Type Inference as an Instance for Modular Analysis

Joonhyup Lee

November 28, 2023

1 For the Simple Module Language

$$
\begin{array}{rrcll}
\text{Identifiers} & x, d & \in & \text{Var} & \\
\text{Expression} & e & \rightarrow & () \mid x \mid \lambda x.e \mid e\ e & \lambda\text{-calculus with unit} \\
 & & \mid & e \bowtie e & \text{linked expression} \\
 & & \mid & \varepsilon & \text{empty module} \\
 & & \mid & d & \text{module identifier} \\
 & & \mid & \texttt{val } x\ e\ e & \text{value binding} \\
 & & \mid & \texttt{mod } d\ e\ e & \text{module binding}
\end{array}
$$

Figure 1: Abstract syntax of the simple module language.

1.1 Operational Semantics

$$
\begin{array}{rrcll}
\text{Environment/Context} & \sigma & \in & \text{Ctx} & \\
\text{Value of expressions} & v & \in & \text{Val} \triangleq \{()\} + \text{Var} \times \text{Expr} \times \text{Ctx} & \\
\text{Value of expressions/modules} & V & \in & \text{Val} + \text{Ctx} & \\
\text{Context} & \sigma & \rightarrow & \bullet & \text{empty stack} \\
 & & \mid & (x, v) :: \sigma & \text{expression binding} \\
 & & \mid & (d, \sigma) :: \sigma & \text{module binding} \\
\text{Value of expressions} & v & \rightarrow & () & \text{unit} \\
 & & \mid & \langle \lambda x.e, \sigma \rangle & \text{closure}
\end{array}
$$

Figure 2: Definition of the semantic domains.

$$\boxed{(e, \sigma) \Downarrow V}$$

$$
[\textsc{Unit}]\ \dfrac{}{((), \sigma) \Downarrow ()} \qquad
[\textsc{ExprID}]\ \dfrac{v = \sigma(x)}{(x, \sigma) \Downarrow v} \qquad
[\textsc{Fn}]\ \dfrac{}{(\lambda x.e, \sigma) \Downarrow \langle \lambda x.e, \sigma \rangle} \qquad
[\textsc{App}]\ \dfrac{\begin{array}{c}(e_1, \sigma) \Downarrow \langle \lambda x.e_\lambda, \sigma_\lambda \rangle \\ (e_2, \sigma) \Downarrow v \\ (e_\lambda, (x, v) :: \sigma_\lambda) \Downarrow v'\end{array}}{(e_1\ e_2, \sigma) \Downarrow v'} \qquad
[\textsc{Link}]\ \dfrac{\begin{array}{c}(e_1, \sigma) \Downarrow \sigma' \\ (e_2, \sigma') \Downarrow V\end{array}}{(e_1 \bowtie e_2, \sigma) \Downarrow V}
$$

$$
[\textsc{Empty}]\ \dfrac{}{(\varepsilon, \sigma) \Downarrow \bullet} \qquad
[\textsc{ModID}]\ \dfrac{\sigma' = \sigma(d)}{(d, \sigma) \Downarrow \sigma'} \qquad
[\textsc{LetE}]\ \dfrac{\begin{array}{c}(e_1, \sigma) \Downarrow v \\ (e_2, (x, v) :: \sigma) \Downarrow \sigma'\end{array}}{(\texttt{val } x\ e_1\ e_2, \sigma) \Downarrow (x, v) :: \sigma'} \qquad
[\textsc{LetM}]\ \dfrac{\begin{array}{c}(e_1, \sigma) \Downarrow \sigma' \\ (e_2, (d, \sigma') :: \sigma) \Downarrow \sigma''\end{array}}{(\texttt{mod } d\ e_1\ e_2, \sigma) \Downarrow (d, \sigma') :: \sigma''}
$$

Figure 3: The big-step operational semantics.

1.2 Typing

The definitions for types are in Figure 4 and the typing rules are in Figure 5.

1.3 Type Safety

Claim 1.1 (Type Safety). For all $e \in \text{Expr}$, if $\bullet \vdash e : \tau$ for some τ, then there exists some $v \in \text{Val}$ such that $(e, \bullet) \Downarrow v$. Likewise, if $\bullet \vdash e : \Gamma$ for some Γ, then there exists some $\sigma \in \text{Ctx}$ such that $(e, \bullet) \Downarrow \sigma$.

$$\begin{array}{rccll}
\text{Types} & \tau & \to & \iota & \text{unit type} \\
& & | & \tau \to \tau & \text{function type} \\
\text{Type Environment} & \Gamma & \to & \bullet & \text{empty environment} \\
& & | & (x, \tau) :: \Gamma & \text{expression binding} \\
& & | & (d, \Gamma) :: \Gamma & \text{module binding}
\end{array}$$

