Iota$_9$ Type System Specification

Computer Science 4120
Cornell University

Version of September 27, 2009

Changes

- September 27: Rule (TUPLEDECL) updated so underscores work. Rule (ARRAYDECL) now allows arbitrary integer expressions as array lengths, as in the spec.

Types

The Iota$_9$ type system uses a somewhat bigger set of types than can be expressed explicitly in the source language:

$$\begin{aligned}
\tau &::= \texttt{int} \\
&\mid \texttt{bool} \\
&\mid _ \\
&\mid \tau\texttt{[]} \\
&\mid (\tau_1, \tau_2, \ldots, \tau_\texttt{n}) \quad {\scriptstyle (n \geq 2)} \\
\sigma &::= \texttt{var}\ \tau \\
&\mid \texttt{fn}\ \tau \rightarrow \tau'
\end{aligned}$$

The _ type is how we will write the unit type, a type that does not appear explicitly in the source language. The unit type is used to give a type to the left-hand side of pattern-matching assignments that use the _ placeholder; this lets their handling be integrated directly into the type system. The _ type is also used to represent the result type of procedures.

The set σ is used to represent typing environment entries, which can either be normal variables (bound to $\texttt{var}\ \tau$ for some type τ) or functions (bound to $\texttt{fn}\ \tau \rightarrow \tau'$ where $\tau' \neq \_$), or procedures (bound to $\texttt{fn}\ \tau \rightarrow \_$), where the "result type" (_) indicates that the procedure result contains no information other than that the procedure call terminated.

Subtyping

We use the following subtyping relation:

$$\frac{}{\tau \leq \tau} \qquad \frac{}{\tau \leq \_}$$

$$\frac{\tau_1 \leq \tau_c \quad \ldots \quad \tau_n \leq \tau_c}{(\tau_1, \ldots, \tau_n) \leq \tau_c\texttt{[]}} \qquad \frac{\tau_{1,a} \leq \tau_{1,b} \quad \ldots \quad \tau_{n,a} \leq \tau_{n,b}}{(\tau_{1,a}, \ldots, \tau_{n,a}) \leq (\tau_{1,b}, \ldots, \tau_{n,b})}$$

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.

We define two typing judgment. 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. There is also another typing judgment, $\vdash_u$ (read: usable as), which indicates how to incorporate subtyping through the following subsumption rule:

$$\frac{\Gamma \vdash e : \tau \quad \tau \leq \tau'}{\Gamma \vdash_u e : \tau'}$$

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

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

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

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

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

$$\frac{\Gamma \vdash_u e_1 : \tau\texttt{[]} \quad \Gamma \vdash_u e_2 : \tau\texttt{[]} \quad \ominus \in \{==, !=\}}{\Gamma \vdash e_1 \ominus e_2 : \texttt{bool}} \quad \frac{\Gamma \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma \vdash e_n : \tau_n \quad n \geq 2}{\Gamma \vdash (e_1, \ldots, e_n) : (\tau_1, \ldots, \tau_n)}$$

$$\frac{\Gamma \vdash e_1 : \tau\texttt{[]} \quad \Gamma \vdash_u e_2 : \texttt{int}}{\Gamma \vdash e_1\ e_2 : \tau} \quad \frac{\Gamma(f) = \texttt{fn}\ \tau \to \tau' \quad \Gamma \vdash_u e : \tau}{\Gamma \vdash f\ e : \tau'}$$

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 _ 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 `true` if we are inside a loop and `false` 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 the form $\Gamma \vdash s : \Gamma'$. (The type of a statement, if we wanted to give it one, would be _.)

