Quick Introduction to Type Systems

David B. MacQueen, 2012

A type is a description that characterizes the expected form of the result of a computation. In particular, if e is an expression, a typing

$$e : \texttt{int}$$

is an assertion that when e is evaluated, its value will be an integer. We call this assertion a *typing judgment*. But beyond predicting the form of the ultimate value of e, this typing judgment also requires that e be *well-typed*, meaning that it is internally consistent, and consistent with its context or environment in the case where it contains free variables.

For instance, what does

$$2 + 1 : \texttt{int}$$

mean? It means that evaluation of $2 + 1$ is expected to produce an integer. But it also requires that the $+$ operator is used properly, *i.e.* it has two arguments, here 2 and 1, which must both be integers.

Let us assume that there is another basic type `bool` and that `true` is a constant having type `bool`. Here is an *invalid* typing judgment (meaning it is not provable using the rules to be introduced below):

$$\texttt{true} : \texttt{int}$$

and here is another:

$$2 + \texttt{true} : \texttt{int}$$

The first is invalid because it mischaracterizes the value `true` as an integer. The second is invalid because $2 +$ `true` is not consistent: `+` is improperly applied to an integer and a boolean and not two integers as it should be. We can consider an expression like $2 +$ `true` to be a kind of nonsense, and we want to use valid typing judgments to weed out such nonsensical expressions.

How do we establish which typing judgments are valid and which are invalid? We use a system of rules to *derive* the valid typing judgments. These rules constitute a specialized logic whose statements are typing judgments that relate expressions and types. If we can prove a judgment of the form

$$e : \tau$$

is should be the case that

1. e is well-typed, meaning that its components fit together properly according to the rules (*e.g.* , operators are applied to the right kinds of arguments), and
2. when e is evaluated, and its evaluation terminates, it produces a value described by τ.

Note that the proviso of termination in (2) is necessary, because a type system is generally not able to answer the question of whether an expression terminates. Typing judgments that cannot be derived by our rules are deemed *invalid*.

Here are some sample typing rules that can be used to derive typing judgments:

Integer constants:

$$(1) \quad \frac{}{\texttt{n} : \texttt{int}} \qquad \text{(where n is an integer constant)}$$

This rule simply says that an integer constant like 3 has the type `int`. The rule is schematic, since it contains a metavariable "n" that ranges over arbitrary integer constants. Here is a trivial, one-step derivation using rule (1):

$$\frac{}{2 : \texttt{int}}\ (1)$$

Function applications:

$$(2)\quad \frac{e_1 : \tau_1 \to \tau_2 \qquad e_2 : \tau_1}{e_1\, e_2 : \tau_2}$$

This rule has a conclusion "$e_1\ e_2\ :\ \tau_2$" concerning a function application which appears below the horizonal line, where e_1 denotes the function being applied, and e_2 denotes its argument. To derive this conclusion, we have to first derive to premises, the judgments above the line:

$e_1 : \tau_1 \to \tau_2$ – e_1 denotes a function mapping τ_1 to τ_2

$e_2 : \tau_1$ – e_2 denotes a value of type τ_1

The rule expresses the fact that to be well typed, the type of the argument must agree with the domain (τ_1) of the function being applied. This constraint is expressed by the fact that the same type metavariable τ_1 appears in both the premises.

Like (1), this is also a rule schema containing metavariables e_1 and e_2 ranging over expressions, and τ_1 and τ_2 ranging over types. A particular use of the rule instantiates these metavariables with particular expressions and types.

Rule (2) also illustrates another common property of expression typing rules – it is *compositional*. This means that the type of a compound construct, such as the application expression e_1e_2 in this case, is derived from the types of its constituent subexpressions, e_1 and e_2. This is a very useful property, particularly when we want to implement a *type checker* (see below). But for more complex type systems, compositionality is sometimes sacrificed for the sake of a more flexible or expressive type system (*e.g.* when subtyping is introduced).