Figure 4: Definition of types.

$$\boxed{\Gamma \vdash e : \tau \text{ or } \Gamma \vdash e : \Gamma}$$

$$[\textsc{Unit}] \; \frac{}{\Gamma \vdash () : \iota} \qquad [\textsc{ExprID}] \; \frac{\tau = \Gamma(x)}{\Gamma \vdash x : \tau} \qquad [\textsc{Fn}] \; \frac{(x, \tau_1) :: \Gamma \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2} \qquad [\textsc{App}] \; \frac{\begin{array}{c}\Gamma \vdash e_1 : \tau' \to \tau \\ \Gamma \vdash e_2 : \tau'\end{array}}{\Gamma \vdash e_1 \, e_2 : \tau} \qquad [\textsc{Link}] \; \frac{\begin{array}{c}\Gamma \vdash e_1 : \Gamma_1 \\ \Gamma_1 \vdash e_2 : \tau_2 \text{ or } \Gamma_2\end{array}}{\Gamma \vdash e_1 \rtimes e_2 : \tau_2 \text{ or } \Gamma_2}$$

$$[\textsc{Empty}] \; \frac{}{\Gamma \vdash \varepsilon : \bullet} \qquad [\textsc{ModID}] \; \frac{\Gamma' = \Gamma(d)}{\Gamma \vdash d : \Gamma'} \qquad [\textsc{LetE}] \; \frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1 \\ (x, \tau_1) :: \Gamma \vdash e_2 : \Gamma_2\end{array}}{\Gamma \vdash \mathtt{val}\, x\, e_1\, e_2 : (x, \tau_1) :: \Gamma_2} \qquad [\textsc{LetM}] \; \frac{\begin{array}{c}\Gamma \vdash e_1 : \Gamma_1 \\ (d, \Gamma_1) :: \Gamma \vdash e_2 : \Gamma_2\end{array}}{\Gamma \vdash \mathtt{mod}\, d\, e_1\, e_2 : (d, \Gamma_1) :: \Gamma_2}$$

Figure 5: The typing judgment.

Proof sketch. We prove this through unary logical relations and induction on the typing judgment.

Value Relation $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \boxed{\mathcal{V}[\![\tau]\!]}$

$$\begin{array}{rcl}
\mathcal{V}[\![\iota]\!] & \triangleq & \{()\} \\
\mathcal{V}[\![\tau_1 \to \tau_2]\!] & \triangleq & \{\langle \lambda x.e, \sigma \rangle \, | \, \forall v \in \mathcal{V}[\![\tau_1]\!] : (e, (x, v) :: \sigma) \in \mathcal{E}[\![\tau_2]\!]\}
\end{array}$$

Expression Relation $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \boxed{\mathcal{E}[\![\tau]\!] \text{ or } \mathcal{E}[\![\Gamma]\!]}$

$$\begin{array}{rcl}
\mathcal{E}[\![\tau]\!] & \triangleq & \{(e, \sigma) \, | \, \exists v \in \mathcal{V}[\![\tau]\!] : (e, \sigma) \Downarrow v\} \\
\mathcal{E}[\![\Gamma]\!] & \triangleq & \{(e, \sigma) \, | \, \exists \sigma' \in \mathcal{C}[\![\Gamma]\!] : (e, \sigma) \Downarrow \sigma'\}
\end{array}$$

Context Relation $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \boxed{\mathcal{C}[\![\Gamma]\!]}$

$$\begin{array}{rcl}
\mathcal{C}[\![\bullet]\!] & \triangleq & \{\bullet\} \\
\mathcal{C}[\![(x, \tau_1) :: \Gamma_2]\!] & \triangleq & \{\sigma \, | \, \sigma(x) \in \mathcal{V}[\![\tau_1]\!] \wedge (\sigma - x) \in \mathcal{C}[\![\Gamma_2 - x]\!]\} \\
\mathcal{C}[\![(d, \Gamma_1) :: \Gamma_2]\!] & \triangleq & \{\sigma \, | \, \sigma(d) \in \mathcal{C}[\![\Gamma_1]\!] \wedge (\sigma - d) \in \mathcal{C}[\![\Gamma_2 - d]\!]\}
\end{array}$$

Semantic Typing $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \boxed{\Gamma \vDash e : \tau \text{ or } \Gamma \vDash e : \Gamma}$

$$\begin{array}{rcl}
\Gamma \vDash e : \tau & \triangleq & \forall \sigma \in \mathcal{C}[\![\Gamma]\!] : (e, \sigma) \in \mathcal{E}[\![\tau]\!] \\
\Gamma \vDash e : \Gamma' & \triangleq & \forall \sigma \in \mathcal{C}[\![\Gamma]\!] : (e, \sigma) \in \mathcal{C}[\![\Gamma']\!]
\end{array}$$

We want to prove that:

$$\Gamma \vdash e : \tau \Rightarrow \Gamma \vDash e : \tau$$

$$\Gamma \vdash e : \Gamma' \Rightarrow \Gamma \vDash e : \Gamma'$$

by induction on $\vdash$.

