Xi Type System Specification

Computer Science 4120
Cornell University

Version of October 2, 2011

Changes

- October 2: Made ArrayDecl more general to match up with the language spec.
- September 30: Fixed bugs reported in recitation and some more. Tuple expressions and types may have 0 or 1 elements.
- September 29: Fixed syntax in ArrAssign.

Types

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

$$
\begin{aligned}
\tau &::= \texttt{int} \\
&\mid \texttt{bool} \\
&\mid \tau\texttt{[]} \\
R &::= \texttt{unit} \mid \texttt{void} \\
T &::= \tau \\
&\mid (\tau_1, \tau_2, \ldots, \tau_n) \quad {}^{(n \geq 0)} \\
&\mid R \\
\sigma &::= \texttt{var}\ \tau \\
&\mid T_1 \rightarrow T_2
\end{aligned}
$$

Ordinary types expressible in the language are denoted by the metavariable τ, which can be `int`, `bool`, or an array type.

The metavariable R represent the outcome of evaluating a statement. It can be either `unit` or `void`.

The type `unit` is used in various ways. It is the type of ordinary statements. It gives a type to the left-hand side of pattern-matching assignments that use the _ placeholder, which lets their handling be integrated directly into the type system. The `unit` type is also used to represent the result type of procedures and the type of statements that may complete normally and permit the following statement to execute.

The type `void` is the type of statements such as `return` that pass control to a following statement. It should not be confused with the C type `void`, which is actually closer to `unit`.

The metavariable T denotes an expanded notion of type that includes the parameter types and return types of procedures and functions. It may be an ordinary type, a tuple type, `unit`, or `void`.

Tuple types represent the parameters or return types of functions that take multiple argument or return multiple results.

The set σ is used to represent typing environment entries, which can either be normal variables (bound to `var` τ for some type τ) or functions (bound to $\tau \rightarrow \tau'$ where $\tau' \neq$ `unit`), or procedures (bound to

$\tau \to$ `unit`), where the "result type" (`unit`) indicates that the procedure result contains no information other than that the procedure call terminated.

Subtyping

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

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

However, there is not (for now) a subsumption rule, so subtyping matters only where it appears explicitly.

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 $\Gamma \vdash e : \tau$ is the rule for the type of an expression; it states that with bindings Γ we can conclude that e has the type τ.

We use the metavariable symbols x or f to represent arbitrary identifiers, n to represent a numeric constant, *string* to represent a literal string constant, and *char* to represent a literal character constant. Using these conventions, the expression typing rules are:

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

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

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

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

$$\frac{\Gamma \vdash e_1 : \tau\texttt{[]} \quad \Gamma \vdash e_2 : \tau\texttt{[]} \quad \ominus \in \{==, !=\}}{\Gamma \vdash e_1 \ominus e_2 : \texttt{bool}} \qquad \frac{\Gamma \vdash e_1 : \tau \quad \ldots \quad \Gamma \vdash e_n : \tau \quad n \geq 0}{\Gamma \vdash (e_1, \ldots, e_n) : \tau\texttt{[]}}$$

$$\frac{\Gamma \vdash e_1 : \tau\texttt{[]} \quad \Gamma \vdash e_2 : \tau\texttt{[]}}{\Gamma \vdash e_1 + e_2 : \tau\texttt{[]}} \qquad \frac{\Gamma \vdash e_1 : \tau\texttt{[]} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1\texttt{[}e_2\texttt{]} : \tau} \qquad \frac{\Gamma(f) = (\tau_1, \ldots, \tau_n) \to \tau' \quad \Gamma \vdash e_i : \tau_i \ ^{(\forall i \in 1..n)}}{\Gamma \vdash f(e_1, \ldots, e_n) : \tau'}$$

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 two special symbols, ρ and β. 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 τ if the statement is part of a function, or `unit` if the statement is in a procedure. For `break` statements, we also need to check whether we are inside a loop, which we will denote as $\Gamma(\beta)$, which is `unit` if we are inside a loop and `void` if we are not. 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 $\Gamma[x \mapsto \tau]$, which is an environment exactly like Γ except that it maps x to τ. We use the metavariable s to denote a statement, so the main typing judgment for statements has either the form $\Gamma \vdash s : \texttt{unit}, \Gamma'$ or the form $\Gamma \vdash s : \texttt{void}, \Gamma'$.

