Modular Analysis

Joonhyup Lee

January 24, 2024

1 Syntax and Semantics

1.1 Abstract Syntax

$$
\begin{array}{rcll}
\text{Identifiers} & x & \in & \text{Var} \\
\text{Expression} & e & \to & x \mid \lambda x.e \mid e\ e & \lambda\text{-calculus} \\
 & & \mid & e \rtimes e & \text{linked expression} \\
 & & \mid & \varepsilon & \text{empty module} \\
 & & \mid & x = e\ ;\ e & \text{binding}
\end{array}
$$

Figure 1: Abstract syntax of the language.

1.2 Operational Semantics

$$
\begin{array}{rcll}
\text{Environment} & \sigma & \in & \text{Env} \\
\text{Location} & \ell & \in & \text{Loc} \\
\text{de Bruijn Index} & n & \in & \mathbb{N} \\
\text{Value} & v & \in & \text{Val} \triangleq \text{Env} + \text{Var} \times \text{Expr} \times \text{Env} \\
\text{Weak Value} & w & \in & \text{WVal} \triangleq \text{Val} + \underline{\text{Val}} \\
\text{Environment} & \sigma & \to & \bullet & \text{empty stack} \\
 & & \mid & (x, w) :: \sigma & \text{weak value binding} \\
 & & \mid & (x, \ell) :: \sigma & \text{free location binding} \\
 & & \mid & (x, n) :: \sigma & \text{bound location binding} \\
\text{Value} & v & \to & \sigma & \text{exported environment} \\
 & & \mid & \langle \lambda x.e, \sigma \rangle & \text{closure} \\
\text{Weak Value} & w & \to & v & \text{value} \\
 & & \mid & \mu.v & \text{recursive value}
\end{array}
$$

Figure 2: Definition of the semantic domains.

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

$$
\textsc{Id}\ \frac{v = \sigma(x)}{(x, \sigma) \Downarrow v}
\qquad
\textsc{RecId}\ \frac{\mu.v = \sigma(x)}{(x, \sigma) \Downarrow v^{\mu.v}}
\qquad
\textsc{Fn}\ \frac{}{(\lambda x.e, \sigma) \Downarrow \langle \lambda x.e, \sigma \rangle}
\qquad
\textsc{App}\ \frac{(e_1, \sigma) \Downarrow \langle \lambda x.e, \sigma_1 \rangle \quad (e_2, \sigma) \Downarrow v_2 \quad (e, (x, v_2) :: \sigma_1) \Downarrow v}{(e_1\ e_2, \sigma) \Downarrow v}
$$

$$
\textsc{Link}\ \frac{(e_1, \sigma) \Downarrow \sigma_1 \quad (e_2, \sigma_1) \Downarrow v}{(e_1 \rtimes e_2, \sigma) \Downarrow v}
\qquad
\textsc{Empty}\ \frac{}{(\varepsilon, \sigma) \Downarrow \bullet}
\qquad
\textsc{Bind}\ \frac{\ell \notin \text{FLoc}(\sigma) \quad (e_1, (x, \ell) :: \sigma) \Downarrow v_1 \quad (e_2, (x, \mu.^{\setminus\ell} v_1) :: \sigma) \Downarrow \sigma_2}{(x = e_1; e_2, \sigma) \Downarrow (x, \mu.^{\setminus\ell} v_1) :: \sigma_2}
$$

Figure 3: The big-step operational semantics.

We use the locally nameless representation, and enforce that all values be *locally closed*. As a consequence, the big-step operational semantics will be *deterministic*, no matter what ℓ is chosen in the Bind rule.

$$
\begin{array}{rcll}
\boxed{e, \sigma, K \to e, \sigma, K} & & & \\
e_1\ e_2, \sigma, K & \to & e_1, \sigma, K \circ (\_\ (e_2, \sigma)) & \\
e_1 \rtimes e_2, \sigma, K & \to & e_1, \sigma, K \circ (\_ \rtimes e_2) & \\
x = e_1; e_2, \sigma, K & \to & e_1, (x, \ell) :: \sigma, K \circ (x = \ell; (e_2, \sigma)) & \ell \notin \mathrm{FLoc}(\sigma) \\[1em]
\boxed{v, K \to e, \sigma, K} & & & \\
\langle \lambda x.e, \sigma_1 \rangle, K \circ (\_\ (e_2, \sigma)) & \to & e_2, \sigma, K \circ (\langle \lambda x.e, \sigma_1 \rangle\ \_) & \\
\sigma_1, K \circ (\_ \rtimes e_2) & \to & e_2, \sigma_1, K & \\
v_1, K \circ (x = \ell; (e_2, \sigma)) & \to & e_2, (x, \mu.{}^{\setminus \ell}v_1) :: \sigma, K \circ (x = \mu.{}^{\setminus \ell}v_1; \_) & \\
v_2, K \circ (\langle \lambda x.e, \sigma_1 \rangle\ \_) & \to & e, (x, v_2) :: \sigma_1, K & \\[1em]
\boxed{v, K \to v, K} & & & \\
\sigma_2, K \circ (x = w_1; \_) & \to & (x, w_1) :: \sigma_2, K & \\[1em]
\boxed{e, \sigma, K \to v, K} & & & \\
x, \sigma, K & \to & v, K & v = \sigma(x) \\
x, \sigma, K & \to & v^{\mu.v}, K & \mu.v = \sigma(x) \\
\lambda x.e, \sigma, K & \to & \langle \lambda x.e, \sigma \rangle, K & \\
\varepsilon, \sigma, K & \to & \bullet, K &
\end{array}
$$