Most of the statements are fairly straightforward, and do not change Γ:

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

$$\frac{}{\Gamma \vdash ; : \Gamma}\ (\textsc{Empty}) \qquad \frac{\Gamma \vdash e : \texttt{bool} \quad \Gamma[\beta \mapsto \texttt{true}] \vdash s : \Gamma'}{\Gamma \vdash \texttt{while}\ (e)\ s : \Gamma}\ (\textsc{While})$$

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

$$\frac{\Gamma(f) = \texttt{fn}\ \tau \to \_ \quad \Gamma \vdash_u e : \tau}{\Gamma \vdash f\ e : \Gamma}\ (\textsc{PrCall}) \qquad \frac{\Gamma(\beta) = \texttt{true}}{\Gamma \vdash \texttt{break} : \Gamma}\ (\textsc{Break})$$

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

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

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

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 $\mathit{typeof}(\_) = \_$, and $\mathit{varsof}(x : \tau) = \{x\}$ and $\mathit{varsof}(\_) = \emptyset$. Using these notations, we have the following rules:

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

$$\frac{\Gamma \vdash e : (\tau_1, \ldots, \tau_n) \qquad \tau_i \leq \mathit{typeof}(d_i)\ ^{(\forall i \in 1..n)} \\ \mathrm{dom}(\Gamma) \cap \mathit{varsof}(d_i) = \emptyset\ ^{(\forall i \in 1..n)} \qquad \mathit{varsof}(d_i) \cap \mathit{varsof}(d_j) = \emptyset\ ^{(\forall i,j \in 1..n \mid j \neq i)}}{\Gamma \vdash (d_1, \ldots, d_n) = e : \Gamma[x_i \mapsto \mathit{typeof}(d_i)\ ^{(\forall i \in 1..n, x_i \mid \mathit{varsof}(d_i) = \{x_i\})}]}\ (\textsc{TupleDecl})$$

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

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

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

The second pass over the program is captured by the judgment $\Gamma \vdash fd$ `decl`, 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:

$$\frac{x \notin \mathrm{dom}(\Gamma) \quad \Gamma[x \mapsto \tau, \rho \mapsto \tau', \beta \mapsto \texttt{false}] \vdash s : \Gamma'}{\Gamma \vdash f(x{:}\tau) : \tau' = s \ \texttt{decl}} \qquad \frac{|\mathrm{dom}(\Gamma) \cup \{x_1, \ldots, x_n\}| = |\mathrm{dom}(\Gamma)| + n \quad \Gamma[x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n, \rho \mapsto \tau', \beta \mapsto \texttt{false}] \vdash s : \Gamma'}{\Gamma \vdash f(x_1{:}\tau_1, \ldots, x_n{:}\tau_n) : \tau' = s \ \texttt{decl}}$$

$$\frac{x \notin \mathrm{dom}(\Gamma) \quad \Gamma[x \mapsto \tau, \rho \mapsto \_, \beta \mapsto \texttt{false}] \vdash s : \Gamma'}{\Gamma \vdash f(x{:}\tau) = s \ \texttt{decl}} \qquad \frac{|\mathrm{dom}(\Gamma) \cup \{x_1, \ldots, x_n\}| = |\mathrm{dom}(\Gamma)| + n \quad \Gamma[x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n, \rho \mapsto \_, \beta \mapsto \texttt{false}] \vdash s : \Gamma'}{\Gamma \vdash f(x_1{:}\tau_1, \ldots, x_n{:}\tau_n) = s \ \texttt{decl}}$$

Checking a program

Using the previous judgments, we can define when an entire program $fd_1 \ldots fd_n$ is well-formed, written $\vdash fd_1 \ldots fd_n$ `prog`:

$$\frac{\emptyset \vdash d_1 : \Gamma_1 \quad \Gamma_1 \vdash d_2 : \Gamma_2 \quad \ldots \quad \Gamma_n \vdash d_n : \Gamma \quad \Gamma \vdash d_1 \ \texttt{decl} \quad \Gamma \vdash d_2 \ \texttt{decl} \quad \ldots \quad \Gamma \vdash d_n \ \texttt{decl}}{\vdash fd_1 \, fd_2 \, \ldots \, fd_n \ \texttt{prog}}$$