CS4120/4121/5120/5121—Spring 2016
Xi Type System Specification
Cornell University
Version of March 16, 2016

0 Changes

- 3/3: Corrected typing rules that involves adding a new binding to the typing context. Headers such as `var` were missing in the new binding.
- 2/27: Fixed IF and WHILE typing rules. A little more clarifying text added about unit vs. void.
- 2/26: Clarified rules for `use` declarations.

1 Types

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

$$\begin{array}{llll} \tau ::= \texttt{int} & T ::= \tau & R ::= \texttt{unit} & \sigma ::= \texttt{var}\ \tau \\ \quad \mid \texttt{bool} & \quad \mid \texttt{unit} & \quad \mid \texttt{void} & \quad \mid \texttt{ret}\ T \\ \quad \mid \tau\texttt{[]} & \quad \mid (\tau_1, \tau_2, \ldots, \tau_n)\ ^{n\geq 2} & & \quad \mid \texttt{fn}\ T \rightarrow T' \end{array}$$

Ordinary types expressible in the language are denoted by the metavariable τ, which can be `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, `unit`, or a tuple type.

The type `unit` does not appear explicitly in the source language. This type is given to an element on the left-hand side of multiple assignments that uses the _ placeholder, allowing their handling be integrated directly into the type system. The `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 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 `var` τ for some type τ), return types (bound to `ret` T), functions (bound to `fn` $T \rightarrow T'$ where $T' \neq$ `unit`), or procedures (bound to `fn` $T \rightarrow$ `unit`), where the "result type" `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:

$$\frac{}{\tau \leq \texttt{unit}}$$

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 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:

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

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

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

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

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

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

$$\frac{\Gamma(f) = \texttt{fn}\ \texttt{unit} \to T' \quad T' \neq \texttt{unit}}{\Gamma \vdash f\texttt{()} : T'} \qquad \frac{\Gamma(f) = \texttt{fn}\ \tau \to T' \quad T' \neq \texttt{unit} \quad \Gamma \vdash e : \tau}{\Gamma \vdash f\texttt{(}e\texttt{)} : T'}$$

$$\frac{\Gamma(f) = \texttt{fn}\ (\tau_1, \ldots, \tau_n) \to T' \quad T' \neq \texttt{unit} \quad \Gamma \vdash e_i : \tau_i\ ^{(\forall i \in 1..n)} \quad n \geq 2}{\Gamma \vdash f\texttt{(}e_1\texttt{,} \ldots \texttt{,}e_n\texttt{)} : T'}$$

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 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 $T \neq \texttt{unit}$ if the statement is part of a function, or $T = \texttt{unit}$ 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 Γ'. To update typing contexts, we write $\Gamma[x \mapsto \texttt{var}\ \tau]$, which is an environment exactly like Γ except that it maps x to $\texttt{var}\ \tau$. 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 Γ:

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

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

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

$$\frac{\Gamma(f) = \texttt{fn}\,\texttt{unit} \rightarrow \texttt{unit}}{\Gamma \vdash f\texttt{()} : \texttt{unit}, \Gamma} \ (\textsc{PrCallUnit}) \qquad \frac{\Gamma(f) = \texttt{fn}\,\tau \rightarrow \texttt{unit} \quad \Gamma \vdash e : \tau}{\Gamma \vdash f\texttt{(}e\texttt{)} : \texttt{unit}, \Gamma} \ (\textsc{PrCall})$$

$$\frac{\Gamma(f) = \texttt{fn}\,(\tau_1, \ldots, \tau_n) \rightarrow \texttt{unit} \quad \Gamma \vdash e_i : \tau_i \ ^{(\forall i \in 1..n)} \quad n \geq 2}{\Gamma \vdash f\texttt{(}e_1\texttt{,} \ldots \texttt{,}e_n\texttt{)} : \texttt{unit}, \Gamma} \ (\textsc{PrCallMulti})$$

$$\frac{\Gamma(\rho) = \texttt{ret}\,\texttt{unit}}{\Gamma \vdash \texttt{return} : \texttt{void}, \Gamma} \ (\textsc{Return}) \qquad \frac{\Gamma(\rho) = \texttt{ret}\,\tau \quad \Gamma \vdash e : \tau}{\Gamma \vdash \texttt{return}\ e : \texttt{void}, \Gamma} \ (\textsc{RetVal})$$

$$\frac{\Gamma(\rho) = \texttt{ret}\,(\tau_1, \tau_2, \ldots, \tau_n) \quad \Gamma \vdash e_i : \tau_i \ ^{(\forall i \in 1..n)} \quad n \geq 2}{\Gamma \vdash \texttt{return}\ e_1\texttt{,}\ e_2\texttt{,}\ \ldots\texttt{,}\ e_n : \texttt{void}, \Gamma} \ (\textsc{RetMulti})$$

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:

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

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. To handle multiple assignments, we define a declaration d that can appear within a multiple assignment:

$$d ::= x\texttt{:}\tau \ \mid \ \texttt{\_}$$

and define functions $\mathit{typeof}(d)$ and $\mathit{varsof}(d)$ as follows:

$$\begin{array}{ll} \mathit{typeof}(x\texttt{:}\tau) = \tau & \mathit{typeof}(\texttt{\_}) = \texttt{unit} \\ \mathit{varsof}(x\texttt{:}\tau) = \{x\} & \mathit{varsof}(\texttt{\_}) = \emptyset \end{array}$$