For the base cases of ι and $\bullet$, the proof is trivial. For inductive cases, we need to show *compatibility* lemmas. That is, we must show that the typing rules for syntactic typing hold for semantic typing as well. Then by the inductive hypothesis and compatibility, the result follows. $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\square$

1.4 Type Inference

For the simple module language, the operational semantics constrains which expressions *must* evaluate to contexts and which expressions *must* evaluate to closures. Therefore, type inference in this language is simple, as the shape of the typing environment can be accurately inferred from the syntax of the program.

First we define the syntax for type constraints.

$$
\begin{array}{llll}
\text{Type Variable} & \alpha & \in & \text{TyVar} \\
\text{Module Path} & p & \to & \epsilon & \text{empty string} \\
& & | & pd & \text{concatenation with module identifier} \\
\text{Types} & \tau & \to & \iota \mid \tau \to \tau & \text{unit/function types} \\
& & | & \alpha & \text{type variables} \\
& & | & [].p.x & \text{types from the external environment} \\
\text{Type Environment} & \Gamma & \to & \bullet & \text{empty environment} \\
& & | & (x, \tau) :: \Gamma & \text{expression binding} \\
& & | & (d, \Gamma) :: \Gamma & \text{module binding} \\
& & | & [].p & \text{modules from the external environment} \\
\text{Type Constraint} & u & \to & \tau \doteq \tau & \text{equality constraint} \\
\text{Set of Constraints} & U & \subseteq & \{u \mid u \text{ type constraint}\}
\end{array}
$$

Figure 6: Definition of type constraints.

Next we define the module access operation $\Gamma(d)$ and the type access operation $\Gamma(x)$:

$$
\begin{array}{ll}
\bullet(d) \triangleq \bot & \bullet(x) \triangleq \bot \\
((x, \_) :: \Gamma)(d) \triangleq \Gamma(d) & ((d, \_) :: \Gamma)(x) \triangleq \Gamma(x) \\
((d, \Gamma) :: \_)(d) \triangleq \Gamma & ((x, \tau) :: \_)(x) \triangleq \tau \\
((d', \_) :: \Gamma)(d) \triangleq \Gamma(d) \quad (d' \neq d) & ((x', \_) :: \Gamma)(x) \triangleq \Gamma(x) \quad (x' \neq x) \\
([].p)(d) \triangleq [].pd & ([].p)(x) \triangleq [].p.x
\end{array}
$$

Now we can define the constraint generation algorithm $V(\Gamma, e, \alpha)$. Note that the **let** $U = \_$ **in** $\_$ notation returns $\bot$ if the right hand side is not a set of type constraints. Likewise, the **let** $(\Gamma, U) = \_$ **in** $\_$ notation returns $\bot$ if the right hand side is not a pair of a type environment and a set of type constraints.

$$\boxed{V(\Gamma, e, \alpha) = U \text{ or } (\Gamma, U) \text{ or } \bot}$$

$$
\begin{array}{lcl}
V(\Gamma, (), \alpha) & \triangleq & \{\alpha \doteq \iota\} \\
V(\Gamma, x, \alpha) & \triangleq & \textbf{let } \tau = \Gamma(x) \textbf{ in} \\
& & \{\alpha \doteq \tau\} \\
V(\Gamma, \lambda x.e, \alpha) & \triangleq & \textbf{let } \alpha_1, \alpha_2 = \textit{fresh } \textbf{in} \\
& & \textbf{let } U = V((x, \alpha_1) :: \Gamma, e, \alpha_2) \textbf{ in} \\
& & \{\alpha \doteq \alpha_1 \to \alpha_2\} \cup U \\
V(\Gamma, e_1\, e_2, \alpha) & \triangleq & \textbf{let } \alpha_1, \alpha_2 = \textit{fresh } \textbf{in} \\
& & \textbf{let } U_1 = V(\Gamma, e_1, \alpha_1) \textbf{ in} \\
& & \textbf{let } U_2 = V(\Gamma, e_2, \alpha_2) \textbf{ in} \\
& & \{\alpha_1 \doteq \alpha_2 \to \alpha\} \cup U_1 \cup U_2 \\
V(\Gamma, e_1 \bowtie e_2, \alpha) & \triangleq & \textbf{let } (\Gamma_1, U_1) = V(\Gamma, e_1, \alpha) \textbf{ in} \\
& & \textbf{case } V(\Gamma_1, e_2, \alpha) : \\
& & \quad U_2 \to U_1 \cup U_2 \\
& & \quad (\Gamma_2, U_2) \to (\Gamma_2, U_1 \cup U_2)
\end{array}
$$