To illustrate how these rules are used in a derivation, assume we have the rule (axiom) that

$$(3)\quad \frac{}{+ : \texttt{int} \to \texttt{int} \to \texttt{int}}$$

(this just asserts the known type of $+$, where $+$ is taken to be a *curried* operator). Now to derive the judgment $2 + 1 : \texttt{int}$ (which translates to $+21 : \texttt{int}$) we construct the following derivation:

$$\dfrac{\dfrac{\dfrac{}{+ : \texttt{int} \to \texttt{int} \to \texttt{int}}\ (3) \qquad \dfrac{}{2 : \texttt{int}}\ (1)}{+\,2 : \texttt{int} \to \texttt{int}}\ (2) \qquad \dfrac{}{1 : \texttt{int}}\ (1)}{+\,2\,1 : \texttt{int}}\ (2)$$

The derivation takes the form of a tree of rule instances, where each rule instance is labelled with its rule number. This derivation contains one instance of rule (3), two instances of rule (1) and two instances of rule (2). The conclusion shows that the expresion $+21$ is internally consistent, given the types of its constituent symbols $+$, 2, and 1, and the type of its value is int (assuming it terminates).

If we try to construct such a derivation for $2 + \texttt{true}$ (or $+\ 2\ \texttt{true}$), we will fail. Assume rule (4) for boolean constants:

$$(4)\quad \frac{}{\texttt{true} : \texttt{bool}}$$

We attempt to construct a derivation as follows:

$$\dfrac{\dfrac{\dfrac{}{+ : \texttt{int} \to \texttt{int} \to \texttt{int}}\ (3) \qquad \dfrac{}{2 : \texttt{int}}\ (1)}{+\,2 : \texttt{int} \to \texttt{int}}\ (2) \qquad \dfrac{}{1 : \texttt{int}}\ (1)}{?}\ (2?)$$

But we can't complete the derivation with an instance of rule (2) because the type of the argument `true`, namely `bool`, does not agree with the domain of the function $(+\ 2)$, namely `int`.

You might ask if there is some more clever and devious way to construct a valid derivation, but it turns out (for this particular set of rules) that we don't have any real choice in how the derivation is structured – the rules are such that the structure of a derivation is always a direct reflection of structure of the expression being typed. Such a rule system is called "syntax directed".

[Note: This property of being syntax directed is not preserved in some more complicated type systems, for instance when subtyping is included, but the desirable property of the type system being deterministic is usually maintained, on way or another (see, e.g, Chapter 16 of Pierce, `Types for Programming Languages`).]

So far, we have only considered simple expressions that do not contain variables or variable bindings, but this is not realistic. So how to we establish judgments like the following?

(a) $x + 1 : \texttt{int}$

Well first of all, this should only be valid if the variable x represents (or ranges over) integers. If x is bound to `true`, for instance, this judgment should be considered false.

So typing judgments where the expression contains free variables have to be considered with respect to some context that tells us what the types of those free variables are. We will represent such contexts as a sequence of associations of variables and types, *e.g.*

$$\begin{aligned}\Gamma_1 &= x : \texttt{int} \\ \Gamma_2 &= f : \texttt{int} \to \texttt{bool},\ y : \texttt{int}\end{aligned}$$

These typing contexts are also known as *type environments*, and we can think of them as finite functions mapping variables to types.

Now we need to revise the notion of a typing judgment to include a typing context that specifies the types of any free variables in the expression:

$$\Gamma \vdash e : \tau$$

This modified judgment asserts that e has the type τ under the assumption that free variables occuring in e have the types specified in Γ. So (a) needs to be changed to something like:

(b) $x : \texttt{int} \vdash x + 1 : \texttt{int}$

A new typing rule is needed to allow us to use the typing context when we need to determine the type of a variable:

$$(5)\quad \frac{}{\Gamma \vdash x : \tau} \qquad \text{(where } x : \tau \in \Gamma\text{)}$$

or equivalently,

$$(5)\quad \frac{}{\Gamma \vdash x : \Gamma(x)}$$

Note that in the second version of this rule, we are interpreting the context Γ as a (finite) function mapping variables to types and denoting the type that it assigns to x by $\Gamma(x)$.

