Type Inference as an Instance for Modular Analysis

Joonhyup Lee

November 29, 2023

1 For the Simple Module Language

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

Figure 1: Abstract syntax of the simple module language.

1.1 Operational Semantics

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

Figure 2: Definition of the semantic domains.

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

$$[\text{Unit}]\ \frac{}{((), \sigma) \Downarrow ()} \qquad [\text{ExprID}]\ \frac{v = \sigma(x)}{(x, \sigma) \Downarrow v} \qquad [\text{Fn}]\ \frac{}{(\lambda x.e, \sigma) \Downarrow \langle \lambda x.e, \sigma \rangle} \qquad [\text{App}]\ \frac{\begin{array}{c}(e_1, \sigma) \Downarrow \langle \lambda x.e_\lambda, \sigma_\lambda \rangle \\ (e_2, \sigma) \Downarrow v \\ (e_\lambda, (x, v) :: \sigma_\lambda) \Downarrow v'\end{array}}{(e_1\ e_2, \sigma) \Downarrow v'} \qquad [\text{Link}]\ \frac{\begin{array}{c}(m_1, \sigma) \Downarrow \sigma' \\ (e_2, \sigma') \Downarrow v\end{array}}{(m_1 \rtimes e_2, \sigma) \Downarrow v}$$

$$[\text{Empty}]\ \frac{}{(\varepsilon, \sigma) \Downarrow \bullet} \qquad [\text{ModID}]\ \frac{\sigma' = \sigma(d)}{(d, \sigma) \Downarrow \sigma'} \qquad [\text{LetE}]\ \frac{\begin{array}{c}(e_1, \sigma) \Downarrow v \\ (m_2, (x, v) :: \sigma) \Downarrow \sigma'\end{array}}{(\texttt{val}\ x\ e_1\ m_2, \sigma) \Downarrow (x, v) :: \sigma'} \qquad [\text{LetM}]\ \frac{\begin{array}{c}(m_1, \sigma) \Downarrow \sigma' \\ (m_2, (d, \sigma') :: \sigma) \Downarrow \sigma''\end{array}}{(\texttt{mod}\ d\ m_1\ m_2, \sigma) \Downarrow (d, \sigma') :: \sigma''}$$

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 $\bullet \vdash m : \Gamma$ for some Γ, then there exists some $\sigma \in \text{Ctx}$ such that $(m, \bullet) \Downarrow \sigma$.

$$
\begin{array}{llcll}
\text{Types} & \tau & \rightarrow & \iota & \text{unit type} \\
 & & \mid & \tau \rightarrow \tau & \text{function type} \\
\text{Type Environment} & \Gamma & \rightarrow & \bullet & \text{empty environment} \\
 & & \mid & (x, \tau) :: \Gamma & \text{value binding} \\
 & & \mid & (d, \Gamma) :: \Gamma & \text{module binding}
\end{array}
$$

Figure 4: Definition of types.

$$\boxed{\Gamma \vdash e : \tau \text{ and } \Gamma \vdash m : \Gamma}$$

$$
[\textsc{Unit}]\ \frac{}{\Gamma \vdash () : \iota} \qquad
[\textsc{ExprID}]\ \frac{\tau = \Gamma(x)}{\Gamma \vdash x : \tau} \qquad
[\textsc{Fn}]\ \frac{(x, \tau_1) :: \Gamma \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \rightarrow \tau_2} \qquad
[\textsc{App}]\ \frac{\Gamma \vdash e_1 : \tau' \rightarrow \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\, e_2 : \tau} \qquad
[\textsc{Link}]\ \frac{\Gamma \vdash m_1 : \Gamma_1 \quad \Gamma_1 \vdash e_2 : \tau_2}{\Gamma \vdash m_1 \rtimes e_2 : \tau_2}
$$

$$
[\textsc{Empty}]\ \frac{}{\Gamma \vdash \varepsilon : \bullet} \qquad
[\textsc{ModID}]\ \frac{\Gamma' = \Gamma(d)}{\Gamma \vdash d : \Gamma'} \qquad
[\textsc{LetE}]\ \frac{\Gamma \vdash e_1 : \tau_1 \quad (x, \tau_1) :: \Gamma \vdash m_2 : \Gamma_2}{\Gamma \vdash \texttt{val}\, x\, e_1\, m_2 : (x, \tau_1) :: \Gamma_2} \qquad
[\textsc{LetM}]\ \frac{\Gamma \vdash m_1 : \Gamma_1 \quad (d, \Gamma_1) :: \Gamma \vdash m_2 : \Gamma_2}{\Gamma \vdash \texttt{mod}\, d\, m_1\, m_2 : (d, \Gamma_1) :: \Gamma_2}
$$

Figure 5: The typing judgment.

