Lecture Notes: Typing Rules

17-363/17-663: Programming Language Pragmatics (Fall 2022)
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. One major part of this information gathering and enforcement concerns name binding and type checking. We will use these tasks to illustrate the kind of work that the semantic analyzer does. Name bindings tell us which declaration corresponds to each use of a name in the program, while types help us ensure that names are used consistently, and provide us with useful context. For example, if an add operation (+) is applied to two integers, we will need to arrange (sometime later) to generate an integer add instruction. If the add operation is applied to an integer and a floating point number, then we must either produce an error (as in Ada) or arrange to generate code that will convert the integer to a floating point number (as Java does) and use a floating point add instruction. Typically the information generated by the semantic analyzer is stored either as annotations in the AST itself or as data in the symbol table or other auxiliary structures

Name binding and type checking can be performed at run time rather than in a semantic analysis compiler phase, but there are disadvantages to doing so. Looking back at the dynamic semantics we defined for the calculator language, several of the rules have premises that check the types of values being computed. If these checks are delayed until run time, their overhead will slow the program down. Moreover, if a check fails, the program will stop with a run-time error. If possible, we'd prefer to pay the cost of checks—and to learn of errors—ahead of time, before a program is placed into production use. A *typechecker* meets these goals, by computing a type for every variable, function, and expression in the program, and by checking for consistency among the types of values and the operations performed on them.

Just as an interpreter implements the dynamic semantics of a program by traversing its AST and computing the output value or environment, a typechecker is implemented by traversing the program's AST and computing the types of subtrees. An interpreter relies on an environment mapping variables to the values they contain; a typechecker will likewise rely on a *type context* (usually represented by the Greek letter Γ, pronounced "gamma") that maps variables x to their types (usually represented by the Greek letter τ, pronounced "tau"). In the language we have been studying, the lambda calculus with natural numbers, we can describe types with the abstract grammar:

$$\tau \quad ::= \quad \text{nat} \mid \tau \to \tau$$

Here nat is the type of natural numbers, and $\tau_1 \to \tau_2$ is the type of 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).

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 a small variant of the lambda calculus with numbers that we described earlier:

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

We've extended the syntax of functions to annotate their argument 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.

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:

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

The function typing rule checks the body of the function assuming that the argument has the annotated type τ_2:

$$\frac{\Gamma, x : \tau_2 \vdash e_1 : \tau_1}{\Gamma \vdash x : \tau_2 \Rightarrow e_1 : \tau_2 \to \tau_1}\ \textit{T-fn}$$

Exercise 2. Before turning the page, try writing a rule for typing application expressions of the form $e_1(e_2)$.

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1(e_2) : \tau_1} \textit{ T-apply}$$

The 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.