$$
\begin{array}{lcl}
V(\Gamma, \varepsilon, \alpha) & \triangleq & (\bullet, \emptyset) \\
V(\Gamma, d, \alpha) & \triangleq & \textbf{let } \Gamma' = \Gamma(d) \textbf{ in} \\
& & (\Gamma', \emptyset) \\
V(\Gamma, \texttt{val } x\, e_1\, e_2, \alpha) & \triangleq & \textbf{let } \alpha_1 = \textit{fresh } \textbf{in} \\
& & \textbf{let } U_1 = V(\Gamma, e_1, \alpha_1) \textbf{ in} \\
& & \textbf{let } (\Gamma_2, U_2) = V((x, \alpha_1) :: \Gamma, e_2, \alpha) \textbf{ in} \\
& & ((x, \alpha_1) :: \Gamma_2, U_1 \cup U_2) \\
V(\Gamma, \texttt{mod } d\, e_1\, e_2, \alpha) & \triangleq & \textbf{let } (\Gamma_1, U_1) = V(\Gamma, e_1, \alpha) \textbf{ in} \\
& & \textbf{let } (\Gamma_2, U_2) = V((d, \Gamma_1) :: \Gamma, e_2, \alpha) \textbf{ in} \\
& & ((d, \Gamma_1) :: \Gamma_2, U_1 \cup U_2)
\end{array}
$$

We want to prove that the constraint generation algorithm is correct.

First, for $\Gamma_{\text{ext}} \in \text{TyEnv}$, define the access operations $\Gamma_{\text{ext}}.p$ and $\Gamma_{\text{ext}}.p.x$ (which may fail):

$$
\Gamma_{\text{ext}}.\epsilon \triangleq \Gamma_{\text{ext}} \qquad\qquad \Gamma_{\text{ext}}.pd \triangleq (\Gamma_{\text{ext}}.p)(d) \qquad\qquad \Gamma_{\text{ext}}.p.x \triangleq (\Gamma_{\text{ext}}.p)(x)
$$

and define the injection operations $\Gamma[\Gamma_{\text{ext}}]$ and $\tau[\Gamma_{\text{ext}}]$:

$$
\begin{array}{ll}
(\bullet)[\Gamma_{\text{ext}}] \triangleq \bullet & ((x, \tau) :: \Gamma)[\Gamma_{\text{ext}}] \triangleq (x, \tau[\Gamma_{\text{ext}}]) :: \Gamma[\Gamma_{\text{ext}}] \\
((d, \Gamma) :: \Gamma')[\Gamma_{\text{ext}}] \triangleq (d, \Gamma[\Gamma_{\text{ext}}]) :: \Gamma'[\Gamma_{\text{ext}}] & ([].p)[\Gamma_{\text{ext}}] \triangleq \Gamma_{\text{ext}}.p \\
(\iota)[\Gamma_{\text{ext}}] \triangleq \iota & (\tau_1 \to \tau_2)[\Gamma_{\text{ext}}] \triangleq \tau_1[\Gamma_{\text{ext}}] \to \tau_2[\Gamma_{\text{ext}}] \\
(\alpha)[\Gamma_{\text{ext}}] \triangleq \alpha & ([].p.x)[\Gamma_{\text{ext}}] \triangleq \Gamma_{\text{ext}}.p.x
\end{array}
$$

Let $\text{Subst} \triangleq \text{TyVar} \xrightarrow{\text{fin}} \text{Type}$ be the set of substitutions. For $S \in \text{Subst}$, define:

$$
\begin{array}{llll}
S\iota \triangleq \iota & & S(\tau_1 \to \tau_2) \triangleq S\tau_1 \to S\tau_2 \\
S\alpha \triangleq \alpha & \text{when } \alpha \notin dom(S) & S\alpha \triangleq \tau & \text{when } \alpha \mapsto \tau \in S \\
S[].\_ \triangleq [].\_ & & S\bullet \triangleq \bullet \\
S(x, \tau) :: \Gamma \triangleq (x, S\tau) :: S\Gamma & & S(d, \Gamma) :: \Gamma' \triangleq (d, S\Gamma) :: S\Gamma'
\end{array}
$$

Define:
$$(S, \Gamma_{\text{ext}}) \vDash U \triangleq \forall (\tau_1 \doteq \tau_2) \in U : (S\tau_1)[\Gamma_{\text{ext}}] = (S\tau_2)[\Gamma_{\text{ext}}]$$