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

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

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

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

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

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

$$
\begin{array}{rcl}
\mathcal{C}[\![\bullet]\!] & \triangleq & \{\bullet\} \\
\mathcal{C}[\![(x, \tau_1) :: \Gamma_2]\!] & \triangleq & \{(x, v_1) :: \sigma_2 | v_1 \in \mathcal{V}[\![\tau_1]\!] \wedge \sigma_2 \in \mathcal{C}[\![\Gamma_2]\!]\} \\
\mathcal{C}[\![(d, \Gamma_1) :: \Gamma_2]\!] & \triangleq & \{(d, \sigma_1) :: \sigma_2 | \sigma_1 \in \mathcal{C}[\![\Gamma_1]\!] \wedge \sigma_2 \in \mathcal{C}[\![\Gamma_2]\!]\}
\end{array}
$$

Semantic Typing $\boxed{\Gamma \vDash e : \tau \text{ and } \Gamma \vDash m : \Gamma}$

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

We want to prove that:

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

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

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. □

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.

Type Variable	α	$\in$	TyVar	
Module Path	p	$\rightarrow$	ϵ	empty string
		$\mid$	pd	concatenation with module identifier
Types	τ	$\rightarrow$	$\iota \mid \tau \rightarrow \tau$	unit/function types
		$\mid$	α	type variables
		$\mid$	$[].p.x$	types from the external environment
Type Environment	Γ	$\rightarrow$	$\bullet$	empty environment
		$\mid$	$(x, \tau) :: \Gamma$	value binding
		$\mid$	$(d, \Gamma) :: \Gamma$	module binding
		$\mid$	$[].p$	modules from the external environment
Type Constraint	u	$\rightarrow$	$\tau \doteq \tau$	equality constraint
Set of Constraints	U	$\subseteq$	$\{u \mid u \text{ type constraint}\}$	

Figure 6: Definition of type constraints.

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

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

Now we can define the constraint generation algorithms $V_1(\Gamma, e, \alpha)$ and $V_2(\Gamma, m)$. Note that the **let** U = _ **in** _ notation returns $\bot$ if the right hand side is $\bot$. Likewise, the **let** (Γ, U) = _ **in** _ notation returns $\bot$ if the right hand side is $\bot$.

$$\boxed{V_1(\Gamma, e, \alpha) = U \text{ and } V_2(\Gamma, m) = (\Gamma, U)}$$

$$\begin{array}{rcl}
V_1(\Gamma, (), \alpha) & \triangleq & \{\alpha \doteq \iota\} \\
V_1(\Gamma, x, \alpha) & \triangleq & \textbf{let } \tau = \Gamma(x) \textbf{ in} \\
 & & \{\alpha \doteq \tau\} \\
V_1(\Gamma, \lambda x.e, \alpha) & \triangleq & \textbf{let } \alpha_1, \alpha_2 = \textit{fresh} \textbf{ in} \\
 & & \textbf{let } U = V_1((x, \alpha_1) :: \Gamma, e, \alpha_2) \textbf{ in} \\
 & & \{\alpha \doteq \alpha_1 \rightarrow \alpha_2\} \cup U \\
V_1(\Gamma, e_1\, e_2, \alpha) & \triangleq & \textbf{let } \alpha_1, \alpha_2 = \textit{fresh} \textbf{ in} \\
 & & \textbf{let } U_1 = V_1(\Gamma, e_1, \alpha_1) \textbf{ in} \\
 & & \textbf{let } U_2 = V_1(\Gamma, e_2, \alpha_2) \textbf{ in} \\
 & & \{\alpha_1 \doteq \alpha_2 \rightarrow \alpha\} \cup U_1 \cup U_2 \\
V_1(\Gamma, m_1 \rtimes e_2, \alpha) & \triangleq & \textbf{let } (\Gamma_1, U_1) = V_2(\Gamma, m_1) \textbf{ in} \\
 & & \textbf{let } U_2 = V_1(\Gamma_1, e_2, \alpha) \textbf{ in} \\
 & & U_1 \cup U_2
\end{array}$$

