

CS4120/4121/5120/5121—Spring 2026
Eta Type System Specification
 Cornell University
 Version of February 14, 2026

0 Changes

- No changes yet.

1 Types

The Eta type system uses a somewhat bigger set of types than can be expressed explicitly in the source language:

$$\begin{array}{llll}
 t ::= \text{int} & T ::= (t_1, t_2, \dots, t_n) \quad n \geq 0 & R ::= \text{unit} & \sigma ::= \text{var } t \\
 | \text{bool} & & | \text{void} & | \text{ret } T \\
 | t[] & & & | \text{fn } T \rightarrow T'
 \end{array}$$

Ordinary types expressible in the language are denoted by the metavariable t , which can be `int`, `bool`, or an array type.

The metavariable T denotes a possibly empty sequence of types, which is useful for procedures, functions, and multiple assignments.

The metavariable R represents the outcome of evaluating a statement, which can be either `unit` or `void`. The `unit` type is the type of statements that *might* complete normally and permit the following statement to execute. The type `void` is the type of statements such as `return` that *never* pass control to the following statement. The `void` type should not be confused with the C/Java type `void`, which is actually closer to `unit`.

The set σ is used to represent typing-environment entries, which can either be normal variables (bound to `var t` for some type t), return types (bound to `ret T`), functions (bound to `fn T → T'` where $T' \neq ()$), or procedures (bound to `fn T → ()`), where the “result type” $()$ indicates that the procedure result contains no information other than that the procedure call terminated.

2 Type-checking expressions

To type-check expressions, we need to know what bound variables and functions are in scope; this is represented by the typing context Γ , which maps names x to types σ .

The judgment $\Gamma \vdash e : t$ is the rule for the type of an expression; it states that with bindings Γ we can conclude that e has the type t .

We use the metavariable symbols x or f to represent arbitrary identifiers, n to represent an integer literal constant, *string* to represent a string literal constant, and *char* to represent a character literal constant. Using these conventions, the expression typing rules are:

$$\begin{array}{c}
 \overline{\Gamma \vdash n : \text{int}} \quad \overline{\Gamma \vdash \text{true} : \text{bool}} \quad \overline{\Gamma \vdash \text{false} : \text{bool}} \quad \overline{\Gamma \vdash \text{string} : \text{int}[]} \quad \overline{\Gamma \vdash \text{char} : \text{int}} \\
 \\
 \frac{\Gamma(x) = \text{var } t}{\Gamma \vdash x : t} \quad \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \quad \oplus \in \{+, -, *, * >>, /, \%\}}{\Gamma \vdash e_1 \oplus e_2 : \text{int}} \\
 \\
 \frac{\Gamma \vdash e : \text{int}}{\Gamma \vdash -e : \text{int}} \quad \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \quad \ominus \in \{==, !=, <, <=, >, >=\}}{\Gamma \vdash e_1 \ominus e_2 : \text{bool}}
 \end{array}$$

$$\begin{array}{c}
\frac{\Gamma \vdash e : \text{bool}}{\Gamma \vdash !e : \text{bool}} \quad \frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{bool} \quad \ominus \in \{=, \neq, \&, |\}}{\Gamma \vdash e_1 \ominus e_2 : \text{bool}} \\
\\
\frac{\Gamma \vdash e : t[]}{\Gamma \vdash \text{length}(e) : \text{int}} \quad \frac{\Gamma \vdash e_1 : t[] \quad \Gamma \vdash e_2 : t[] \quad \ominus \in \{=, \neq\}}{\Gamma \vdash e_1 \ominus e_2 : \text{bool}} \\
\\
\frac{\Gamma \vdash e_1 : t \quad \dots \quad \Gamma \vdash e_n : t \quad n \geq 0}{\Gamma \vdash \{e_1, \dots, e_n\} : t[]} \quad \frac{\Gamma \vdash e_1 : t[] \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1[e_2] : t} \quad \frac{\Gamma \vdash e_1 : t[] \quad \Gamma \vdash e_2 : t[]}{\Gamma \vdash e_1 + e_2 : t[]} \\
\\
\frac{\Gamma(f) = \text{fn } (t_1, \dots, t_n) \rightarrow (t') \quad \Gamma \vdash e_i : t_i \quad (\forall i \in 1..n) \quad n \geq 0}{\Gamma \vdash f(e_1, \dots, e_n) : t'}
\end{array}$$