Using these notations, we have the following rules:

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

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

$$\frac{\begin{array}{c}\Gamma \vdash e : (\tau_1, \ldots, \tau_n) \quad \tau_i \leq \mathit{typeof}(d_i) \ ^{(\forall i \in 1..n)} \\ \mathrm{dom}(\Gamma) \cap \mathit{varsof}(d_i) = \emptyset \ ^{(\forall i \in 1..n)} \quad \mathit{varsof}(d_i) \cap \mathit{varsof}(d_j) = \emptyset \ ^{(\forall i,j \in 1..n . j \neq i)}\end{array}}{\Gamma \vdash d_1\texttt{,}\ldots\texttt{,}d_n\ \texttt{=}\ e : \texttt{unit}, \Gamma[x_i \mapsto \texttt{var}\,\mathit{typeof}(d_i) \ ^{(\forall i \in 1..n \forall x_i . \mathit{varsof}(d_i) = \{x_i\})}]} \ (\textsc{MultiAssign})$$

The final premise in rule MULTIASSIGN prevents shadowing by ensuring that dom(Γ) and all of the *varsof*(d_i)'s are disjoint from each other.

5 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 $\Gamma \vdash fd : \Gamma'$ to state that the function or procedure declaration fd extends top-level bindings Γ to Γ':

$$\frac{f \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash f\texttt{()}\ \ s : \Gamma[f \mapsto \texttt{fn}\,\texttt{unit} \to \texttt{unit}]} \qquad \frac{f \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash f\texttt{(}x\texttt{:}\tau\texttt{)}\ \ s : \Gamma[f \mapsto \texttt{fn}\,\tau \to \texttt{unit}]}$$

$$\frac{f \notin \mathrm{dom}(\Gamma) \quad \Gamma' = \Gamma[f \mapsto \texttt{fn}\,(\tau_1, \ldots, \tau_n) \to \texttt{unit}] \quad n \geq 2}{\Gamma \vdash f\texttt{(}x_1\texttt{:}\tau_1\texttt{,}\ \ldots\texttt{,}x_n\texttt{:}\tau_n\texttt{)}\ \ s : \Gamma'}$$

$$\frac{f \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash f\texttt{():}\tau'\ \ s : \Gamma[f \mapsto \texttt{fn}\,\texttt{unit} \to \tau']} \qquad \frac{f \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash f\texttt{(}x\texttt{:}\tau\texttt{):}\tau'\ \ s : \Gamma[f \mapsto \texttt{fn}\,\tau \to \tau']}$$

$$\frac{f \notin \mathrm{dom}(\Gamma) \quad \Gamma' = \Gamma[f \mapsto \texttt{fn}\,(\tau_1, \ldots, \tau_n) \to \tau'] \quad n \geq 2}{\Gamma \vdash f\texttt{(}x_1\texttt{:}\tau_1\texttt{,}\ \ldots\texttt{,}x_n\texttt{:}\tau_n\texttt{):}\tau'\ \ s : \Gamma'}$$

$$\frac{f \notin \mathrm{dom}(\Gamma) \quad \Gamma' = \Gamma[f \mapsto \texttt{fn}\,\texttt{unit} \to (\tau'_1, \ldots, \tau'_m)] \quad m \geq 2}{\Gamma \vdash f\texttt{():}\tau'_1\texttt{,}\ \ldots\texttt{,}\tau'_m\ \ s : \Gamma'}$$

$$\frac{f \notin \mathrm{dom}(\Gamma) \quad \Gamma' = \Gamma[f \mapsto \texttt{fn}\,\tau \to (\tau'_1, \ldots, \tau'_m)] \quad m \geq 2}{\Gamma \vdash f\texttt{(}x\texttt{:}\tau\texttt{):}\tau'_1\texttt{,}\ \ldots\texttt{,}\tau'_m\ \ s : \Gamma'}$$

$$\frac{f \notin \mathrm{dom}(\Gamma) \quad \Gamma' = \Gamma[f \mapsto \texttt{fn}\,(\tau_1, \ldots, \tau_n) \to (\tau'_1, \ldots, \tau'_m)] \quad n \geq 2 \quad m \geq 2}{\Gamma \vdash f\texttt{(}x_1\texttt{:}\tau_1\texttt{,}\ \ldots\texttt{,}x_n\texttt{:}\tau_n\texttt{):}\tau'_1\texttt{,}\ \ldots\texttt{,}\tau'_m\ \ s : \Gamma'}$$