Figure 4: The equivalent small-step operational semantics.

1.3 Adding Memory

The first step towards abstraction is reformulating the semantics into a version with memory.

$$
\begin{array}{rccll}
\text{Environment} & \sigma & \in & \mathrm{Env} & \\
\text{Location} & \ell & \in & \mathrm{Loc} & \\
\text{Memory} & m & \in & \mathrm{Mem} \triangleq \mathrm{Loc} \xrightarrow{\text{fin}} \mathrm{Val} & \\
\text{Value} & v & \in & \mathrm{Val} \triangleq \mathrm{Env} + \mathrm{Var} \times \mathrm{Expr} \times \mathrm{Env} & \\
\text{Environment} & \sigma & \to & \bullet & \text{empty stack} \\
 & & \mid & (x, \ell) :: \sigma & \text{location binding} \\
\text{Value} & v & \to & \sigma & \text{exported environment} \\
 & & \mid & \langle \lambda x.e, \sigma \rangle & \text{closure}
\end{array}
$$

Figure 5: Definition of the semantic domains with memory.

$$
\begin{array}{rcll}
\boxed{e, \sigma, m, K \to e, \sigma, m, K} & & & \\
e_1\ e_2, \sigma, m, K & \to & e_1, \sigma, m, K \circ (\_\ (e_2, \sigma)) & \\
e_1 \rtimes e_2, \sigma, m, K & \to & e_1, \sigma, m, K \circ (\_ \rtimes e_2) & \\
x = e_1; e_2, \sigma, m, K & \to & e_1, (x, \ell) :: \sigma, m, K \circ (x = \ell; (e_2, \sigma)) & \ell \notin \mathrm{dom}(m) \cup \mathrm{FLoc}(K) \\[1em]
\boxed{v, m, K \to e, \sigma, m, K} & & & \\
\langle \lambda x.e, \sigma_1 \rangle, m, K \circ (\_\ (e_2, \sigma)) & \to & e_2, \sigma, m, K \circ (\langle \lambda x.e, \sigma_1 \rangle\ \_) & \\
\sigma_1, m, K \circ (\_ \rtimes e_2) & \to & e_2, \sigma_1, m, K & \\
v_1, m, K \circ (x = \ell; (e_2, \sigma)) & \to & e_2, (x, \ell) :: \sigma, m[\ell \mapsto v_1], K \circ (x = \ell; \_) & \\
v_2, m, K \circ (\langle \lambda x.e, \sigma_1 \rangle\ \_) & \to & e, (x, \ell) :: \sigma_1, m[\ell \mapsto v_2], K & \ell \notin \mathrm{dom}(m) \cup \mathrm{FLoc}(K) \\[1em]
\boxed{v, m, K \to v, m, K} & & & \\
\sigma_2, m, K \circ (x = \ell; \_) & \to & (x, \ell) :: \sigma_2, m, K & \\[1em]
\boxed{e, \sigma, m, K \to v, m, K} & & & \\
x, \sigma, m, K & \to & v, m, K & \ell = \sigma(x), v = m(\ell) \\
\lambda x.e, \sigma, m, K & \to & \langle \lambda x.e, \sigma \rangle, m, K & \\
\varepsilon, \sigma, m, K & \to & \bullet, m, K &
\end{array}
$$

Figure 6: The small-step operational semantics with memory.

1.4 Reconciling the Two Semantics

We need to prove that the two semantics simulate each other. Thus, we need to define a notion of equivalence between the two semantic domains.

