0 Changes

• None yet; watch this space.

1 Types

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

\[
\begin{align*}
\tau ::= & \text{int} & T ::= & \tau & R ::= & \text{unit} & \sigma ::= & \text{var } \tau \\
& \text{bool} & & \text{unit} & & \text{void} & & \text{fn } T \to T' \\
& \tau[] & & (\tau_1, \tau_2, \ldots, \tau_n) & n \geq 2
\end{align*}
\]

Ordinary types expressible in the language are denoted by the metavariable \( \tau \), which can be \text{int, bool, or an array type.}

The metavariable \( T \) denotes an expanded notion of type that represents possible types in procedures, functions, and multiple assignments. It may be an ordinary type, \text{unit}, or a tuple type.

The \text{unit} type is how we will write the unit type, a type that does not appear explicitly in the source language. The \text{unit} type is given to an element on the left-hand side of multiple assignments that uses the \_ placeholder, which lets their handling be integrated directly into the type system. The \text{unit} type is also used to represent the result type of procedures.

Tuple types represent the parameter types of procedures and functions that take multiple arguments, or the return types of functions that return multiple results.

The metavariable \( R \) represent the outcome of evaluating a statement, which can be either \text{unit or void.} Statements of type \text{unit do not always interrupt the flow of execution; statements of type \text{void do.}}

The set \( \sigma \) is used to represent typing environment entries, which can either be normal variables (bound to \text{var } \tau for some type \( \tau \)), functions (bound to \text{fn } T \to T' where \( T' \neq \text{unit} \)), or procedures (bound to \text{fn } T \to \text{unit} \), where the “result type” (\text{unit}) indicates that the procedure result contains no information other than that the procedure call terminated.

2 Subtyping

The subtyping relation on \( T \) is the least partial order consistent with this rule:

\[
\tau \leq \text{unit}
\]

For now, however, there is no subsumption rule, so subtyping matters only where it appears explicitly.

3 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, \( string \) to represent a string literal constant, and \( char \) to represent a character literal constant. Using these conventions, the expression typing rules are:

\[
\begin{align*}
\Gamma \vdash n : \text{int} & \quad \Gamma \vdash \text{true} : \text{bool} & \quad \Gamma \vdash \text{false} : \text{bool} & \quad \Gamma \vdash \text{string} : \text{int}[\] \\
\Gamma(x) = \text{var} \tau & \quad \Gamma \vdash e_1 : \text{int} & \quad \Gamma \vdash e_2 : \text{int} & \quad \tau \in \{+,-,*,**,/,\%\} \\
\Gamma \vdash e : \text{int} & \quad \Gamma \vdash -e : \text{int} & \quad \Gamma \vdash e_1 \oplus e_2 : \text{int} \\
\Gamma \vdash e : \text{bool} & \quad \Gamma \vdash \text{true} \oplus \text{false} : \text{bool} \\
\Gamma \vdash e : \text{bool} & \quad \Gamma \vdash \text{true} \oplus \text{false} : \text{bool} & \quad \tau \in \{==,!=,\langle,\rangle\} \\
\Gamma \vdash e_1 : \tau[\] & \quad \Gamma \vdash e_2 : \tau[\] & \quad \tau \in \{==,!=\} \\
\Gamma \vdash \text{length}(e) : \text{int} & \quad \Gamma \vdash e_1 \oplus e_2 : \text{bool} \\
\Gamma \vdash e_1 : \tau \ldots \Gamma \vdash e_n : \tau & n \geq 0 & \quad \Gamma \vdash \{e_1, \ldots, e_n\} : \tau[\] \\
\Gamma \vdash e_1[\tau_1] \ldots e_n[\tau_n] : \tau[\] & \quad \Gamma \vdash e_1[\tau_1] \ldots e_n[\tau_n] : \tau[\] \\
\Gamma(f) = \text{fn} \text{unit} \to T' & T' \neq \text{unit} & \quad \Gamma \vdash f : T' \\
\Gamma(f) = \text{fn} \tau \to T' & T' \neq \text{unit} & \quad \Gamma \vdash e : \tau \\
\Gamma(f) = \text{fn} (\tau_1, \ldots, \tau_n) \to T' & T' \neq \text{unit} & \quad \Gamma \vdash e_i \tau \text{ for } i \in 1..n & n \geq 2 \\
\Gamma \vdash f(e_1, \ldots, e_n) : T'
\end{align*}
\]

4 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 \( \Gamma(\rho) \), which is some type \( T \neq \text{unit} \) if the statement is part of a function, or \( \text{unit} \) 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 \( \Gamma[x \mapsto \tau] \), which is an environment exactly like \( \Gamma \) except that it maps \( x \) to \( \tau \). We use the metavariable \( s \) to denote a statement, so the main typing judgment for statements has the form \( \Gamma \vdash s : R, \Gamma' \).