$$\begin{array}{rcl}
V_2(\Gamma, \varepsilon) & \triangleq & (\bullet, \emptyset) \\
V_2(\Gamma, d) & \triangleq & \textbf{let } \Gamma' = \Gamma(d) \textbf{ in} \\
 & & (\Gamma', \emptyset) \\
V_2(\Gamma, \texttt{val}\, x\, e_1\, m_2) & \triangleq & \textbf{let } \alpha_1 = \textit{fresh} \textbf{ in} \\
 & & \textbf{let } U_1 = V_1(\Gamma, e_1, \alpha_1) \textbf{ in} \\
 & & \textbf{let } (\Gamma_2, U_2) = V_2((x, \alpha_1) :: \Gamma, m_2) \textbf{ in} \\
 & & ((x, \alpha_1) :: \Gamma_2, U_1 \cup U_2) \\
V_2(\Gamma, \texttt{mod}\, d\, m_1\, m_2) & \triangleq & \textbf{let } (\Gamma_1, U_1) = V_2(\Gamma, m_1) \textbf{ in} \\
 & & \textbf{let } (\Gamma_2, U_2) = V_2((d, \Gamma_1) :: \Gamma, m_2) \textbf{ in} \\
 & & ((d, \Gamma_1) :: \Gamma_2, U_1 \cup U_2)
\end{array}$$

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):

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

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

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

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

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

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 $e \in$ Expr, $m \in$ Module, $\Gamma, \Gamma_{\text{ext}} \in$ TyEnv, $\alpha \in$ TyVar, $S \in$ Subst:

$$\begin{aligned} (S, \Gamma_{\text{ext}}) \vDash U &\Leftrightarrow (S\Gamma)[\Gamma_{\text{ext}}] \vdash e : (S\alpha)[\Gamma_{\text{ext}}] && \text{when } V_1(\Gamma, e, \alpha) = U \\ (S, \Gamma_{\text{ext}}) \vDash U &\Leftrightarrow (S\Gamma)[\Gamma_{\text{ext}}] \vdash m : (S\Gamma')[\Gamma_{\text{ext}}] && \text{when } V_2(\Gamma, m) = (\Gamma', U) \end{aligned}$$

Proof sketch. Mutual induction on e, m. □

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:

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

Then we can prove:

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

$$\begin{aligned} V_1(\Gamma[\Gamma_{\text{ext}}], e, \alpha) &= V_1(\Gamma, e, \alpha)[\Gamma_{\text{ext}}] \\ V_2(\Gamma[\Gamma_{\text{ext}}], m) &= V_2(\Gamma, m)[\Gamma_{\text{ext}}] \end{aligned}$$

Proof sketch. Structural induction on Γ. □

2 For the Language with First-Class Modules

Identifiers	x	$\in$	Var	
Expression	e	$\rightarrow$	$x \mid \lambda x.e \mid e\ e$	λ-calculus
		$\mid$	$e \rtimes\!\!\!\ltimes e$	linked expression
		$\mid$	ε	empty module
		$\mid$	$\texttt{val}\ x\ e\ e$	value binding

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

2.1 Operational Semantics

Environment/Context	σ	$\in$	Ctx	
Value	v	$\in$	Val $\triangleq$ Ctx + Var $\times$ Expr $\times$ Ctx	
Context	σ	$\rightarrow$	$\bullet$	empty stack
		$\mid$	$(x, v) :: \sigma$	value binding
Value	v	$\rightarrow$	σ	exported context
		$\mid$	$\langle \lambda x.e, \sigma \rangle$	closure

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.

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

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

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

Figure 9: The big-step operational semantics.

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

Figure 10: Definition of types.

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

$$[\textsc{Id}]\ \frac{\tau=\Gamma(x)}{\Gamma\vdash x:\tau}\qquad [\textsc{Fn}]\ \frac{(x,\tau_1)::\Gamma\vdash e:\tau_2}{\Gamma\vdash\lambda x.e:\tau_1\rightarrow\tau_2}\qquad [\textsc{App}]\ \frac{\begin{array}{c}\Gamma\vdash e_1:\tau_1\rightarrow\tau\\ \Gamma\vdash e_2:\tau_2\\ \tau_1\leq\tau_2\end{array}}{\Gamma\vdash e_1\ e_2:\tau}\qquad [\textsc{Link}]\ \frac{\begin{array}{c}\Gamma\vdash e_1:\Gamma_1\\ \Gamma_1\vdash e_2:\tau_2\end{array}}{\Gamma\vdash e_1\rtimes e_2:\tau_2}$$

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

Figure 11: The typing judgment.

