Lecture Notes: Typing Rules

17-363/17-663: Programming Language Pragmatics (Fall 2021)
Jonathan Aldrich
jonathan.aldrich@cs.cmu.edu

1 Semantic Analysis

After parsing, semantic analysis and (intermediate) code generation are the next two phases of a typical compiler. The semantic analyzer starts by constructing an abstract syntax tree (AST). If the parser produces a concrete syntax tree, then the AST can be produced by traversing it. Otherwise, the AST is typically produced by a series of inline *semantic actions* that are triggered when various productions are recognized by the parser.

The semantic analyzer's main job is then to enforce semantic rules and to gather information that is needed by the code generator. The information gathering and enforcement focuses primarily on evaluating name bindings and applying typing rules. Name bindings tell us which variable declaration corresponds to each use in the program, while types tell us what operations to apply. For example, if an add operation is applied to two integers, we will later need to generate an integer add instruction. If the add operation is applied to an integer and a floating point number, then we will later need to generate code to convert the first integer to a floating point number and use a floating point add instruction applied to the two numbers. Typically the information generated by the semantic analyzer is stored either in the AST itself, as annotations, or in auxiliary data structures such as the symbol table.

2 Typing Rules

We can capture the work of the semantic analyzer with typing rules, analogous to the operational semantics rules we described earlier. We'll demonstrate the idea by applying typing rules to the little language we described earlier:

$$\begin{array}{lll} e & ::= & x \mid n \mid e + e \mid \text{let } x = e \text{ in } e \mid x : \tau \Rightarrow e \mid e(e) \\ \tau & ::= & \text{nat} \mid \tau \rightarrow \tau \end{array}$$

We've extended the syntax in two ways. First of all, function arguments are now annotated with a type τ. This ensures that whoever calls a function knows what kind of value to pass in, and it also enables the typechecker to verify that the right kind of value is actually passed. Second, we've given a grammar for types: right now types just consist of natural numbers (nat) and function types. A function type $\tau_1 \rightarrow \tau_2$ denotes a function that takes an argument of type τ_1 and returns a result of type τ_2 (as before, our simple model only includes 1-argument functions; this is easy but a bit tedious to generalize).

To reason about typing a program, we'll use the judgment $e : \tau$ which can be read "e has type τ." We'll expand this judgment shortly, when we talk about variables. For now, let's write our first inference rule:

$$\frac{}{n : \text{nat}}\ \textit{T-num}$$

This rule simply states that the type of a number literal is nat. We can write a more interesting rule for arithmetic operations like addition. The premises check that the things we are adding up are numbers, and the conclusion tells us that we get a number as a result:

$$\frac{e_1 : \text{nat} \quad e_2 : \text{nat}}{e_1 + e_2 : \text{nat}}\ \textit{T-plus}$$

How shall we typecheck a variable x? Intuitively, the only way we can know the type of x is if we have kept track of the type with which it was declared. We will therefore extend our judgment to the form $\Gamma \vdash e : \tau$, which can be read "In the context of typing environment Γ, expression e has type τ." Γ is a symbol that represents a list of bindings from variable to type. We can express this with a grammar:

$$\Gamma \quad ::= \quad \bullet \mid \Gamma, x : \tau$$

Here $\bullet$ represents the empty typing environment, and then the other alternative builds up one environment from another by adding a binding from x to type τ. We will write $x : \tau \in \Gamma$ to mean that the last binding for x in Γ is to τ. It is important to distinguish "last binding" here because in general Γ might contain several bindings for a variable x if that variable has an outer binding that shadows an inner one.

Now we can write typing rules for variables and let expressions:

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \textit{T-var} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2}\ \textit{T-let}$$

The first rule says that a variable x has type τ if the binding $x : \tau$ appears in the typing environment Γ. The second rule, for let, first typechecks the expression e_1, discovering that in environment Γ it has type τ_1. We then typecheck the body of the let, e_2, in a typing environment that consists of Γ extended with a binding $x : \tau_1$. The type of the body τ_2 is also the type of the whole let expression.

The form of judgment we are using, with an environment Γ to the left of the turnstile symbol $\vdash$, is called a *hypothetical judgment*, because it means that e has type τ assuming (hypothetically) that the variables in Γ have the types given there. We'll modify our typing rules for number literals and addition to also use this hypothetical judgment form:

$$\frac{}{\Gamma \vdash n : \text{nat}}\ \textit{T-num} \qquad \frac{\Gamma \vdash e_1 : \text{nat} \quad \Gamma \vdash e_2 : \text{nat}}{\Gamma \vdash e_1 + e_2 : \text{nat}}\ \textit{T-plus}$$

In the case of *T-num*, the typing environment isn't used; we just include it so that the form of the judgment is consistent across all rules. This consistency is important because of rules like *T-let* that have the judgment in the premise, and assume a standard form for it. The *T-plus* rule doesn't use Γ directly, but it's very important to pass it on, because one of the subexpressions e_1 or e_2 might be a variable, or have a variable further inside it. So *T-plus* typechecks the subexpressions in the same typing environment that was used for the overall addition expression.

Exercise 1. Show the derivation for typing the following program:
let $x = 1$ in $x + 2$

Answer:

$$\dfrac{\dfrac{}{\bullet \vdash 1 : \text{nat}}\ \textit{T-num} \quad \dfrac{\dfrac{}{\bullet, x : \text{nat} \vdash x : \text{nat}}\ \textit{T-var} \quad \dfrac{}{\bullet, x : \text{nat} \vdash 2 : \text{nat}}\ \textit{T-num}}{\bullet, x : \text{nat} \vdash x + 2 : \text{nat}}\ \textit{T-plus}}{\bullet \vdash \text{let } x = 1 \text{ in } x + 2 : \text{nat}}\ \textit{T-let}$$

We can now write the last two typing rules, for function definition and function application:

$$\dfrac{\Gamma, x : \tau_2 \vdash e_1 : \tau_1}{\Gamma \vdash x : \tau_2 \Rightarrow e_1 : \tau_2 \to \tau_1}\ \textit{T-fn} \qquad \dfrac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1(e_2) : \tau_1}\ \textit{T-apply}$$

The function typing rule checks the body of the function assuming that the argument has the annotated type τ_2. The apply rule requires the expression in function position to have a function type, and the expression in argument position to have a type matching the function's argument type; the overall result is the function's result type.