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:

\[
\begin{align*}
t &::= \text{int} \mid \text{bool} \mid t[n] \\
T &::= (t_1, t_2, \ldots, t_n)^{n \geq 0} \mid \text{unit} \mid \text{void} \mid \text{ret } T \\
\sigma &::= \text{var } t \mid \text{ret } T \mid \text{fn } T \rightarrow T'
\end{align*}
\]

Ordinary types expressible in the language are denoted by the metavariable \( t \), which can be \text{int}, \text{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 \text{unit} or \text{void}. The \text{unit} type is the the type of statements that might complete normally and permit the following statement to execute. The type \text{void} is the type of statements such as \text{return} that never pass control to the following statement. The \text{void} type should not be confused with the C/Java type \text{void}, which is actually closer to \text{unit}.

The set \( \sigma \) is used to represent typing-environment entries, which can either be normal variables (bound to \text{var } t for some type \( t \)), return types (bound to \text{ret } T), functions (bound to \text{fn } T \rightarrow T' where \( T' \neq () \)), or procedures (bound to \text{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 \( \Gamma \), which maps names \( x \) to types \( \sigma \).

The judgment \( \Gamma \vdash e : t \) is the rule for the type of an expression; it states that with bindings \( \Gamma \) 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, \( \text{string} \) to represent a string literal constant, and \( \text{char} \) to represent a character literal constant. Using these conventions, the expression typing rules are:

\[
\begin{align*}
\Gamma \vdash n : \text{int} & \quad \Gamma \vdash \text{true} : \text{bool} & \quad \Gamma \vdash \text{false} : \text{bool} & \quad \Gamma \vdash \text{string} : \text{int[]} & \quad \Gamma \vdash \text{char} : \text{int} \\
\Gamma(x) = \text{var } t & \quad \Gamma \vdash e_1 : \text{int} & \quad \Gamma \vdash e_2 : \text{int} & \quad \Theta \in \{+,-,*,\gg,/,\%\} & \quad \Gamma \vdash e_1 \Theta e_2 : \text{int} \\
\Gamma \vdash e : \text{int} & \quad \Gamma \vdash e_1 : \text{int} & \quad \Gamma \vdash e_2 : \text{int} & \quad \Theta \in \{==,!=,<,<=,>,>=\} & \quad \Gamma \vdash e_1 \Theta e_2 : \text{bool} \\
\Gamma \vdash -e : \text{int} & \quad \Gamma \vdash e_1 : \text{int} & \quad \Gamma \vdash e_2 : \text{int} & \quad \Theta \in \{+,-,*,\gg,/,\%\} & \quad \Gamma \vdash e_1 \Theta e_2 : \text{int}
\end{align*}
\]
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 \( \Gamma \). In addition, we extend the domain of \( \Gamma \) a little to include a special symbol \( \rho \). 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 \( \Gamma' \). To update typing contexts, we write \( \Gamma[x \to \text{var } t] \), which is an environment exactly like \( \Gamma \) 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 \( \Gamma \vdash s : R \Rightarrow \Gamma' \).

Most of the statements are fairly straightforward and do not change \( \Gamma \):

\[
\frac{\Gamma \vdash e : \text{bool} \quad \Gamma \vdash s : R \Rightarrow \Gamma'}{\Gamma \vdash \text{if } (e) \ s : \text{unit } \Rightarrow \Gamma'} \quad \frac{\Gamma \vdash e : \text{bool} \quad \Gamma \vdash s_1 : R \Rightarrow \Gamma' \quad \Gamma \vdash s_2 : \text{unit } \Rightarrow \Gamma'}{\Gamma \vdash \text{if } (e) \ s_1 \text{ else } s_2 : \text{lub}(R_1, R_2) \Rightarrow \Gamma'} \quad \frac{\Gamma \vdash e : \text{bool}}{\Gamma \vdash \text{while } (e) \ s : \text{unit } \Rightarrow \Gamma'} \quad \frac{\Gamma(\rho) = \text{fn } (t_1, \ldots, t_n) \to (t') \quad \Gamma \vdash e_i : t_i \ (\forall i \in 1..n) \quad n \geq 0}{\Gamma \vdash \text{return } e_1, e_2, \ldots, e_n : \text{void } \Rightarrow \Gamma'}
\]

The function \( \text{lub} \) 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 \text{if} is \text{void} only if all branches have that type.

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

\[
\frac{\Gamma(\rho) = \text{var } t \quad \Gamma \vdash e : t}{\Gamma \vdash x = e : \text{unit } \Rightarrow \Gamma} \quad \frac{\Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : \text{int} \quad \Gamma \vdash e_3 : t}{\Gamma \vdash e_1[e_2] = e_3 : \text{unit } \Rightarrow \Gamma}
\]

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.
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:

\[
\begin{align*}
\frac{x \not\in \text{dom}(\Gamma)}{\Gamma \mid x : \text{unit} \rightarrow \text{var} t} & \quad \text{(VARDECL)} \\
\frac{x \not\in \text{dom}(\Gamma)}{\Gamma \mid x = e : \text{unit} \rightarrow \text{var} t} & \quad \text{(VARINIT)}
\end{align*}
\]

\[
\frac{x \not\in \text{dom}(\Gamma)}{\Gamma \mid e_1 : \text{int}} \quad \frac{\forall i \in \{1..n\}}{\frac{n \geq 1}{\frac{\Gamma \mid e_1 \ldots e_n : \text{unit} \rightarrow \text{var} t \ldots t}{\text{ARRAYDECL} (n+m)}}}
\]