Then we can show that:

Claim 1.2 (Correnctness of V). For $e \in \text{Expr}$, $\Gamma, \Gamma_{\text{ext}} \in \text{TyEnv}$, $\alpha \in \text{TyVar}$, $S \in \text{Subst}$:

$$(S, \Gamma_{\text{ext}}) \vDash U \Leftrightarrow (S\Gamma)[\Gamma_{\text{ext}}] \vdash e : (S\alpha)[\Gamma_{\text{ext}}] \qquad \text{when } V(\Gamma, e, \alpha) = U$$
$$(S, \Gamma_{\text{ext}}) \vDash U \Leftrightarrow (S\Gamma)[\Gamma_{\text{ext}}] \vdash e : (S\Gamma')[\Gamma_{\text{ext}}] \qquad \text{when } V(\Gamma, e, \alpha) = (\Gamma', U)$$

Proof sketch. Structural induction on e. □

Note that by including $[].p$ in type environments, we can naturally generate constraints about the external environment $[]$. Also, by injection, we can utilize constraints generated *in advance* to obtain constraints generated from a more informed environment. We extend injection to the output of the constraint-generating algorithm:

$$\begin{aligned}
\perp[\Gamma_{\text{ext}}] &\triangleq \perp \\
U[\Gamma_{\text{ext}}] &\triangleq \{\tau_1[\Gamma_{\text{ext}}] \doteq \tau_2[\Gamma_{\text{ext}}] | (\tau_1 \doteq \tau_2) \in U\} & \text{when all injections succeed} \\
U[\Gamma_{\text{ext}}] &\triangleq \perp & \text{when injection fails} \\
(\Gamma, U)[\Gamma_{\text{ext}}] &\triangleq (\Gamma[\Gamma_{\text{ext}}], U[\Gamma_{\text{ext}}])
\end{aligned}$$

Then we can prove:

Claim 1.3 (Advance). For $e \in \text{Expr}$, $\Gamma, \Gamma_{\text{ext}} \in \text{TyEnv}$, $\alpha \in \text{TyVar}$:

$$V(\Gamma[\Gamma_{\text{ext}}], e, \alpha) = V(\Gamma, e, \alpha)[\Gamma_{\text{ext}}]$$

Proof sketch. Structural induction on Γ. □

2 For the Language with First-Class Modules

$$\begin{array}{llll}
\text{Identifiers} & x & \in & \text{Var} \\
\text{Expression} & e & \rightarrow & x \mid \lambda x.e \mid e\ e \quad \lambda\text{-calculus} \\
& & \mid & e \bowtie e \qquad\quad \text{linked expression} \\
& & \mid & \varepsilon \qquad\qquad\ \text{empty module} \\
& & \mid & \texttt{val}\ x\ e\ e \quad\ \text{value binding}
\end{array}$$

Figure 7: Abstract syntax of the language where modules are first-class.

2.1 Operational Semantics

$$\begin{array}{llll}
\text{Environment/Context} & \sigma & \in & \text{Ctx} \\
\text{Value} & v & \in & \text{Val} \triangleq \text{Ctx} + \text{Var} \times \text{Expr} \times \text{Ctx} \\
\text{Context} & \sigma & \rightarrow & \bullet \qquad\qquad\quad \text{empty stack} \\
& & \mid & (x, v) :: \sigma \qquad \text{expression binding} \\
\text{Value} & v & \rightarrow & \sigma \qquad\qquad\quad \text{module} \\
& & \mid & \langle \lambda x.e, \sigma \rangle \qquad \text{closure}
\end{array}$$

Figure 8: Definition of the semantic domains.

2.2 Typing

The definitions for types are in Figure 10 and the typing rules are in Figure 11. The definitions for subtyping are in Figure 12.

$$\boxed{(e, \sigma) \Downarrow v}$$

$$[\text{ID}] \ \frac{v = \sigma(x)}{(x, \sigma) \Downarrow v} \qquad [\text{FN}] \ \frac{}{(\lambda x.e, \sigma) \Downarrow \langle \lambda x.e, \sigma \rangle} \qquad [\text{APP}] \ \frac{\begin{array}{c} (e_1, \sigma) \Downarrow \langle \lambda x.e_\lambda, \sigma_\lambda \rangle \\ (e_2, \sigma) \Downarrow v \\ (e_\lambda, (x, v) :: \sigma_\lambda) \Downarrow v' \end{array}}{(e_1\, e_2, \sigma) \Downarrow v'} \qquad [\text{LINK}] \ \frac{\begin{array}{c} (e_1, \sigma) \Downarrow \sigma' \\ (e_2, \sigma') \Downarrow v \end{array}}{(e_1 \bowtie e_2, \sigma) \Downarrow v}$$

$$[\text{EMPTY}] \ \frac{}{(\varepsilon, \sigma) \Downarrow \bullet} \qquad [\text{BIND}] \ \frac{\begin{array}{c} (e_1, \sigma) \Downarrow v \\ (e_2, (x, v) :: \sigma) \Downarrow \sigma' \end{array}}{(\texttt{val } x\, e_1\, e_2, \sigma) \Downarrow (x, v) :: \sigma'}$$