3 Type-checking statements

To type-check statements, we need all the information used to type-check expressions, plus the types of procedures, which are included in Γ . In addition, we extend the domain of Γ a little to include a special symbol ρ . To check the return statement we need to know what the return type of the current function is or if it is a procedure. Let this be denoted by $\Gamma(\rho) = \text{ret } T$, where $T \neq ()$ if the statement is part of a function, or $T = ()$ if the statement is part of a procedure. Since statements include declarations, they can also produce new variable bindings, resulting in an updated typing context which we will denote as Γ' . To update typing contexts, we write $\Gamma[x \mapsto \text{var } t]$, which is an environment exactly like Γ except that it maps x to $\text{var } t$. We use the metavariable s to denote a statement, so the main typing judgment for statements has the form $\Gamma \vdash s : R \dashv \Gamma'$.

Most of the statements are fairly straightforward and do not change Γ :

$$\frac{\Gamma \vdash s_1 : \text{unit} \dashv \Gamma_1 \quad \Gamma_1 \vdash s_2 : \text{unit} \dashv \Gamma_2 \quad \dots \quad \Gamma_{n-1} \vdash s_n : R \dashv \Gamma_n}{\Gamma \vdash \{s_1 \ s_2 \ \dots \ s_n\} : R \dashv \Gamma} \text{ (SEQ)} \quad \frac{}{\Gamma \vdash \{\} : \text{unit} \dashv \Gamma} \text{ (EMPTY)}$$

$$\frac{\Gamma \vdash e : \text{bool} \quad \Gamma \vdash s : R \dashv \Gamma'}{\Gamma \vdash \text{if } (e) \ s : \text{unit} \dashv \Gamma} \text{ (IF)} \quad \frac{\Gamma \vdash e : \text{bool} \quad \Gamma \vdash s_1 : R_1 \dashv \Gamma' \quad \Gamma \vdash s_2 : R_2 \dashv \Gamma''}{\Gamma \vdash \text{if } (e) \ s_1 \ \text{else } s_2 : \text{lub}(R_1, R_2) \dashv \Gamma} \text{ (IFELSE)}$$

$$\frac{\Gamma \vdash e : \text{bool} \quad \Gamma \vdash s : R \dashv \Gamma'}{\Gamma \vdash \text{while } (e) \ s : \text{unit} \dashv \Gamma} \text{ (WHILE)}$$

$$\frac{\Gamma(f) = \text{fn } (t_1, \dots, t_n) \rightarrow () \quad \Gamma \vdash e_i : t_i \quad (\forall i \in 1..n) \quad n \geq 0}{\Gamma \vdash f(e_1, \dots, e_n) : \text{unit} \dashv \Gamma} \text{ (PRCALL)}$$

$$\frac{\Gamma(\rho) = \text{ret } (t_1, t_2, \dots, t_n) \quad \Gamma \vdash e_i : t_i \quad (\forall i \in 1..n) \quad n \geq 0}{\Gamma \vdash \text{return } e_1, e_2, \dots, e_n : \text{void} \dashv \Gamma} \text{ (RETURN)}$$

The function lub is defined as follows:

$$\text{lub}(R, R) = R \quad \text{lub}(\text{unit}, R) = \text{lub}(R, \text{unit}) = \text{unit}$$

Therefore, the type of an `if` is `void` only if all branches have that type.

