CS4120/4121/5120/5121—Spring 2023
Eta Type System Specification
Cornell University
Version of February 27, 2023

0 Changes

- 2/20: Updated rules for checking multiple assignment and simplified handling of function and procedure types.

1 Types

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

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

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 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 $\texttt{var}\ t$ for some type t), return types (bound to $\texttt{ret}\ T$), functions (bound to $\texttt{fn}\ T \rightarrow T'$ where $T' \neq ()$), or procedures (bound to $\texttt{fn}\ T \rightarrow ()$), 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:

$$
\frac{}{\Gamma \vdash n : \texttt{int}} \qquad \frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}} \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}} \qquad \frac{}{\Gamma \vdash \mathit{string} : \texttt{int[]}} \qquad \frac{}{\Gamma \vdash \mathit{char} : \texttt{int}}
$$

$$
\frac{\Gamma(x) = \texttt{var}\ t}{\Gamma \vdash x : t} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int} \quad \oplus \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{*>>}, \texttt{/}, \texttt{\%}\}}{\Gamma \vdash e_1 \oplus e_2 : \texttt{int}}
$$

$$
\frac{\Gamma \vdash e : \texttt{int}}{\Gamma \vdash \texttt{-}e : \texttt{int}} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int} \quad \ominus \in \{\texttt{==}, \texttt{!=}, \texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}\}}{\Gamma \vdash e_1 \ominus e_2 : \texttt{bool}}
$$

$$\frac{\Gamma \vdash e : \texttt{bool}}{\Gamma \vdash \texttt{!}e : \texttt{bool}} \qquad \frac{\Gamma \vdash e_1 : \texttt{bool} \quad \Gamma \vdash e_2 : \texttt{bool} \quad \ominus \in \{\texttt{==}, \texttt{!=}, \texttt{\&}, \texttt{|}\}}{\Gamma \vdash e_1 \ominus e_2 : \texttt{bool}}$$

$$\frac{\Gamma \vdash e : t\texttt{[]}}{\Gamma \vdash \texttt{length}(e) : \texttt{int}} \qquad \frac{\Gamma \vdash e_1 : t\texttt{[]} \quad \Gamma \vdash e_2 : t\texttt{[]} \quad \ominus \in \{\texttt{==}, \texttt{!=}\}}{\Gamma \vdash e_1 \ominus e_2 : \texttt{bool}}$$

$$\frac{\Gamma \vdash e_1 : t \quad \ldots \quad \Gamma \vdash e_n : t \quad n \geq 0}{\Gamma \vdash \texttt{\{}e_1\texttt{,} \ldots \texttt{,}e_n\texttt{\}} : t\texttt{[]}} \qquad \frac{\Gamma \vdash e_1 : t\texttt{[]} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1\texttt{[}e_2\texttt{]} : t} \qquad \frac{\Gamma \vdash e_1 : t\texttt{[]} \quad \Gamma \vdash e_2 : t\texttt{[]}}{\Gamma \vdash e_1 \texttt{ + } e_2 : t\texttt{[]}}$$

$$\frac{\Gamma(f) = \texttt{fn}\,(t_1, \ldots, t_n) \to (t') \quad \Gamma \vdash e_i : t_i \;\; {}^{(\forall i \in 1..n)} \quad n \geq 0}{\Gamma \vdash f\texttt{(}e_1\texttt{,} \ldots \texttt{,}e_n\texttt{)} : t'}$$

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 $\Gamma(\rho) = \texttt{ret}\ T$, 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 $\Gamma[x \mapsto \texttt{var}\ t]$, which is an environment exactly like Γ except that it maps x to $\texttt{var}\ t$. 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:

$$\frac{\Gamma \vdash s_1 : \texttt{unit} \dashv \Gamma_1 \quad \Gamma_1 \vdash s_2 : \texttt{unit} \dashv \Gamma_2 \quad \ldots \quad \Gamma_{n-1} \vdash s_n : R \dashv \Gamma_n}{\Gamma \vdash \texttt{\{}s_1\ s_2\ \ldots\ s_n\texttt{\}} : R \dashv \Gamma}\ (\textsc{Seq}) \qquad \frac{}{\Gamma \vdash \texttt{\{\}} : \texttt{unit} \dashv \Gamma}\ (\textsc{Empty})$$

$$\frac{\Gamma \vdash e : \texttt{bool} \quad \Gamma \vdash s : R \dashv \Gamma'}{\Gamma \vdash \texttt{if (}e\texttt{)}\ s : \texttt{unit} \dashv \Gamma}\ (\textsc{If}) \qquad \frac{\Gamma \vdash e : \texttt{bool} \quad \Gamma \vdash s_1 : R_1 \dashv \Gamma' \quad \Gamma \vdash s_2 : R_2 \dashv \Gamma''}{\Gamma \vdash \texttt{if (}e\texttt{)}\ s_1\ \texttt{else}\ s_2 : \mathit{lub}(R_1, R_2) \dashv \Gamma}\ (\textsc{IfElse})$$

$$\frac{\Gamma \vdash e : \texttt{bool} \quad \Gamma \vdash s : R \dashv \Gamma'}{\Gamma \vdash \texttt{while (}e\texttt{)}\ s : \texttt{unit} \dashv \Gamma}\ (\textsc{While})$$