Figure 9: The big-step operational semantics.

$$\begin{array}{llll} \text{Types} & \tau & \to & \Gamma & \text{module type} \\ & & | & \tau \to \tau & \text{function type} \\ \text{Typing Environment} & \Gamma & \to & \bullet & \text{empty environment} \\ & & | & (x, \tau) :: \Gamma & \text{binding} \end{array}$$

Figure 10: Definition of types.

$$\boxed{\Gamma \vdash e : \tau}$$

$$[\text{ID}] \ \frac{\tau = \Gamma(x)}{\Gamma \vdash x : \tau} \qquad [\text{FN}] \ \frac{(x, \tau_1) :: \Gamma \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2} \qquad [\text{APP}] \ \frac{\begin{array}{c} \Gamma \vdash e_1 : \tau_1 \to \tau \\ \Gamma \vdash e_2 : \tau_2 \\ \tau_1 \leq \tau_2 \end{array}}{\Gamma \vdash e_1\, e_2 : \tau} \qquad [\text{LINK}] \ \frac{\begin{array}{c} \Gamma \vdash e_1 : \Gamma_1 \\ \Gamma_1 \vdash e_2 : \tau_2 \end{array}}{\Gamma \vdash e_1 \bowtie e_2 : \tau_2}$$

$$[\text{EMPTY}] \ \frac{}{\Gamma \vdash \varepsilon : \bullet} \qquad [\text{BIND}] \ \frac{\begin{array}{c} \Gamma \vdash e_1 : \tau_1 \\ (x, \tau_1) :: \Gamma \vdash e_2 : \Gamma_2 \end{array}}{\Gamma \vdash \texttt{val } x\, e_1\, e_2 : (x, \tau_1) :: \Gamma_2}$$

Figure 11: The typing judgment.

$$\boxed{\tau \leq \tau}$$

$$[\text{EMPTY}] \ \frac{}{\bullet \leq \bullet} \qquad [\text{BIND}] \ \frac{\begin{array}{c} \Gamma(x) \leq \tau \\ \Gamma - x \leq \Gamma' \end{array}}{\Gamma \leq (x, \tau) :: \Gamma'} \qquad [\text{FN}] \ \frac{\begin{array}{c} \tau_2 \leq \tau_1 \\ \tau_1' \leq \tau_2' \end{array}}{\tau_1 \to \tau_1' \leq \tau_2 \to \tau_2'}$$

Figure 12: The subtype relation.

2.3 Type Safety

Claim 2.1 (Type Safety). For all $e \in$ Expr, if $\bullet \vdash e : \tau$ for some τ, then there exists some $v \in$ Val such that $(e, \bullet) \Downarrow v$.

Proof sketch. We prove this through unary logical relations and induction on the typing judgment.

Value Relation $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \boxed{\mathcal{V}[\![\tau]\!]}$

$$\begin{array}{rcl} \mathcal{V}[\![\bullet]\!] & \triangleq & \text{Ctx} \\ \mathcal{V}[\![(x, \tau) :: \Gamma]\!] & \triangleq & \{\sigma \,|\, \sigma(x) \in \mathcal{V}[\![\tau]\!] \land (\sigma - x) \in \mathcal{V}[\![\Gamma - x]\!]\} \\ \mathcal{V}[\![\tau_1 \to \tau_2]\!] & \triangleq & \{\langle \lambda x.e, \sigma \rangle \,|\, \forall v \in \mathcal{V}[\![\tau_1]\!], (e, (x, v) :: \sigma) \in \mathcal{E}[\![\tau_2]\!]\} \end{array}$$

Expression Relation $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \boxed{\mathcal{E}[\![\tau]\!]}$

$$\mathcal{E}[\![\tau]\!] \quad \triangleq \quad \{(e, \sigma) \,|\, \exists v \in \mathcal{V}[\![\tau]\!] : (e, \sigma) \Downarrow v\}$$

Semantic Typing $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \boxed{\Gamma \vDash e : \tau}$

$$\Gamma \vDash e : \tau \quad \triangleq \quad \forall \sigma \in \mathcal{C}[\![\Gamma]\!] : (e, \sigma) \in \mathcal{E}[\![\tau]\!]$$

We want to prove that:

$$\Gamma \vdash e : \tau \Rightarrow \Gamma \vDash e : \tau$$

by induction on $\vdash$.

