

Semantics for Modular Analysis

Joonhyup Lee

1 Abstract Syntax

In this section we define the abstract syntax for a simple language that captures the essence of modules and linking. The language is basically an extension of untyped lambda calculus with modules and the linking operator.

x	\in	ExprVar	
M	\in	ModVar	
e	\in	Expr	
e	$::=$	x	<i>identifier, expression</i>
		$ \lambda x.e$	<i>function</i>
		$ e e$	<i>application</i>
		$ e!e$	<i>linked expression</i>
		$ \varepsilon$	<i>empty module</i>
		$ M$	<i>identifier, module</i>
		$ \text{let } x \ e \ e$	<i>let-binding, expression</i>
		$ \text{let } M \ e \ e$	<i>let-binding, module</i>

1.1 Rationale for the design of the simple language

There are no recursive modules, first-class modules, or functors in the simple language that is defined. Also, note that the nonterminals for the modules and expressions are not separated. Why is this so?

The rationale for the exclusion of recursive modules/first-class modules/functors is because we want to enforce static scoping. That is, we need to be able to statically determine where variables were bound when using them. To enforce static scoping when function applications might return modules, we need to employ signatures to project the dynamically computed modules onto a statically known context. Concretely, we need to define signatures S where $\lambda M :> S.e$ statically resolves the context when M is used in the body e , and $e :> S$ enforces that a dynamic computation is resolved into one static form.

The rationale for not separating modules and expressions in the syntax is because we want to utilize the linking operator to link both modules to expressions and modules to modules. That is, we want expressions to be parsed as $(m_1!m_2)!e$. $m_1!m_2$ links a module with a module, and $(m_1!m_2)!e$ links a module with an expression. Why this is convenient will be clear when we explain separate analysis; we want to link modules with modules as well as expressions.

2 Big-Step Operational Semantics

In this section we give the big-step operational semantics for the dynamic execution of the module language. The big-step evaluation relation relates the initial state(memory and time) and configuration(the subexpression being evaluated, and the surrounding dynamic context) with the resulting state and value.

This relation is nonstandard in that the *environment* that is often used to define closures in the call-by-value dynamics is not a finite map from variables to values. Rather, the surrounding *syntactic* context annotated with the *binding times* for the variables serve as the environment. To access the value of the variable x from the context C , one has to read off the closest binding time from the context and look up the value bound at that time from the memory. To access the exported context from the variable M , one has to look up the exported context from C , not from the memory.

This separation between where we store modules and where we store the evaluated values from expressions emphasizes the fact that *where* the variables are bound is guided by syntax. The only thing that is dynamic is *when* the variables are bound, which is represented by the time component.

Now, we start by defining what we mean by *time* and *context*, which is the essence of our model.

2.1 Time and Context

We first define sets that are parametrized by our choice of the time domain, mainly the *value*, *memory*, and *context* domains. Also, we present the notational conventions used in this paper to represent members of each domain.

t	\in	\mathbb{T}	
v	\in	$\text{Val } \mathbb{T}$	
C	\in	$\text{Ctx } \mathbb{T}$	
V	\in	$\text{Val } \mathbb{T} + \text{Ctx } \mathbb{T}$	
σ	\in	$\text{Mem } \mathbb{T} \triangleq \mathbb{T} \xrightarrow{\text{fin}} \text{Val } \mathbb{T}$	
C	$::=$	$[]$	<i>hole</i>
		$ \ \lambda x^t.C$	<i>function parameter binding</i>
		$ \ \text{let } x^t C$	<i>let expression binding</i>
		$ \ \text{let } M C C$	<i>let module context binding</i>
v	$::=$	$\langle \lambda x.e, C \rangle$	<i>closure</i>

Above, there are no constraints placed upon the set \mathbb{T} . Now we give the conditions that the concrete time domain must satisfy.