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

\[
\frac{x \not\in \text{dom}(\Gamma)}{\Gamma, \Gamma' \mid x : t \rightarrow \Gamma'[x \rightarrow \text{var} t]} \quad \frac{\Gamma(x) = \text{var} t}{\Gamma, \Gamma' \mid x : t \rightarrow \Gamma'} \quad \frac{\Gamma' \mid e_1 : t}{} \quad \frac{\Gamma \mid e_2 : \text{int}}{\Gamma, \Gamma' \mid e_2 : t} \quad \Gamma' \rightarrow t \rightarrow \Gamma' \quad \frac{\Gamma \mid e_1 : t}{} \quad \frac{\Gamma \mid e_2 : \text{int}}{\Gamma, \Gamma' \mid e_2 : t} \quad \frac{\Gamma \mid e_2 : \text{int}}{\Gamma, \Gamma' \mid e_2 : t} \quad \Gamma' \rightarrow t \rightarrow \Gamma' \]

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:

\[
\frac{\Gamma \mid e_i : t_i \ (\forall i \in \{1..n\}) \quad \Gamma_1 = \Gamma \quad \Gamma, \Gamma_i \mid d_i : t_i \rightarrow \Gamma_i+1 \ (\forall i \in \{1..n\})}{\Gamma \mid d_1, \ldots, d_n = e_1, \ldots, e_n : \text{unit} \rightarrow \Gamma_{n+1}} \quad \frac{\Gamma(f) = \text{fn} (t_1, \ldots, t_m) \rightarrow (t_1', \ldots, t_m')}{\Gamma \mid d_1, \ldots, d_n = f(e_1, \ldots, e_m) : \text{unit} \rightarrow \Gamma_{n+1}} \quad \frac{\Gamma \mid e_i : t_i \ (\forall i \in \{1..m\}) \quad \Gamma_1 = \Gamma \quad \Gamma, \Gamma_i \mid d_i : t_i \rightarrow \Gamma_i+1 \ (\forall i \in \{1..n\})}{\Gamma \mid d_1, \ldots, d_n = f(e_1, \ldots, e_m) : \text{unit} \rightarrow \Gamma_{n+1}} \quad \frac{\Gamma \mid e_i : t_i \ (\forall i \in \{1..m\}) \quad \Gamma_1 = \Gamma \quad \Gamma, \Gamma_i \mid d_i : t_i \rightarrow \Gamma_i+1 \ (\forall i \in \{1..n\})}{\Gamma \mid d_1, \ldots, d_n = f(e_1, \ldots, e_m) : \text{unit} \rightarrow \Gamma_{n+1}}}
\]

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 \(\Gamma \mid gd \rightarrow \Gamma'\) to state that the top-level (global) declaration \(gd\) extends top-level bindings \(\Gamma\) to \(\Gamma'\):

\[
\frac{x \not\in \Gamma \quad \Gamma' = \Gamma[x \rightarrow \text{var} t]}{\Gamma \mid x : t \rightarrow \Gamma'} \quad \frac{x \not\in \Gamma \quad \Gamma' = \Gamma[x \rightarrow \text{var} t]}{\Gamma \mid x : t = e \rightarrow \Gamma'} \quad \frac{f \not\in \text{dom}(\Gamma) \quad \Gamma' = \Gamma[f \rightarrow \text{fn} (t_1, \ldots, t_n) \rightarrow ()]}{\Gamma \mid f(x_1 : t_1, \ldots, x_n : t_n) \rightarrow s \rightarrow \Gamma'}
\]
f ∉ dom(Γ) \quad Γ' = Γ[ f \rightarrow \text{fn}(t_1, \ldots, t_n) \rightarrow (t'_1, \ldots, t'_m)]
\quad Γ \vdash f(x_1 : t_1, \ldots, x_n : t_n) : t'_1, \ldots, t'_m \quad s \vdash Γ'

The second pass over the program is captured by the judgment \(Γ \vdash gd\ \text{def,f} \), 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 \text{void}, which ensures that the function body does not fall off the end without returning.

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

\[|\text{dom}(Γ) \cup \{x_1, \ldots, x_n\}| = |\text{dom}(Γ)| + n\]
\[Γ[x_1 \mapsto \text{var}\ t_1, \ldots, x_n \mapsto \text{var}\ t_n, \rho \mapsto \text{ret}\ \{\}\] \vdash s : R \vdash Γ'\]
\[Γ \vdash f(x_1 : t_1, \ldots, x_n : t_n) \quad s \quad \text{def}\]
\[Γ \vdash x : t \quad \text{def} \quad \Gamma \vdash x : t \quad e \text{ is a numeric, boolean, or character literal} \quad Γ \vdash x : t \quad e \text{ def}\]

### 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 \(\vdash gd_1 \ gd_2 \ldots \ gd_n \text{ prog}\):

\[\emptyset \vdash gd_1 \vdash Γ_1 \quad Γ_1 \vdash gd_2 \vdash Γ_2 \quad \ldots \quad Γ_{n-1} \vdash gd_n \vdash Γ \quad Γ \vdash gd_1 \text{ def } (\forall i \in 1..n) \quad \vdash gd_1 \ gd_2 \ldots \ gd_n \text{ prog}\]

For brevity, the rules for adding declarations appearing in interfaces are omitted. These rules are slightly different from those of the form \(Γ \vdash gd \vdash Γ'\) in Section 4, where \(f \notin \text{dom}(Γ)\) 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.