Now a derivation of (b) will look like this, assuming $\Gamma_0 = x : \texttt{int}$:

$$\dfrac{\dfrac{\dfrac{}{\Gamma_0 \vdash + : \texttt{int} \to \texttt{int} \to \texttt{int}}\,(3) \quad \dfrac{}{\Gamma_0 \vdash x : \texttt{int}}\,(5)}{\Gamma_0 \vdash +\,2 : \texttt{int} \to \texttt{int}}\,(2) \quad \dfrac{}{\Gamma_0 \vdash 1 : \texttt{int}}\,(1)}{\Gamma_0 \vdash +\,2\,1 : \texttt{int}}\,(2)$$

Notice that all the earlier rules we used before have to be modified to add type contexts:

$$(1)\quad \frac{}{\Gamma \vdash \mathtt{n} : \mathtt{int}} \qquad \text{(where n is an integer constant)}$$

meaning the conclusion judgment holds for arbitrary type contexts and arbitrary integer constants n.

$$(2)\quad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{e_1\ e_2 : \tau_2}$$

where the same context Γ must be used in all three judgments in the rule, and so on for the other rules.

But where do contexts come from? Do we arbitrarily conjure them from thin air? No, contexts may be supplied as initial assumptions about variables and primitive operators, but they can also evolve as part of derivations of certain kinds of compound expressions involving variable bindings. An example where a context grows by the addition of a new variable typing, we have the rule for function expressions (commonly known as λ-expressions):

$$(6)\quad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash (\lambda x\colon \tau_1.\, e) : \tau_1 \to \tau_2}$$

($\lambda x\colon \tau_1.\, e$ is an expression denoting a function whose formal parameter is x, constrained to be of type τ_1, and whose body is e. A typical such expression is: $\lambda x : \mathtt{int}.\, x + 1$, denoting the function that takes an integer argument and returns its successor.)

Here in the premise of the rule, we are allowed to use the additional assumption that $x : \tau_1$ while deriving the type of the body e. The notation $\Gamma, x : \tau_1$ represents an extension of the context Γ with a new mapping from x to τ_1. If Γ already contained a typing for x, this new mapping overrides the existing one.

A similar rule for typing let-expressions would be:

$$(7)\quad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathtt{let}\ x\colon \tau_1\ =\ e_1\ \mathtt{in}\ e_2 : \tau_1 \to \tau_2}$$

We often start with a base context Γ_0 that includes type assignments for the basic operators like $+$ and $*$, and other global constants.

$$\Gamma_0 = +, * : \mathtt{int} \to \mathtt{int} \to \mathtt{int}, \ldots$$

This global context eliminates the need for special typing axioms like rule (3) for typing primitive operators. Then as we unwind binding constructs like and let, type assignment for local bound variable get added to the context for use in the scope of the respective bindings. If we want to type an expression e that contains free variables as well as global constants, we must provide an appropriate context Γ that extends Γ_0 with type assignments for all the free variables that occur in e. Thus:

$$\Gamma_0, x : \mathtt{int} \vdash x + 2 : \mathtt{int}$$

Here are a couple of questions one can ask about a type system presented as a set of typing rules of the sort we have considered above.

1. *Uniqueness*: For any expression e and typing context Γ, is there at most one type τ such that one can derive the typing judgment $\Gamma \vdash e : \tau$? Of course we know that for some *ill-typed* expressions such as $2 + \mathtt{true}$, there will be no such τ.

2. Assuming that there is a unique τ such that $\Gamma \vdash e : \tau$, is there a unique derivation of that judgment, or could there be multiple derivations.

3. Is there an algorithm for deciding, given Γ and e, whether $\Gamma \vdash e : \tau$ is derivable for some τ, *i.e.* is typability decidable?

Type checking. The process of type checking takes an expression e and a context Γ and computes a type τ such that $\Gamma \vdash e : \tau$, reporting a type error if such does not exist. For cases where the typing rules are compositional, this is usually fairly straightforward to implement, since one can recurse over the structure of e, computing the types of is components, and using them to construct the type of e, providing some consistency tests are passed (like agreement of function and argument types for rule (2) above).

Type checking gets more complicated when non-compositional rules are involved, as in the case where subtyping is used, or in the case of polymorphic type inference in ML or Haskell. But it is still highly desirable that there be a decidable (that is, always terminating) type checking algorithm that either computes the type of an expression or reports an error indicating that the expression is ill-typed.