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

\[
\begin{align*}
\Gamma \vdash s_1 : \text{unit}, \Gamma_1 & \quad \Gamma_1 \vdash s_2 : \text{unit}, \Gamma_2 & \quad \ldots & \quad \Gamma_{n-1} \vdash s_n : R, \Gamma_n & \quad \Gamma \vdash \{s_1 \ldots s_n\} : R, \Gamma \quad (\text{SEQ}) \\
\Gamma \vdash e : \text{bool} & \quad \Gamma_1 \vdash s : R, \Gamma' & \quad \Gamma \vdash s_1 : R, \Gamma' & \quad \Gamma \vdash s_2 : R, \Gamma'' \quad (\text{IFELSE})
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash \text{if } (e) \text{ s1 } & \quad \text{else } s_2 \text{ : lub}(R_1, R_2) \quad (\text{IF})
\end{align*}
\]
The function $lub$ is defined as follows:

$$lub(R, R) = R \quad lub(unit, R) = lub(R, unit) = unit$$

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:

$$\Gamma \vdash x : \tau : \text{unit}, \Gamma$$ (ASSIGN)

$$\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \text{int} \quad \Gamma \vdash e_3 : \tau$$

The following rules:

$$\Gamma \vdash \text{RETURN} : \text{void}, \Gamma$$ (RETURN)

$$\Gamma \vdash \text{RETURN} e : \text{void}, \Gamma$$ (RETVAL)

The final premise in rule MULTIASSIGN prevents shadowing by ensuring that $\text{dom}(\Gamma)$ and all of the $\text{varsof}(d_i)$’s are disjoint from each other.
5 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 $\Gamma \vdash fd : \Gamma'$ to state that the function or procedure declaration $fd$ extends top-level bindings $\Gamma$ to $\Gamma'$:

$$
\begin{align*}
\text{if } f \notin \text{dom}(\Gamma) \\
\Gamma \vdash f \circ s : \Gamma[f \mapsto \text{fn } \tau \mapsto \text{unit}] \\
\text{then } f \notin \text{dom}(\Gamma) \\
\Gamma \vdash f(x : \tau) \ s : \Gamma[f \mapsto \text{fn } \tau \mapsto \text{unit}] \\
\text{else } f \notin \text{dom}(\Gamma) \\
\Gamma \vdash f(\circ : \tau') \ s : \Gamma[f \mapsto \text{fn } \tau \mapsto \tau'] \\
\text{else } f \notin \text{dom}(\Gamma) \\
\Gamma \vdash f(x : \tau) : \tau' \ s : \Gamma[f \mapsto \text{fn } \tau \mapsto \tau'] \\
\text{end}
\end{align*}
$$

The second pass over the program is captured by the judgment $\Gamma \vdash f \circ s \ s \ \text{def}$, which defines how to check well-formedness of each function definition against a 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 the unit type. 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.

$$
\begin{align*}
\text{if } f \notin \text{dom}(\Gamma) \\
\Gamma[\rho \mapsto \text{unit}] \vdash s : R, \Gamma' \\
\Gamma \vdash f(\circ : \tau') \ s \ \text{def} \\
\text{end}
\end{align*}
$$

$$
\begin{align*}
\text{if } x \notin \text{dom}(\Gamma) \\
\Gamma[x \mapsto \tau_1, \rho \mapsto \text{unit}] \vdash s : R, \Gamma' \\
\Gamma \vdash f(x : \tau) \ s \ \text{def} \\
\text{else } x \notin \text{dom}(\Gamma) \\
\Gamma[\rho \mapsto \tau'] \vdash s : \text{void}, \Gamma' \\
\Gamma \vdash f(\circ : \tau') \ s \ \text{def} \\
\text{end}
\end{align*}
$$

$$
\begin{align*}
|\text{dom}(\Gamma) \cup \{x_1, \ldots, x_n\}| = |\text{dom}(\Gamma)| + n \quad n \geq 2 \\
\Gamma[x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n, \rho \mapsto \text{unit}] \vdash s : R, \Gamma' \\
\Gamma \vdash f(x_1 : \tau_1, \ldots, x_n : \tau_n) \ s \ \text{def}
\end{align*}
$$
\[ |\text{dom}(\Gamma) \cup \{x_1, \ldots, x_n\}| = |\text{dom}(\Gamma)| + n \quad n \geq 2 \]
\[ \Gamma[x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n, \rho \mapsto \tau]' \vdash s : \text{void}, \Gamma' \]
\[ \Gamma \vdash f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau' \quad s \text{ def} \]
\[ \Gamma[\rho \mapsto (\tau'_1, \ldots, \tau'_m)] \vdash s : \text{void}, \Gamma' \quad m \geq 2 \]
\[ \Gamma \vdash f() : \tau'_1, \ldots, \tau'_m \quad s \text{ def} \]
\[ x \notin \text{dom}(\Gamma) \quad \Gamma[x \mapsto \tau, \rho \mapsto (\tau'_1, \ldots, \tau'_m)] \vdash s : \text{void}, \Gamma' \quad m \geq 2 \]
\[ \Gamma \vdash f(x : \tau) : \tau'_1, \ldots, \tau'_m \quad s \text{ def} \]
\[ |\text{dom}(\Gamma) \cup \{x_1, \ldots, x_n\}| = |\text{dom}(\Gamma)| + n \quad n \geq 2 \quad m \geq 2 \]
\[ \Gamma[x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n, \rho \mapsto (\tau'_1, \ldots, \tau'_m)] \vdash s : \text{void}, \Gamma' \]
\[ \Gamma \vdash f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau'_1, \ldots, \tau'_m \quad s \text{ def} \]

6 Checking a program

Using the previous judgments, we can define when an entire program \(f_1 \ f_2 \ldots \ f_n\) is well-formed, written \(\vdash f_1 \ f_2 \ldots \ f_n \ \text{prog}\):

\[ \emptyset \vdash f_1 : \Gamma_1 \quad \Gamma_1 \vdash f_2 : \Gamma_2 \quad \ldots \quad \Gamma_{n-1} \vdash f_n : \Gamma \quad \Gamma \vdash f_i \text{ def } (\forall i \in 1..n) \]
\[ \vdash f_1 \ f_2 \ldots \ f_n \ \text{prog} \]