Assignments require checking the left-hand side to make sure it is assignable:

$$\frac{\Gamma(x) = \text{var } t \quad \Gamma \vdash e : t}{\Gamma \vdash x = e : \text{unit} \dashv \Gamma} \text{ (ASSIGN)} \quad \frac{\Gamma \vdash e_1 : t[] \quad \Gamma \vdash e_2 : \text{int} \quad \Gamma \vdash e_3 : t}{\Gamma \vdash e_1[e_2] = e_3 : \text{unit} \dashv \Gamma} \text{ (ARRASSIGN)}$$

Declarations are the source of new bindings. Three kinds of declarations can appear in the source language: variable declarations, multiple assignments, and function/procedure declarations. We are only concerned with the first two kinds within a function body.

$$\frac{x \notin \text{dom}(\Gamma)}{\Gamma \vdash x : t : \text{unit} \dashv \Gamma[x \mapsto \text{var } t]} \text{ (VARDECL)} \quad \frac{x \notin \text{dom}(\Gamma) \quad \Gamma \vdash e : t}{\Gamma \vdash x : t = e : \text{unit} \dashv \Gamma[x \mapsto \text{var } t]} \text{ (VARINIT)}$$

$$\frac{x \notin \text{dom}(\Gamma) \quad \Gamma \vdash e_i : \text{int} \quad (\forall i \in 1..n) \quad n \geq 1 \quad m \geq 0 \quad t \in \{\text{int}, \text{bool}\}}{\Gamma \vdash x : t[e_1] \dots [e_n] \underbrace{[\dots]}_m : \text{unit} \dashv \Gamma[x \mapsto \text{var } t \underbrace{[\dots]}_{n+m}]} \text{ (ARRAYDECL)}$$

With respect to (ARRAYDECL), note that the case of declaring an array with no dimension sizes specified ($n = 0$) is already covered by (VARDECL).

To handle multiple assignments, we define an assignable expression d (destination), which may be a variable declaration:

$$d ::= x : t \mid x \mid e_1[e_2] \mid _$$

We define a new “helper” judgment $\Gamma, \Gamma' \vdash d :: t \dashv \Gamma''$ that determines the type of an assignable expression and also extends the context as necessary. Here, Γ represents the original context before a set of declarations is processed, and Γ' represents the context including the declarations previously brought into scope; Γ'' represents the context after the destination is taken into account, possibly extending Γ' with another binding.

$$\frac{x \notin \text{dom}(\Gamma')}{\Gamma, \Gamma' \vdash x : t :: t \dashv \Gamma'[x \mapsto \text{var } t]} \quad \frac{\Gamma(x) = \text{var } t}{\Gamma, \Gamma' \vdash x :: t \dashv \Gamma'} \quad \frac{}{\Gamma, \Gamma' \vdash _ :: t \dashv \Gamma'} \quad \frac{\Gamma \vdash e_1 : t[] \quad \Gamma \vdash e_2 : \text{int}}{\Gamma, \Gamma' \vdash e_1[e_2] :: t \dashv \Gamma'}$$

Note that in the rule for $_$, any type t can be selected. This makes the type system slightly non-syntax-directed, but a type checker can represent this as a special symbol that can be equated with any possible type.

Using this judgment, we have the following rules for multiple assignment:

$$\frac{\Gamma \vdash e_i : t_i \quad (\forall i \in 1..n) \quad \Gamma_1 = \Gamma \quad \Gamma, \Gamma_i \vdash d_i :: t_i \dashv \Gamma_{i+1} \quad (\forall i \in 1..n)}{\Gamma \vdash d_1, \dots, d_n = e_1, \dots, e_n : \text{unit} \dashv \Gamma_{n+1}} \text{ (MULTIASSIGN)}$$

$$\frac{\Gamma \vdash e_i : t_i \quad (\forall i \in 1..m) \quad \Gamma_1 = \Gamma \quad \Gamma, \Gamma_i \vdash d_i :: t'_i \dashv \Gamma_{i+1} \quad (\forall i \in 1..n) \quad \Gamma(f) = \text{fn } (t_1, \dots, t_m) \rightarrow (t'_1, \dots, t'_n)}{\Gamma \vdash d_1, \dots, d_n = f(e_1, \dots, e_m) : \text{unit} \dashv \Gamma_{n+1}} \text{ (MULTIASSIGNCALL)}$$

These rules actually subsume the earlier (ASSIGN), (ARRASSIGN), and (VARINIT) rules in the case where $n = 1$, so it is redundant to implement those rules directly.

