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:

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

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:

\[
\begin{align*}
  \Gamma \vdash n : \text{int} &\quad \Gamma \vdash \text{true} : \text{bool} &\quad \Gamma \vdash \text{false} : \text{bool} &\quad \Gamma \vdash \text{string} : \text{int}[\ ] &\quad \Gamma \vdash \text{char} : \text{int} \\
  \Gamma(x) = \text{var}\ t &\quad \Gamma \vdash e_1 : \text{int} &\quad \Gamma \vdash e_2 : \text{int} &\quad \Theta \in \{+,-,\ast,\ast\ast,\ast\ast\ast,\ast\ast\ast\ast,\%/\}
  &\quad \Gamma \vdash e_1 \Theta e_2 : \text{int} \\
  \Gamma \vdash e : \text{int} &\quad \Gamma \vdash e_1 : \text{int} &\quad \Gamma \vdash e_2 : \text{int} &\quad \Theta \in \{==,\neq,\lt,\leq,\gt,\geq\}
  &\quad \Gamma \vdash e_1 \Theta e_2 : \text{bool}
\end{align*}
\]
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 $\Gamma[x \rightarrow \text{var } t]$, which is an environment exactly like $\Gamma$ except that it maps $x$ to 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 $\Gamma$:

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

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

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

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

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

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 if is void only if all branches have that type.

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

$$\frac{\Gamma(x) = \text{var } t \quad \Gamma \vdash e : t}{\Gamma \vdash x = e : \text{unit} \rightarrow \Gamma} \quad (\text{ASSIGN})$$

$$\frac{\Gamma \vdash e_1 : t[\ ] \quad \Gamma \vdash e_2 : \text{int} \quad \Gamma \vdash e_3 : t}{\Gamma \vdash e_1[e_2] = e_3 : \text{unit} \rightarrow \Gamma} \quad (\text{ARRASSIGN})$$

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.
We define a new "helper" judgment

\[ \Gamma : t \Rightarrow t' \]

With respect to \( (A \Gamma) \) we use the judgment

\[ \Gamma \vdash t : t' \]

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, \( 4 \) Checking top-level declarations

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

\[ \Gamma : t \Rightarrow t' \]

\[ \Gamma \vdash x : t \]

\[ \Gamma : t \Rightarrow t' \]

\[ \Gamma \vdash x : t \]

\[ \Gamma : t \Rightarrow t' \]

\[ \Gamma \vdash x : t \]

\[ \Gamma : t \Rightarrow t' \]

\[ \Gamma \vdash x : t \]

\[ \Gamma : t \Rightarrow t' \]

\[ \Gamma \vdash x : t \]

\[ \Gamma : t \Rightarrow t' \]

\[ \Gamma \vdash x : t \]

\[ \Gamma : t \Rightarrow t' \]

\[ \Gamma \vdash x : t \]

\[ \Gamma : t \Rightarrow t' \]

\[ \Gamma \vdash x : t \]

\[ \Gamma : t \Rightarrow t' \]

\[ \Gamma \vdash x : t \]

\[ \Gamma : t \Rightarrow t' \]

\[ \Gamma \vdash x : t \]

\[ \Gamma : t \Rightarrow t' \]

\[ \Gamma \vdash x : t \]

\[ \Gamma : t \Rightarrow t' \]

\[ \Gamma \vdash x : t \]

\[ \Gamma : t \Rightarrow t' \]

\[ \Gamma \vdash x : t \]

\[ \Gamma : t \Rightarrow t' \]

\[ \Gamma \vdash x : t \]

\[ \Gamma : t \Rightarrow t' \]

\[ \Gamma \vdash x : t \]

\[ \Gamma : t \Rightarrow t' \]

\[ \Gamma \vdash x : t \]

\[ \Gamma : t \Rightarrow t' \]

\[ \Gamma \vdash x : t \]

\[ \Gamma : t \Rightarrow t' \]

\[ \Gamma \vdash x : t \]

\[ \Gamma : t \Rightarrow t' \]

\[ \Gamma \vdash x : t \]

\[ \Gamma : t \Rightarrow t' \]

\[ \Gamma \vdash x : t \]

\[ \Gamma : t \Rightarrow t' \]

\[ \Gamma \vdash x : t \]

\[ \Gamma : t \Rightarrow t' \]

\[ \Gamma \vdash x : t \]

\[ \Gamma : t \Rightarrow t' \]

\[ \Gamma \vdash x : t \]

\[ \Gamma : t \Rightarrow t' \]

\[ \Gamma \vdash x : t \]

\[ \Gamma : t \Rightarrow t' \]

\[ \Gamma \vdash x : t \]

\[ \Gamma : t \Rightarrow t' \]

\[ \Gamma \vdash x : t \]

\[ \Gamma : t \Rightarrow t' \]

\[ \Gamma \vdash x : t \]

\[ \Gamma : t \Rightarrow t' \]

\[ \Gamma \vdash x : t \]

\[ \Gamma : t \Rightarrow t' \]

\[ \Gamma \vdash x : t \]

\[ \Gamma : t \Rightarrow t' \]
The second pass over the program is captured by the judgment $\Gamma \vdash gd \ def$, which defines how to check well-formedness of each global definition against the top-level environment $\Gamma$, 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.

$$|\text{dom}(\Gamma) \cup \{x_1, \ldots, x_n\}| = |\text{dom}(\Gamma)| + n$$

$$\Gamma[\{x_1 \mapsto \text{var } t_1, \ldots, x_n \mapsto \text{var } t_n, \rho \mapsto \text{ret } (t'_1, \ldots, t'_m)\}] \vdash s : \text{void} \land \Gamma'$$

$$\Gamma \vdash f(x_1:t_1, \ldots, x_n:t_n): t'_1, \ldots, t'_m \ def$$

$$|\text{dom}(\Gamma) \cup \{x_1, \ldots, x_n\}| = |\text{dom}(\Gamma)| + n$$

$$\Gamma[\{x_1 \mapsto \text{var } t_1, \ldots, x_n \mapsto \text{var } t_n, \rho \mapsto \text{ret } ()\}] \vdash s : R \land \Gamma'$$

$$\Gamma \vdash f(x_1:t_1, \ldots, x_n:t_n) \ s \ def$$

$$\Gamma \vdash x : t \ def$$

$\Gamma \vdash x : t \ e \ is \ a \ numeric, \ boolean, \ or \ character \ literal$\n
$$\Gamma \vdash x : t = e \ def$$

### 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 $\vdash gd_1 gd_2 \ldots gd_n \ prog$:

$$\emptyset \vdash gd_1 \Gamma_1 \quad \Gamma_1 \vdash gd_2 \Gamma_2 \quad \ldots \quad \Gamma_n-1 \vdash gd_n \Gamma \quad \Gamma \vdash gd_j \ (\forall i \in 1..n)$$

$$\vdash gd_1 gd_2 \ldots gd_n \ prog$$

For brevity, the rules for adding declarations appearing in interfaces are omitted. These rules are slightly different from those of the form $\Gamma \vdash gd \land \Gamma'$ 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.