For the base cases of $\bullet$, the proof is trivial. For inductive cases, we need to show *compatibility* lemmas. That is, we must show that the typing rules for syntactic typing hold for semantic typing as well. For this, we need the *subtyping* lemma:

$$\tau_1 \leq \tau_2 \Rightarrow \mathcal{V}[\![\tau_2]\!] \subseteq \mathcal{V}[\![\tau_1]\!]$$

Then by the inductive hypothesis and compatibility, the result follows. $\qquad\square$

2.4 Type Inference

When modules are first-class, type variables can go in the place of type environments.

First we define the syntax for type constraints.

$$
\begin{array}{rcll}
\text{Type Variable} & \alpha & \in & \text{TyVar} \\
\text{Path} & p & \rightarrow & \epsilon & \text{empty string} \\
& & | & px & \text{concatenation with identifier} \\
\text{Types} & \tau & \rightarrow & \Gamma \mid \tau \rightarrow \tau & \text{module/function types} \\
\text{Type Environment} & \Gamma & \rightarrow & \bullet & \text{empty environment} \\
& & | & (x, \tau) :: \Gamma & \text{binding} \\
& & | & \alpha.p & \text{type variable} \\
& & | & [].p & \text{type from the external environment} \\
\text{Type Constraint} & u & \rightarrow & \tau \doteq \tau & \text{equality constraint} \\
& & | & \tau \mathrel{\dot{\leq}} \tau & \text{subtyping constraint} \\
\text{Set of Constraints} & U & \subseteq & \{u | u \text{ type constraint}\}
\end{array}
$$

Figure 13: Definition of type constraints.

Next we define the type access operation $\Gamma(x)$:

$$
\begin{aligned}
\bullet(x) &\triangleq \bot \\
((x, \tau) :: \_)(x) &\triangleq \tau \\
((x', \_) :: \Gamma)(x) &\triangleq \Gamma(x) \qquad (x' \neq x) \\
(\alpha.p)(x) &\triangleq \alpha.px \\
([].p)(x) &\triangleq [].px
\end{aligned}
$$

Now we can define the constraint generation algorithm $V(\Gamma, e, \alpha)$. Note that the **let** $U = \_$ **in** $\_$ notation returns $\bot$ if the right hand side is not a set of type constraints. Also note that we write α for $\alpha.\epsilon$ as well.

$$\boxed{V(\Gamma, e, \alpha) = U \text{ or } \bot}$$

$$
\begin{aligned}
V(\Gamma, \varepsilon, \alpha) &\triangleq \{\alpha \doteq \bullet\} \\
V(\Gamma, x, \alpha) &\triangleq \textbf{let } \tau = \Gamma(x) \textbf{ in} \\
& \quad \{\alpha \doteq \tau\} \\
V(\Gamma, \lambda x.e, \alpha) &\triangleq \textbf{let } \alpha_1, \alpha_2 = \textit{fresh} \textbf{ in} \\
& \quad \textbf{let } U = V((x, \alpha_1) :: \Gamma, e, \alpha_2) \textbf{ in} \\
& \quad \{\alpha \doteq \alpha_1 \rightarrow \alpha_2\} \cup U \\
V(\Gamma, e_1\, e_2, \alpha) &\triangleq \textbf{let } \alpha_1, \alpha_2, \alpha_3 = \textit{fresh} \textbf{ in} \\
& \quad \textbf{let } U_1 = V(\Gamma, e_1, \alpha_1) \textbf{ in} \\
& \quad \textbf{let } U_2 = V(\Gamma, e_2, \alpha_2) \textbf{ in} \\
& \quad \{\alpha_1 \doteq \alpha_3 \rightarrow \alpha, \alpha_3 \mathrel{\dot{\leq}} \alpha_2\} \cup U_1 \cup U_2
\end{aligned}
$$

$$
\begin{aligned}
V(\Gamma, e_1 \bowtie e_2, \alpha) &\triangleq \textbf{let } \alpha_1 = \textit{fresh} \textbf{ in} \\
& \quad \textbf{let } U_1 = V(\Gamma, e_1, \alpha_1) \textbf{ in} \\
& \quad \textbf{let } U_2 = V(\alpha_1, e_2, \alpha) \textbf{ in} \\
& \quad U_1 \cup U_2 \\
V(\Gamma, \texttt{val } d\, e_1\, e_2, \alpha) &\triangleq \textbf{let } \alpha_1, \alpha_2 = \textit{fresh} \textbf{ in} \\
& \quad \textbf{let } U_1 = V(\Gamma, e_1, \alpha_1) \textbf{ in} \\
& \quad \textbf{let } U_2 = V((x, \alpha_1) :: \Gamma, e_2, \alpha_2) \textbf{ in} \\
& \quad \{\alpha \doteq (x, \alpha_1) :: \alpha_2\} \cup U_1 \cup U_2
\end{aligned}
$$

We want to prove that the constraint generation algorithm is correct.