$$\frac{\Gamma(f) = \texttt{fn}\,(t_1, \ldots, t_n) \to () \quad \Gamma \vdash e_i : t_i \;\; {}^{(\forall i \in 1..n)} \quad n \geq 0}{\Gamma \vdash f\texttt{(}e_1\texttt{,} \ldots \texttt{,}e_n\texttt{)} : \texttt{unit} \dashv \Gamma}\ (\textsc{PrCall})$$

$$\frac{\Gamma(\rho) = \texttt{ret}\,(t_1, t_2, \ldots, t_n) \quad \Gamma \vdash e_i : t_i \;\; {}^{(\forall i \in 1..n)} \quad n \geq 0}{\Gamma \vdash \texttt{return}\ e_1\texttt{,}\ e_2\texttt{,}\ \ldots\texttt{,}\ e_n : \texttt{void} \dashv \Gamma}\ (\textsc{Return})$$

The function *lub* is defined as follows:

$$\mathit{lub}(R, R) = R \qquad \mathit{lub}(\texttt{unit}, R) = \mathit{lub}(R, \texttt{unit}) = \texttt{unit}$$

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:

$$\frac{\Gamma(x) = \texttt{var}\ t \quad \Gamma \vdash e : t}{\Gamma \vdash x\ \texttt{=}\ e : \texttt{unit} \dashv \Gamma}\ (\textsc{Assign}) \qquad \frac{\Gamma \vdash e_1 : t\texttt{[]} \quad \Gamma \vdash e_2 : \texttt{int} \quad \Gamma \vdash e_3 : t}{\Gamma \vdash e_1\texttt{[}e_2\texttt{]}\ \texttt{=}\ e_3 : \texttt{unit} \dashv \Gamma}\ (\textsc{ArrAssign})$$

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.

$$\frac{x \notin \text{dom}(\Gamma)}{\Gamma \vdash x\texttt{:}t : \texttt{unit} \dashv \Gamma[x \mapsto \texttt{var}\ t]}\ (\textsc{VarDecl}) \qquad \frac{x \notin \text{dom}(\Gamma) \quad \Gamma \vdash e : t}{\Gamma \vdash x\texttt{:}t\ \texttt{=}\ e : \texttt{unit} \dashv \Gamma[x \mapsto \texttt{var}\ t]}\ (\textsc{VarInit})$$

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

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:

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

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.

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

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:

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

$$\frac{\begin{array}{c}\Gamma \vdash e_i : t_i\ ^{(\forall i \in 1..m)} \quad \Gamma_1 = \Gamma \quad \Gamma, \Gamma_i \vdash d_i :: t'_i \dashv \Gamma_{i+1}\ ^{(\forall i \in 1..n)} \\ \Gamma(f) = \texttt{fn}\ (t_1, \dots, t_m) \to\ (t'_1, \dots, t'_n)\end{array}}{\Gamma \vdash d_1\texttt{,} \dots \texttt{,} d_n = f\texttt{(}e_1\texttt{,} \dots \texttt{,} e_m\texttt{)} : \texttt{unit} \dashv \Gamma_{n+1}}\ (\textsc{MultiAssignCall})$$

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':

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

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

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

The second pass over the program is captured by the judgment $\Gamma \vdash gd\ \ \texttt{def}$, 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.

$$\frac{\begin{array}{c}|\mathrm{dom}(\Gamma) \cup \{x_1,\ldots,x_n\}| = |\mathrm{dom}(\Gamma)| + n \\ \Gamma[x_1 \mapsto \texttt{var}\ t_1,\ldots,x_n \mapsto \texttt{var}\ t_n, \rho \mapsto \texttt{ret}\ (t'_1,\ldots,t'_m)] \vdash s : \texttt{void} \dashv \Gamma'\end{array}}{\Gamma \vdash f\texttt{(}x_1\texttt{:}t_1\texttt{,}\ \ldots\texttt{,}x_n\texttt{:}t_n\texttt{)}\texttt{:}t'_1\texttt{,}\ \ldots\texttt{,}t'_m\ \ s\ \ \texttt{def}}$$

$$\frac{\begin{array}{c}|\mathrm{dom}(\Gamma) \cup \{x_1,\ldots,x_n\}| = |\mathrm{dom}(\Gamma)| + n \\ \Gamma[x_1 \mapsto \texttt{var}\ t_1,\ldots,x_n \mapsto \texttt{var}\ t_n, \rho \mapsto \texttt{ret}\ ()] \vdash s : R \dashv \Gamma'\end{array}}{\Gamma \vdash f\texttt{(}x_1\texttt{:}t_1\texttt{,}\ \ldots\texttt{,}x_n\texttt{:}t_n\texttt{)}\ \ s\ \ \texttt{def}}$$

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

5 Checking a program

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

$$\frac{\varnothing \vdash gd_1 \dashv \Gamma_1 \quad \Gamma_1 \vdash gd_2 \dashv \Gamma_2 \quad \ldots \quad \Gamma_{n-1} \vdash gd_n \dashv \Gamma \quad \Gamma \vdash gd_i\ \ \texttt{def}^{\ (\forall i \in 1..n)}}{\vdash gd_1\ gd_2\ \ldots\ gd_n\ \ \texttt{prog}}$$

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 $f \notin \mathrm{dom}(\Gamma)$ 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.