4 Checking top-level declarations

At the top level of the program, we need to figure out the types of procedures and functions, and make sure their bodies are well-typed. Since mutual recursion is supported, this needs to be done in two passes. First, we use the judgment $\Gamma \vdash gd \dashv \Gamma'$ to state that the top-level (global) declaration gd extends top-level bindings Γ to Γ' :

$$\frac{x \notin \Gamma \quad \Gamma' = \Gamma[x \mapsto \text{var } t]}{\Gamma \vdash x : t \dashv \Gamma'} \quad \frac{x \notin \Gamma \quad \Gamma' = \Gamma[x \mapsto \text{var } t]}{\Gamma \vdash x : t = e \dashv \Gamma'}$$

$$\frac{f \notin \text{dom}(\Gamma) \quad \Gamma' = \Gamma[f \mapsto \text{fn } (t_1, \dots, t_n) \rightarrow ()]}{\Gamma \vdash f(x_1 : t_1, \dots, x_n : t_n) s \dashv \Gamma'}$$

$$\frac{f \notin \text{dom}(\Gamma) \quad \Gamma' = \Gamma[f \mapsto \text{fn}(t_1, \dots, t_n) \rightarrow (t'_1, \dots, t'_m)]}{\Gamma \vdash f(x_1:t_1, \dots, x_n:t_n):t'_1, \dots, t'_m \quad s \dashv \Gamma'}$$

The second pass over the program is captured by the judgment $\Gamma \vdash gd \text{ def}$, which defines how to check well-formedness of each global definition against the top-level environment Γ , ensuring that parameters do not shadow anything and that the body is well-typed. We treat procedures just like functions that return no values. The body of a procedure definition may have any type, but the body of a function definition must have type `void`, which ensures that the function body does not fall off the end without returning.

$$\frac{|\text{dom}(\Gamma) \cup \{x_1, \dots, x_n\}| = |\text{dom}(\Gamma)| + n \quad \Gamma[x_1 \mapsto \text{var } t_1, \dots, x_n \mapsto \text{var } t_n, \rho \mapsto \text{ret}(t'_1, \dots, t'_m)] \vdash s : \text{void} \dashv \Gamma'}{\Gamma \vdash f(x_1:t_1, \dots, x_n:t_n):t'_1, \dots, t'_m \quad s \text{ def}}$$

$$\frac{|\text{dom}(\Gamma) \cup \{x_1, \dots, x_n\}| = |\text{dom}(\Gamma)| + n \quad \Gamma[x_1 \mapsto \text{var } t_1, \dots, x_n \mapsto \text{var } t_n, \rho \mapsto \text{ret}()] \vdash s : R \dashv \Gamma'}{\Gamma \vdash f(x_1:t_1, \dots, x_n:t_n) \quad s \text{ def}}$$

$$\frac{}{\Gamma \vdash x : t \text{ def}} \quad \frac{\Gamma \vdash e : t \quad e \text{ is a numeric, boolean, or character literal}}{\Gamma \vdash x : t = e \text{ def}}$$

5 Checking a program

Using the previous judgments, we can define when an entire program $gd_1 gd_2 \dots gd_n$ that does not contain a use declaration is well-formed, written $\vdash gd_1 gd_2 \dots gd_n \text{ prog}$:

$$\frac{\emptyset \vdash gd_1 \dashv \Gamma_1 \quad \Gamma_1 \vdash gd_2 \dashv \Gamma_2 \quad \dots \quad \Gamma_{n-1} \vdash gd_n \dashv \Gamma \quad \Gamma \vdash gd_i \text{ def} \quad (\forall i \in 1..n)}{\vdash gd_1 gd_2 \dots gd_n \text{ prog}}$$

For brevity, the rules for adding declarations appearing in interfaces are omitted. These rules are slightly different from those of the form $\Gamma \vdash gd \dashv \Gamma'$ in Section 4, where $f \notin \text{dom}(\Gamma)$ is replaced with appropriate conditions. Once added, these declarations also permits declarations in the source file of identical signature. See Section 8 of the Eta Language Specification.