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

$$[\textsc{Empty}]\ \frac{}{\bullet\leq\bullet}\qquad [\textsc{Bind}]\ \frac{\begin{array}{c}\Gamma(x)\leq\tau\\ \Gamma-x\leq\Gamma'\end{array}}{\Gamma\leq(x,\tau)::\Gamma'}\qquad [\textsc{Fn}]\ \frac{\begin{array}{c}\tau_2\leq\tau_1\\ \tau_1'\leq\tau_2'\end{array}}{\tau_1\rightarrow\tau_1'\leq\tau_2\rightarrow\tau_2'}$$

Figure 12: The subtype relation.

2.3 Type Safety

Claim 2.1 (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 $(e,\bullet)\Downarrow v$.

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

Value Relation $\boxed{\mathcal{V}\llbracket\tau\rrbracket}$

$$\begin{array}{rcl}
\mathcal{V}\llbracket\bullet\rrbracket & \triangleq & \text{Ctx}\\
\mathcal{V}\llbracket(x,\tau)::\Gamma\rrbracket & \triangleq & \{\sigma\mid\sigma(x)\in\mathcal{V}\llbracket\tau\rrbracket\wedge(\sigma-x)\in\mathcal{V}\llbracket\Gamma-x\rrbracket\}\\
\mathcal{V}\llbracket\tau_1\rightarrow\tau_2\rrbracket & \triangleq & \{\langle\lambda x.e,\sigma\rangle\mid\forall v\in\mathcal{V}\llbracket\tau_1\rrbracket:(e,(x,v)::\sigma)\in\mathcal{E}\llbracket\tau_2\rrbracket\}
\end{array}$$

Expression Relation $\boxed{\mathcal{E}\llbracket\tau\rrbracket}$

$$\mathcal{E}\llbracket\tau\rrbracket\ \triangleq\ \{(e,\sigma)\mid\exists v\in\mathcal{V}\llbracket\tau\rrbracket:(e,\sigma)\Downarrow v\}$$

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

$$\Gamma\vDash e:\tau\ \triangleq\ \forall\sigma\in\mathcal{V}\llbracket\Gamma\rrbracket:(e,\sigma)\in\mathcal{E}\llbracket\tau\rrbracket$$

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. $\square$

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.

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

Figure 13: Definition of type constraints.

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

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

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.

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

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

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

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

First, for $\tau \in$ Type, define the access operation $\tau.p$ (which may fail):

$$\tau.\epsilon \triangleq \tau \qquad \tau.px \triangleq (\tau.p)(x) \quad \text{when } \tau.p \neq \bot \qquad \tau.px \triangleq \bot \quad \text{when } \tau.p = \bot$$

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

$$\begin{aligned} (\bullet)[\Gamma_{\text{ext}}] &\triangleq \bullet & ((x,\tau) :: \Gamma)[\Gamma_{\text{ext}}] &\triangleq (x, \tau[\Gamma_{\text{ext}}]) :: \Gamma[\Gamma_{\text{ext}}] \\ (\alpha.p)[\Gamma_{\text{ext}}] &\triangleq \alpha.p & ([].p)[\Gamma_{\text{ext}}] &\triangleq \Gamma_{\text{ext}}.p \\ (\tau_1 \rightarrow \tau_2)[\Gamma_{\text{ext}}] &\triangleq \tau_1[\Gamma_{\text{ext}}] \rightarrow \tau_2[\Gamma_{\text{ext}}] \end{aligned}$$

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

$$\begin{aligned} S\bullet &\triangleq \bullet & S(\tau_1 \rightarrow \tau_2) &\triangleq S\tau_1 \rightarrow S\tau_2 \\ S(\alpha.p) &\triangleq \alpha.p \quad \text{when } \alpha \notin \mathit{dom}(S) & S(\alpha.p) &\triangleq \tau.p \quad \text{when } \alpha \mapsto \tau \in S \\ S([].p) &\triangleq [].p \end{aligned}$$

Define:

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

where subtyping rules are the same as Figure 12 and subtyping between type variables are not defined.

Then we can show that:

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

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

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:

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

Then we can prove:

Claim 2.3 (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. □