First, for $\Gamma_{\text{ext}} \in \text{TyEnv}$, define the access operation $\Gamma_{\text{ext}}.p$ (which may fail):

$$\Gamma_{\text{ext}}.\epsilon \triangleq \Gamma_{\text{ext}} \qquad\qquad \Gamma_{\text{ext}}.px \triangleq (\Gamma_{\text{ext}}.p)(x)$$

and define the injection operations $\Gamma[\Gamma_{\text{ext}}]$ and $\tau[\Gamma_{\text{ext}}]$:

$$
\begin{aligned}
(\bullet)[\Gamma_{\text{ext}}] &\triangleq \bullet & ((x, \tau) :: \Gamma)[\Gamma_{\text{ext}}] &\triangleq (x, \tau[\Gamma_{\text{ext}}]) :: \Gamma[\Gamma_{\text{ext}}] \\
(\alpha.p)[\Gamma_{\text{ext}}] &\triangleq \alpha.p & ([].p)[\Gamma_{\text{ext}}] &\triangleq \Gamma_{\text{ext}}.p
\end{aligned}
$$

Let Subst $\triangleq$ TyVar $\xrightarrow{\text{fin}}$ Type be the set of substitutions. For $S \in$ Subst, define:

$$S\bullet \triangleq \bullet \qquad\qquad\qquad\qquad\qquad S(\tau_1 \to \tau_2) \triangleq S\tau_1 \to S\tau_2$$
$$S(\alpha.p) \triangleq \alpha.p \qquad \text{when } \alpha \notin dom(S) \qquad S(\alpha.p) \triangleq \Gamma.p \qquad \text{when } \alpha \mapsto \Gamma \in S$$
$$S(\alpha.p) \triangleq \bot \qquad \text{when } \alpha \mapsto \tau \in S \text{ and } \tau \notin \text{TyEnv} \qquad S([].p) \triangleq [].p$$

Define:

$$(S, \Gamma_{\text{ext}}) \vDash U \triangleq \forall (\tau_1 \doteq \tau_2) \in U : (S\tau_1)[\Gamma_{\text{ext}}] = (S\tau_2)[\Gamma_{\text{ext}}]$$
$$\text{and}$$
$$\forall (\tau_1 \dot{\leq} \tau_2) \in U : (S\tau_1)[\Gamma_{\text{ext}}] \leq (S\tau_2)[\Gamma_{\text{ext}}]$$

where subtyping between type variables are defined as...(Work in Progress)

Then we can show that:

Claim 2.2 (Correnctness of V)**.** For $e \in$ Expr, $\Gamma, \Gamma_{\text{ext}} \in$ TyEnv, $\alpha \in$ TyVar, $S \in$ Subst:

$$(S, \Gamma_{\text{ext}}) \vDash U \Leftrightarrow (S\Gamma)[\Gamma_{\text{ext}}] \vdash e : (S\alpha)[\Gamma_{\text{ext}}] \qquad\qquad \text{when } V(\Gamma, e, \alpha) = U$$

Proof sketch. Structural induction on e. $\qquad\square$

Note that by including $[].p$ in type environments, we can naturally generate constraints about the external environment $[]$. Also, by injection, we can utilize constraints generated *in advance* to obtain constraints generated from a more informed environment. We extend injection to the output of the constraint-generating algorithm:

$$\bot[\Gamma_{\text{ext}}] \triangleq \bot$$
$$U[\Gamma_{\text{ext}}] \triangleq \{\tau_1[\Gamma_{\text{ext}}] \doteq \tau_2[\Gamma_{\text{ext}}] | (\tau_1 \doteq \tau_2) \in U\} \cup \{\tau_1[\Gamma_{\text{ext}}] \dot{\leq} \tau_2[\Gamma_{\text{ext}}] | (\tau_1 \dot{\leq} \tau_2) \in U\} \quad \text{when all injections succeed}$$
$$U[\Gamma_{\text{ext}}] \triangleq \bot \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{when injection fails}$$

Then we can prove:

Claim 2.3 (Advance)**.** For $e \in$ Expr, $\Gamma, \Gamma_{\text{ext}} \in$ TyEnv, $\alpha \in$ TyVar:

$$V(\Gamma[\Gamma_{\text{ext}}], e, \alpha) = V(\Gamma, e, \alpha)[\Gamma_{\text{ext}}]$$

Proof sketch. Structural induction on Γ. $\qquad\square$