Most of the statements are fairly straightforward, and do not change Γ. However, statements like `break` and `return` are a bit tricky because their type is `void`.

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

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

$$\frac{\Gamma \vdash s_1 : \texttt{unit}, \Gamma_1 \quad \Gamma_1 \vdash s_2 : \texttt{unit}, \Gamma_2 \quad \ldots \quad \Gamma_{n-2} \vdash s_{n-1} : \texttt{unit}, \Gamma_{n-1} \quad \Gamma_{n-1} \vdash s_n : R, \Gamma_n}{\Gamma \vdash \{s_1; s_2; \ldots; s_n\} : R, \Gamma_n}\ (\textsc{Block})$$

$$\frac{\Gamma(f) = \tau \rightarrow \texttt{unit} \quad \Gamma \vdash e : \tau}{\Gamma \vdash f(e_1, \ldots, e_n) : \texttt{unit}, \Gamma}\ (\textsc{PrCall}) \qquad \frac{\Gamma(\beta) = \texttt{unit}}{\Gamma \vdash \texttt{break} : \texttt{void}, \Gamma}\ (\textsc{Break})$$

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

The function *lub* is defined as $\mathit{lub}(R, R) = R$ and $\mathit{lub}(\texttt{unit}, R) = \mathit{lub}(R, \texttt{unit}) = \texttt{unit}$. Therefore the type of an `if` is `void` only if both branches have that type.

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

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

Declarations are the source of new bindings. Three kinds of declarations can appear in the source language: regular variable declarations, tuple declarations, and function/procedure declarations. We are only concerned with the first two kinds within a function body. To handle tuples, we define a declaration d that can appear within a tuple:

$$d ::= x : \tau \mid \_$$

and define functions $\mathit{typeof}(d)$ and $\mathit{varsof}(d)$ as follows: $\mathit{typeof}(x : \tau) = \tau$ and $\mathit{typeof}(\_) = \texttt{unit}$, and $\mathit{varsof}(x{:}\tau) = \{x\}$ and $\mathit{varsof}(\_) = \emptyset$. Using these notations, we have the following rules:

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

The following rule for array declarations with specified sizes is intended to capture the essence of type checking them, though we have sacrificed a bit of formal precision for the sake of readability. The declared variable is added to the typing context with a type with the same number of array dimensions.

$$\frac{x \notin \mathrm{dom}(\Gamma) \quad \Gamma \vdash e_i : \texttt{int}\ ^{(\forall i \in 1..n)}}{\Gamma \vdash x{:}\tau\,\overrightarrow{\texttt{[}e\texttt{]}}\,\overrightarrow{\texttt{[]}} : \Gamma[x \mapsto \tau\,\overrightarrow{\texttt{[]}}\,\overrightarrow{\texttt{[]}}]}\ (\textsc{ArrayDecl})$$

$$\frac{\begin{array}{lll} \Gamma(f) = (\tau'_1, \ldots, \tau'_m) \rightarrow (\tau_1, \ldots, \tau_n) & \Gamma \vdash e_i : \tau'_i\ ^{(\forall i \in 1..m)} & \tau_i \leq \mathit{typeof}(d_i)\ ^{(\forall i \in 1..n)} \\ \mathrm{dom}(\Gamma) \cap \mathit{varsof}(d_i) = \emptyset\ ^{(\forall i \in 1..n)} & \mathit{varsof}(d_i) \cap \mathit{varsof}(d_j) = \emptyset\ ^{(\forall i,j \in 1..n \mid j \neq i)} & \end{array}}{\Gamma \vdash d_1, \ldots, d_n = f(e_1, \ldots, e_m) : \Gamma[x_i \mapsto \mathit{typeof}(d_i)\ ^{(\forall i \in 1..n, x_i \mid \mathit{varsof}(d_i) = \{x_i\})}]}\ (\textsc{TupleDecl})$$

The final premise in rule TupleDecl prevents shadowing by ensuring that $\mathrm{dom}(\Gamma)$ and all of the $\mathit{varsof}(d_i)$ are disjoint from each other.