$$\boxed{w \sim_f v, m}$$

$$\textsc{Eq-Nil}\ \frac{}{\bullet \sim_f \bullet} \qquad \textsc{Eq-ConsFree}\ \frac{\ell \notin \mathrm{dom}(f) \qquad \ell \notin \mathrm{dom}(m) \qquad \sigma \sim_f \sigma'}{(x,\ell) :: \sigma \sim_f (x,\ell) :: \sigma'} \qquad \textsc{Eq-ConsBound}\ \frac{f(\ell) = \ell' \qquad \ell' \in \mathrm{dom}(m) \qquad \sigma \sim_f \sigma'}{(x,\ell) :: \sigma \sim_f (x,\ell') :: \sigma'}$$

$$\textsc{Eq-ConsWVal}\ \frac{m(\ell') = v' \qquad w \sim_f v' \qquad \sigma \sim_f \sigma'}{(x,w) :: \sigma \sim_f (x,\ell') :: \sigma'} \qquad \textsc{Eq-Clos}\ \frac{\sigma \sim_f \sigma'}{\langle \lambda x.e, \sigma \rangle \sim_f \langle \lambda x.e, \sigma' \rangle} \qquad \textsc{Eq-Rec}\ \frac{L \text{ finite} \qquad m(\ell') = v' \qquad \forall \ell \notin L,\ v^\ell \sim_{f[\ell \mapsto \ell']} v'}{\mu.v \sim_f v'}$$

Figure 7: The equivalence relation between weak values in the original semantics and values in the semantics with memory. $f \in \mathrm{Loc} \xrightarrow{\text{fin}} \mathrm{Loc}$ tells what the free locations in w should be mapped to in memory.

Lemma 1.1 (Equivalence under Substitution)**.** For all $w_1, w_2, \ell, f, v_1', v_2', \ell', m$,

$$(w_1 \sim_{f[\ell \mapsto \ell']} v_1', m - \ell') \wedge (v_2' = m(\ell')) \wedge (w_2 \sim_f v_2', m) \Rightarrow w_1[w_2/\ell] \sim_f v_1', m$$

2 Typing (Without Recursive Bindings)

The definitions for types are in Figure 8 and the typing rules are in Figure 9. The definitions for subtyping are in Figure 10.

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

Figure 8: Definition of types.

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

$$\textsc{T-Id}\ \frac{\tau = \Gamma(x)}{\Gamma \vdash x : \tau} \qquad \textsc{T-Fn}\ \frac{(x,\tau_1) :: \Gamma \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \rightarrow \tau_2} \qquad \textsc{T-App}\ \frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau \qquad \Gamma \vdash e_2 : \tau_2 \qquad \tau_1 \geq \tau_2}{\Gamma \vdash e_1\, e_2 : \tau}$$

$$\textsc{T-Link}\ \frac{\Gamma \vdash e_1 : \Gamma_1 \qquad \Gamma_1 \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \rtimes e_2 : \tau_2} \qquad \textsc{T-Empty}\ \frac{}{\Gamma \vdash \varepsilon : \bullet} \qquad \textsc{T-Bind}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad (x,\tau_1) :: \Gamma \vdash e_2 : \Gamma_2}{\Gamma \vdash x = e_1; e_2 : (x,\tau_1) :: \Gamma_2}$$

Figure 9: The typing judgment.

2.1 Type Safety

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

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

$\boxed{\tau \geq \tau}$

$$\frac{}{\bullet \geq \bullet}\textsc{Nil} \qquad \frac{x \notin \mathsf{dom}(\Gamma) \quad \Gamma \geq \Gamma'}{\Gamma \geq (x,\tau) :: \Gamma'}\textsc{ConsFree} \qquad \frac{\Gamma(x) \geq \tau \quad \Gamma - x \geq \Gamma'}{\Gamma \geq (x,\tau) :: \Gamma'}\textsc{ConsBound} \qquad \frac{\tau_2 \geq \tau_1 \quad \tau_1' \geq \tau_2'}{\tau_1 \to \tau_1' \geq \tau_2 \to \tau_2'}\textsc{Arrow}$$

Figure 10: The subtype relation.

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

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

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

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

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

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

We want to prove that:

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

by induction on $\vdash$.

For the base case 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:

$$\tau_1 \geq \tau_2 \Rightarrow \mathcal{V}[\![\tau_1]\!] \supseteq \mathcal{V}[\![\tau_2]\!]$$

Then by the inductive hypothesis and compatibility, the result follows. □

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

Figure 11: Definition of type constraints.

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

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

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 $\bot$. Also note that we write α for $\alpha.\epsilon$ as well.

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

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

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

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

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

$$\tau.\epsilon \triangleq \tau \qquad\qquad \tau.px \triangleq (\tau.p)(x)$$

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

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

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

$$
\begin{array}{rcll}
S\bullet & \triangleq & \bullet & \\
S(\alpha.p) & \triangleq & \alpha.p & \text{when } \alpha \notin \textit{dom}(S) \\
S([].p) & \triangleq & [].p &
\end{array}
\qquad
\begin{array}{rcll}
S(\tau_1 \to \tau_2) & \triangleq & S\tau_1 \to S\tau_2 & \\
S(\alpha.p) & \triangleq & \tau.p & \text{when } \alpha \mapsto \tau \in S
\end{array}
$$

Define:

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

where subtyping rules are the same as Figure 10 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:

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

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. □