Bool}}\ (t\text{-}eq\text{-}bool)$$

Finally, let us examine the expression `if` e_c `then` e_t `else` e_f . Here, e_c must have type Bool — if e_c is a number, then our operational semantics rules would get *stuck* here (since neither of them applies).

It is less clear what the types on e_t and e_f should be, and what type we want for the `if...` expression itself. If $e_t : \mathsf{Nat}$ and $e_f : \mathsf{Nat}$, then the conditional will always compute a Nat, so it makes sense to set `if` c `then` e_t `else` $e_f : \mathsf{Nat}$. On the other hand, if $e_t : \mathsf{Bool}$ and $e_f : \mathsf{Bool}$, then we would want to set the type `if` c `then` e_t `else` $e_f : \mathsf{Bool}$.

But what should we do if the types of e_t and e_f disagree? Our operational semantics can give us a meaningful result for the program

```
4 + (if true then 5 else false)
```

so it would be nice to be able to have a rule such as the following:

$$\frac{e_c \Downarrow \mathrm{true} \quad e_t : \mathsf{Nat}}{\texttt{if } e_c \texttt{ then } e_t \texttt{ else } e_f : \mathsf{Nat}}\ (t\text{-}if\text{-}true\text{-}dynamic)$$

Now, mathematics is versatile and patient, and nothing prevents us from writing down such a rule. However, the type judgement in this rule depends on the result of evaluating the program ($c \Downarrow \mathrm{true}$). This means that we can only do this kind of type checking dynamically. Moreover, since we need the result of this rule to typecheck e.g. the expression `4 + (if true then 5 else false)`, using such a rule means that we would potentially have to defer all or almost all type checking to runtime.

While this is a possibility, we here instead opt for a type system that we can check statically. Thus, we require that the types of e_t and e_f agree — either both are Bool or both are Nat.

As with our earlier rule for the `=` operator, we could accomplish this form of variability by having two rules, one for Bool and one for Nat.

Figure 3 summarises our type system, and rule *t-if* describes the typing rule for `if...` expressions. Here we require $e_2 : \alpha$ and $e_t : \alpha$, where α is a *type variable*. That means that our rule doesn't know which type e_1 and e_2 have, just that both of them have the same type, and that that type is also the type that we assign to the entire `if...` expression.

As we can see, types are an abstraction over the possible values that our expressions may assume, and the same holds when we move on to variables, statements, and more general language constructs. Thus, we can think of types as a framework for modelling the possible behaviour of our program. Types allow us to confine this behaviour, and the expressive power of the type system governs how precise and how accurate these models can be.

4.1 Properties of Type Systems

We can construct any arbitrary type system that we find interesting, but not all of them will be useful to end users. Language designers have found a number of valuable properties that they want type systems to have.

The first of these properties is that *the type system should allow us to predict the outcome of computations.*

In most practical programming languages, the set of values is at least partly a subset of the set of expressions. For instance in our language AddEqIf, the booleans and natural numbers are part of the input language (the left-hand side of the evaluation arrow), but they also form the output language (the right-hand side of the arrow). This means that if we evaluate an expression

$$e \Downarrow v$$

we can assign a type both to e and to the result v, simply by using the same typing rules for the (right-hand side) values. Intuitively, when the type system promises us that an expression will give us an integer value, we want the evaluation relation to also deliver that integer value, in other words, evaluation and the type system should agree. We call this property *type preservation*:

Definition 9 *A type system has the* type preservation *(or* subject reduction*) property if for any* $e \Downarrow v$*,* $e : \tau$ *implies* $v : \tau$*.*

Type preservation is one of the three properties that we want a type system to have:

1. *Type preservation*: The predictions of the type system agree with the evaluation rules.
2. *Progress*: The type system only assigns a type if the evaluation rules will not get 'stuck' due to a missing evaluation rule.

   This is not the same as guaranteeing that the program itself terminates — a well-typed Java program may certainly enter an endless loop — but it does guarantee that the language implementation will never run into a situation in which it doesn't know what to do next. For a more precise formalisation of this property, we would need a different form of semantics (structural operational semantics) that we are not covering in this course.
3. *Termination*: We want the type system to be *decidable*, that is, we want an automatic mechanism that performs type checking.