Definition 2.1 (Concrete time). $(\mathbb{T}, \leq, \text{tick})$ is a *concrete time* when

1. (\mathbb{T}, \leq) is a total order.
2. $\text{tick} \in \text{Ctx } \mathbb{T} \rightarrow \text{Mem } \mathbb{T} \rightarrow \mathbb{T} \rightarrow \text{ExprVar} \rightarrow \text{Val } \mathbb{T} \rightarrow \mathbb{T}$ gives a strictly larger timestamp. That is,

$$\forall C \in \text{Ctx } \mathbb{T}, \sigma \in \text{Mem } \mathbb{T}, t \in \mathbb{T}, x \in \text{ExprVar}, v \in \text{Val } \mathbb{T}, t < \text{tick } C \sigma t x v.$$

tick updates the timestamp when v is bound to x under the context C and state (σ, t) .

Now for the auxiliary operators that is used when defining the evaluation relation. We first define the plugin operator for the dynamic context.

$$C_1[C_2] \triangleq \begin{cases} C_2 & (C_1 = []) \\ \lambda x^t.C'[C_2] & (C_1 = \lambda x^t.C') \\ \text{let } x^t C'[C_2] & (C_1 = \text{let } x^t C') \\ \text{let } M C' C''[C_2] & (C_1 = \text{let } M C' C'') \end{cases}$$

Next, the function that extracts the address for an ExprVar must be defined.

$$\text{addr}(C, x) \triangleq \begin{cases} \perp & (C = []) \\ \perp & (C = \lambda x'^t.C' \wedge x' \neq x \wedge \text{addr}(C', x) = \perp) \\ \perp & (C = \text{let } x'^t C' \wedge x' \neq x \wedge \text{addr}(C', x) = \perp) \\ t & (C = \lambda x'^t.C' \wedge x' = x \wedge \text{addr}(C', x) = \perp) \\ t & (C = \text{let } x'^t C' \wedge x' = x \wedge \text{addr}(C', x) = \perp) \\ t' & (C = \lambda x'^t.C' \wedge \text{addr}(C', x) = t') \\ t' & (C = \text{let } x'^t C' \wedge \text{addr}(C', x) = t') \\ \text{addr}(C'', x) & (C = \text{let } M C' C'') \end{cases}$$

Finally, the function that looks up the dynamic context bound to a module variable M must be defined.

$$\text{ctx}(C, M) \triangleq \begin{cases} \perp & (C = []) \\ C' & (C = \text{let } M' C' C'' \wedge M' = M \wedge \text{ctx}(C'', M) = \perp) \\ \text{ctx}(C'', M) & (C = \text{let } M' C' C'' \wedge \text{ctx}(C'', M) \neq \perp) \\ \text{ctx}(C'', M) & (C = \text{let } M' C' C'' \wedge M' \neq M) \\ \text{ctx}(C', M) & (C = \lambda x^t.C') \\ \text{ctx}(C', M) & (C = \text{let } x^t C') \end{cases}$$

2.2 The Evaluation Relation

Now we are in a position to define the big-step evaluation relation. The relation \Downarrow relates $(e, C, \sigma, t) \in \text{Expr} \times \text{Ctx } \mathbb{T} \times \text{Mem } \mathbb{T} \times \mathbb{T}$ with $(V, \sigma, t) \in (\text{Val } \mathbb{T} + \text{Ctx } \mathbb{T}) \times \text{Mem } \mathbb{T} \times \mathbb{T}$. Note that we constrain whether the evaluation relation returns $v \in \text{Val } \mathbb{T}$ (when the expression being evaluated is not a module) or $C \in \text{Ctx } \mathbb{T}$ by the definition of the relation.

$$\begin{aligned} [\text{EXPRVAR}] \quad & \frac{t_x = \text{addr}(C, x) \quad \sigma(t_x) = v}{(x, C, \sigma, t) \Downarrow (v, \sigma, t)} \\ [\text{FN}] \quad & \frac{}{(\lambda x.e, C, \sigma, t) \Downarrow (\langle \lambda x.e, C \rangle, \sigma, t)} \end{aligned}$$