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

$$\frac{f \notin \text{dom}(\Gamma)}{\Gamma \vdash f(x\!:\!\tau) : \tau' = s : \Gamma[f \mapsto \tau \to \tau']} \qquad \frac{f \notin \text{dom}(\Gamma) \quad n \geq 2 \quad \Gamma' = \Gamma[f \mapsto (\tau_1, \tau_2, \ldots, \tau_n) \to \tau_r]}{\Gamma \vdash f(x_1\!:\!\tau_1, x_2\!:\!\tau_2, \ldots, x_n\!:\!\tau_n) : \tau_r = s : \Gamma'}$$

$$\frac{f \notin \text{dom}(\Gamma)}{\Gamma \vdash f(x\!:\!\tau) = s : \Gamma[f \mapsto \tau \to \texttt{unit}]} \qquad \frac{f \notin \text{dom}(\Gamma) \quad n \geq 2 \quad \Gamma' = \Gamma[f \mapsto (\tau_1, \tau_2, \ldots, \tau_n) \to \texttt{unit}]}{\Gamma \vdash f(x_1\!:\!\tau_1, x_2\!:\!\tau_2, \ldots, x_n\!:\!\tau_n) = s : \Gamma'}$$

The second pass over the program is captured by the judgment $\Gamma \vdash \mathit{fd}\ \texttt{def}$, 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. The body of a function definition must have type `void`, which ensures that the function body does not fall off the end without returning. We treat procedures just like functions that return the unit type. Therefore their bodies are allowed to have type `unit`.

$$\frac{x \notin \text{dom}(\Gamma) \quad \Gamma[x \mapsto \tau, \rho \mapsto \tau', \beta \mapsto \texttt{void}] \vdash s : \texttt{void}, \Gamma'}{\Gamma \vdash f(x\!:\!\tau) : \tau' = s\ \ \texttt{def}}$$

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

$$\frac{x \notin \text{dom}(\Gamma) \quad \Gamma[x \mapsto \tau, \rho \mapsto \texttt{unit}, \beta \mapsto \texttt{void}] \vdash s : \texttt{unit}, \Gamma'}{\Gamma \vdash f(x\!:\!\tau) = s\ \ \texttt{def}}$$

$$\frac{\begin{array}{c}|\text{dom}(\Gamma) \cup \{x_1, \ldots, x_n\}| = |\text{dom}(\Gamma)| + n \\ \Gamma[x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n, \rho \mapsto \texttt{unit}, \beta \mapsto \texttt{void}] \vdash s : \texttt{unit}, \Gamma'\end{array}}{\Gamma \vdash f(x_1\!:\!\tau_1, \ldots, x_n\!:\!\tau_n) = s\ \texttt{def}}$$

Checking a program

Using the previous judgments, we can define when an entire program $\mathit{fd}_1 \ldots \mathit{fd}_n$ is well-formed, written $\vdash \mathit{fd}_1 \ldots \mathit{fd}_n\ \texttt{prog}$:

$$\frac{\begin{array}{c}\emptyset \vdash d_1 : \Gamma_1 \quad \Gamma_1 \vdash d_2 : \Gamma_2 \quad \ldots \quad \Gamma_{n-1} \vdash d_n : \Gamma_n \\ \Gamma_n \vdash d_1\ \texttt{def} \quad \Gamma_n \vdash d_2\ \texttt{def} \quad \ldots \quad \Gamma_n \vdash d_n\ \texttt{def}\end{array}}{\vdash \mathit{fd}_1\ \mathit{fd}_2\ \ldots\ \mathit{fd}_n\ \texttt{prog}}$$