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

$$\frac{\Gamma[\rho \mapsto \texttt{ret}\,\texttt{unit}] \vdash s : R, \Gamma'}{\Gamma \vdash f\texttt{()}\ \ s\ \ \texttt{def}} \qquad \frac{x \notin \mathrm{dom}(\Gamma) \quad \Gamma[x \mapsto \texttt{var}\,\tau, \rho \mapsto \texttt{ret}\,\texttt{unit}] \vdash s : R, \Gamma'}{\Gamma \vdash f\texttt{(}x\texttt{:}\tau\texttt{)}\ \ s\ \ \texttt{def}}$$

$$\frac{\begin{array}{c}|\mathrm{dom}(\Gamma) \cup \{x_1, \ldots, x_n\}| = |\mathrm{dom}(\Gamma)| + n \quad n \geq 2 \\ \Gamma[x_1 \mapsto \texttt{var}\,\tau_1, \ldots, x_n \mapsto \texttt{var}\,\tau_n, \rho \mapsto \texttt{ret}\,\texttt{unit}] \vdash s : R, \Gamma'\end{array}}{\Gamma \vdash f\texttt{(}x_1\texttt{:}\tau_1\texttt{,}\ \ldots\texttt{,}x_n\texttt{:}\tau_n\texttt{)}\ \ s\ \ \texttt{def}}$$

$$\frac{\Gamma[\rho \mapsto \texttt{ret}\ \tau'] \vdash s : \texttt{void}, \Gamma'}{\Gamma \vdash f\texttt{()}\texttt{:}\tau'\ \ s\ \ \texttt{def}} \qquad \frac{x \notin \mathrm{dom}(\Gamma) \quad \Gamma[x \mapsto \texttt{var}\ \tau, \rho \mapsto \texttt{ret}\ \tau'] \vdash s : \texttt{void}, \Gamma'}{\Gamma \vdash f\texttt{(}x\texttt{:}\tau\texttt{)}\texttt{:}\tau'\ \ s\ \ \texttt{def}}$$

$$\frac{\begin{array}{c}|\mathrm{dom}(\Gamma) \cup \{x_1, \ldots, x_n\}| = |\mathrm{dom}(\Gamma)| + n \quad n \geq 2 \\ \Gamma[x_1 \mapsto \texttt{var}\ \tau_1, \ldots, x_n \mapsto \texttt{var}\ \tau_n, \rho \mapsto \texttt{ret}\ \tau'] \vdash s : \texttt{void}, \Gamma'\end{array}}{\Gamma \vdash f\texttt{(}x_1\texttt{:}\tau_1\texttt{,}\ \ldots\texttt{,}x_n\texttt{:}\tau_n\texttt{)}\texttt{:}\tau'\ \ s\ \ \texttt{def}}$$

$$\frac{\Gamma[\rho \mapsto \texttt{ret}\ (\tau'_1, \ldots, \tau'_m)] \vdash s : \texttt{void}, \Gamma' \quad m \geq 2}{\Gamma \vdash f\texttt{()}\texttt{:}\tau'_1\texttt{,}\ \ldots\texttt{,}\tau'_m\ \ s\ \ \texttt{def}}$$

$$\frac{x \notin \mathrm{dom}(\Gamma) \quad \Gamma[x \mapsto \texttt{var}\ \tau, \rho \mapsto \texttt{ret}\ (\tau'_1, \ldots, \tau'_m)] \vdash s : \texttt{void}, \Gamma' \quad m \geq 2}{\Gamma \vdash f\texttt{(}x\texttt{:}\tau\texttt{)}\texttt{:}\tau'_1\texttt{,}\ \ldots\texttt{,}\tau'_m\ \ s\ \ \texttt{def}}$$

$$\frac{\begin{array}{c}|\mathrm{dom}(\Gamma) \cup \{x_1, \ldots, x_n\}| = |\mathrm{dom}(\Gamma)| + n \quad n \geq 2 \quad m \geq 2 \\ \Gamma[x_1 \mapsto \texttt{var}\ \tau_1, \ldots, x_n \mapsto \texttt{var}\ \tau_n, \rho \mapsto \texttt{ret}\ (\tau'_1, \ldots, \tau'_m)] \vdash s : \texttt{void}, \Gamma'\end{array}}{\Gamma \vdash f\texttt{(}x_1\texttt{:}\tau_1\texttt{,}\ \ldots\texttt{,}x_n\texttt{:}\tau_n\texttt{)}\texttt{:}\tau'_1\texttt{,}\ \ldots\texttt{,}\tau'_m\ \ s\ \ \texttt{def}}$$

6 Checking a program

Using the previous judgments, we can define when an entire program $fd_1\, fd_2\ \ldots\ fd_n$ that does not contain a `use` declaration is well-formed, written $\vdash fd_1\, fd_2\ \ldots\ fd_n\ \ \texttt{prog}$:

$$\frac{\emptyset \vdash fd_1 : \Gamma_1 \quad \Gamma_1 \vdash fd_2 : \Gamma_2 \quad \ldots \quad \Gamma_{n-1} \vdash fd_n : \Gamma \quad \Gamma \vdash fd_i\ \ \texttt{def}\ ^{(\forall i \in 1..n)}}{\vdash fd_1\, fd_2\ \ldots\ fd_n\ \ \texttt{prog}}$$

For brevity, the rules for adding declarations appearing in interfaces are omitted. These rules are slightly different from those of the form $\Gamma \vdash fd : \Gamma'$ in Section 5, 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 **??** of the Xi Language Specification.