$$[\text{APP}] \frac{\begin{array}{c} (e_1, C, \sigma, t) \Downarrow (\langle \lambda x. e_\lambda, C_\lambda \rangle, \sigma_\lambda, t_\lambda) \\ (e_2, C, \sigma_\lambda, t_\lambda) \Downarrow (v, \sigma_a, t_a) \\ (e_\lambda, C_\lambda [\lambda x^{t_a}. []], \sigma_a [t_a \mapsto (v, C_a)], \text{tick } C \sigma_a t_a x v) \Downarrow (v', \sigma', t') \end{array}}{(e_1 e_2, C, \sigma, t) \Downarrow (v', \sigma', t')}$$

$$[\text{LINKING}] \frac{\begin{array}{c} (e_1, C, \sigma, t) \Downarrow (C', \sigma', t') \\ (e_2, C', \sigma', t') \Downarrow (V, \sigma'', t'') \end{array}}{(e_1 ! e_2, C, \sigma, t) \Downarrow (V, \sigma'', t')}$$

Note that we do not constrain whether v or C is returned by e_2 in the linking case. That is, linking may return either values or modules.

$$[\text{EMPTY}] \frac{}{(\varepsilon, C, \sigma, t) \Downarrow (C, \sigma, t)}$$

$$[\text{MODVAR}] \frac{C' = \text{ctx}(C, M)}{(M, C, \sigma, t) \Downarrow (C', \sigma, t)}$$

$$[\text{LETE}] \frac{\begin{array}{c} (e_1, C, \sigma, t) \Downarrow (v, \sigma', t') \\ (e_2, C[\text{let } x^{t'} []], \sigma'[t' \mapsto v], \text{tick } C \sigma' t' x v) \Downarrow (C', \sigma'', t'') \end{array}}{(\text{let } x e_1 e_2, C, \sigma, t) \Downarrow (C', \sigma'', t')}$$

$$[\text{LETM}] \frac{\begin{array}{c} (e_1, C, \sigma, t) \Downarrow (C', \sigma', t') \\ (e_2, C[\text{let } M C' []], \sigma', t') \Downarrow (C'', \sigma'', t'') \end{array}}{(\text{let } M e_1 e_2, C, \sigma, t) \Downarrow (C'', \sigma'', t')}$$

3 Collecting Semantics

We want to enable separate analysis

Definition 3.1 (Reachable stuck states).

$$\underline{e}^\times(C, \sigma, t) \triangleq \{(e', C', \sigma, t') \mid (e, C, \sigma, t) \rightsquigarrow (e', C', \sigma', t') \wedge \forall v, \sigma'', t'', \neg((e', C' \sigma', t') \Downarrow (v, \sigma'', t''))\}$$

Definition 3.2 (Reachable resolved states).

$$\underline{e}^\circ(C, \sigma, t) \triangleq \{((e', C', \sigma', t'), (v, \sigma'', t'')) \mid (e, C, \sigma, t) \rightsquigarrow (e', C', \sigma', t') \wedge (e', C' \sigma', t') \Downarrow (v, \sigma'', t'')\}$$

Definition 3.3 (Collecting semantics). The semantics for an expression e under context C , memory σ and time t is:

$$\llbracket e \rrbracket(C, \sigma, t) \triangleq (\underline{e}^\circ(C, \sigma, t), \underline{e}^\times(C, \sigma, t))$$

Lemma 3.1 (Separation of linking, resolved part).

$$\underline{e_1 ! e_2}^\circ(C, \sigma, t) = \underline{e_1}^\circ(C, \sigma, t) \cup \left(\bigcup_{(e_1, C, \sigma, t) \Downarrow (C', \sigma', t')} \underline{e_2}^\circ(C', \sigma', t') \right)$$

Lemma 3.2 (Separation of linking, stuck part).

$$\underline{e_1 ! e_2}^\times(C, \sigma, t) = \underline{e_1}^\times(C, \sigma, t) \cup \left(\bigcup_{(e_1, C, \sigma, t) \Downarrow (C', \sigma', t')} \underline{e_2}^\times(C', \sigma', t') \right)$$

Now we only need a way to calculate $\llbracket e_2 \rrbracket(C', \sigma', t')$ by utilizing the separately analyzed $\llbracket e_2 \rrbracket([], \emptyset, t_0)$.

Lemma 3.3 (Separate analysis).