Not all language designers agree that termination is necessary (though it is certainly desirable). For instance, the programming language Agda is statically typed but has an undecidable type system: the programmer needs to help the language implementation to see whether the program is correctly typed.

4.2 Dynamic Type Checking

Whenever a typing rule depends on the evaluation relation $\Downarrow$, we must defer type checking to runtime. In some cases, only a small amount of type checks must be done at runtime; for example, in a language like Modula-3, we can do most of the type-checking statically, while we can implement the remaining checks (array out-of-bounds checking and null pointer checking) fairly efficiently:

- For array out-of-bounds checking, all we need to do is to compare array indices against array bounds before every access. Since array indices are always integers, all we need to do is to ask the CPU to compare these integers against the maximum and minimum index that is allowed for the array.

  These checks still take execution time, which is why some languages (C, C++) choose not to do them, sacrificing robustness for decreased execution cost.

- For null-pointer accesses the check is even simpler: if a piece of code tries to read from or write to a null pointer, the CPU automatically interrupts the program and calls the operating system. Without going into detail, this means that null pointer checking comes for free as long as there are no null pointer errors, but when we do run into a null pointer dereferences, the runtime cost is relatively high.

However, languages like Python or JavaScript that are dynamically typed defer *all* type checking to runtime. That means that for any value we compute, the language has to be able to tell *at run time* what the type of the value is. Thus, these languages take up extra storage to be able to identify values.

Tagging For example, consider a language that has 32-bit integer and floating point values and needs to be able to distinguish them at runtime. One approach is to reserve a single bit and mark it as 1 if the value is an integer, and as 0 if the value is a floating point number. This effectively reduces the range of integers to 31 bits and slightly reduces the precision of floating point numbers. It also requires additional checks before and after operations on the numbers.

This approach is called *tagging*. Traditional implementations of LISP as well as Standard ML of New Jersey choose this implementation scheme (even though the latter is fully statically typed, it uses these tags to simplify parts of its runtime system).

An alternative approach is to store values as pairs $\langle type, value \rangle$. This approach scales up to a larger number of types and avoids sacrificing precision or integer range, but takes up more memory.

5 Overloading

Many languages allow programmers to re-use operators for different purposes. For example, in C we can use the + operator for adding not only different kinds if integers, but also floating point numbers. In Python and Java, we can use the same operator to concatenate strings:

```
i = 1 + 2
f = 1.0 + 2.5
s = "foo" + "bar"
```

Each of these three uses of the + operator is a fundamentally different operation. We thus say that the operator + is *overloaded*:

Definition 10 *An operator is* overloaded *if it performs different actions depending on the types of its arguments.*

Overloading is also known as *ad-hoc polymorphism*; we will discuss it more when we examine the behaviour of subprograms.

From the perspective of operational semantics, overloading is interesting because it corresponds to the occurrence of typing judgements in operational semantics rules. To see this, let us extend AddEqIf with strings:

$$\texttt{"string"} : \mathsf{String}$$

Here, String is the type of strings.

We now want to overload the + operator. First, we ensure that numeric addition only applies if the parameters evaluate to natural numbers:

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad v_1 : \mathsf{Nat} \quad v_2 : \mathsf{Nat} \quad v = v_1 + v_2}{e_1 + e_2 \Downarrow v} \ (add)$$

Now we add a rule that concatenates strings:

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad v_1 : \mathsf{String} \quad v_2 : \mathsf{String} \quad v = v_1 +\!\!+ v_2}{e_1 \ \mathsf{+} \ e_2 \Downarrow v} \ (concat)$$

Here, $+\!\!+$ is the mathematical string concatenation operator — we leave its precise definition (which depends on the definition of strings) as an exercise to the reader.

We see how both rules depend on the typing judgement (:), unlike the rules from Figure 2. This dependence is the source of the operator's overloading.

5.1 User-defined Operator Overloading

In some languages, such as C++, users can overload existing operators for their own types (or even for existing types). In others, like Python and Smalltalk, such operators are just 'syntactic sugar' and get translated to a subroutine call; in essence, this also allows user-defined operator overloading by piggybacking on a different language mechanism.

In Haskell and Rust, there exists a systematic overloading mechanism called *Type Classes*, which we will look into later in this course.