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*	
m	$::=$	ε	*empty module*
	$\mid$	M	*identifier, module*
	$\mid$	$\texttt{let}\ x\ e\ m$	*let-binding, expression*
	$\mid$	$\texttt{let}\ M\ m\ m$	*let-binding, module*
e	$::=$	x	*identifier, expression*
	$\mid$	$\lambda x.e$	*function*
	$\mid$	$e\ e$	*application*
	$\mid$	$m!e$	*linked expression*

1.1 Rationale for the design of the simple language

We want to statically determine what names are available. This was the simplest language I could think of that enforces this criterion. Of course, modules with signatures(ref: Rossberg, 1ML) enforce this criterion as well, but that is not the point here. The abstraction that we make is possible iff the context of the expression to be evaluated can always be determined statically. Thus, the same abstraction can readily be extended to any other language satisfying this criterion.

2 Big-Step Operational Semantics

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

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 together with the memory serves as the environment. To access the value of the variable x from the context C, one has to read the list of times starting from the nearest binding time for x in the surrounding context up until the root, and use that list as an address to search the memory for the value.

The reason the semantics is defined as such is for the convenience and precision of abstraction. One only has to finitize the time component to make the search space finite. Furthermore, one can reason precisely about how the syntactic change on the surrounding context affects the evaluation of the expression inside the hole of C by including C in the configuration.

Before presenting the inference rules for the big-step relation, we first present the domains for the state(σ, t) and configuration(C, e). The concrete time t is defined to be a natural number, C is inside the inductively defined set Ctx, and σ is a finite map from Path $\triangleq$ list(time) to $\mathbb{V} \times$ Ctx, when $\mathbb{V}$ is the subset of the set of expressions that satisfy the *value* predicate. In this language, only functions satisfy this predicate.

t	$\in$	$\mathbb{N}$	
p	$\in$	Path	
C	$\in$	Ctx	
σ	$\in$	$\text{Path} \xrightarrow{\text{fin}} \mathbb{V} \times \mathbb{C}$	
p	$::=$	$\texttt{nil}$	*nil*
	$\mid$	$t :: p$	*cons*
C	$::=$	$\texttt{[]}$	*hole*
	$\mid$	$\lambda x^t.C$	*function parameter binding*
	$\mid$	$\texttt{let}\ x^t\ C$	*let expression binding*
	$\mid$	$\texttt{let}\ M\ C\ C$	*let module context binding*

We define the plugin operator for the dynamic context.

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

Now, for the operational semantics, the functions that extract information about the binding times from the dynamic context must be defined. The first function to be defined is the level function, which returns the list of binding times counted from the hole upwards.

$$\mathsf{level}(C) \triangleq \begin{cases} \mathsf{nil} & (C = []) \\ \mathsf{level}(C') +\!\!+ [t] & (C = \lambda x^t.C' \lor \texttt{let}\ x^t\ C') \\ \mathsf{level}(C'') & (C = \texttt{let}\ M\ C'\ C'') \end{cases}$$

Also, the function that calculates the address from the dynamic context C and variable x must be defined. Note that nil is used as the null address, which should never be accessed.

$$\mathsf{addr}(C, x) \triangleq \begin{cases} \mathsf{nil} & (C = []) \\ \mathsf{nil} & (C = \lambda x'^t.C' \land x' \neq x \land \mathsf{addr}(C', x) = \mathsf{nil}) \\ \mathsf{nil} & (C = \texttt{let}\ x'^t C' \land x' \neq x \land \mathsf{addr}(C', x) = \mathsf{nil}) \\ [t] & (C = \lambda x'^t.C' \land x' = x \land \mathsf{addr}(C', x) = \mathsf{nil}) \\ [t] & (C = \texttt{let}\ x'^t C' \land x' = x \land \mathsf{addr}(C', x) = \mathsf{nil}) \\ p +\!\!+ [t] & (C = \lambda x'^t.C' \land \mathsf{addr}(C', x) = p \neq \mathsf{nil}) \\ p +\!\!+ [t] & (C = \texttt{let}\ x'^t\ C' \land \mathsf{addr}(C', x) = p \neq \mathsf{nil}) \\ \mathsf{addr}(C'', x) & (C = \texttt{let}\ M\ C'\ C'') \end{cases}$$

Finally, the function that looks up the dynamic context bound to a module variable M must be defined. Note that this function returns $\bot$ when the module M is not found.

$$\mathsf{ctx}(C, M) \triangleq \begin{cases} \bot & (C = []) \\ C' & (C = \texttt{let}\ M'\ C'\ C'' \land M' = M \land \mathsf{ctx}(C'', M) = \bot) \\ \mathsf{ctx}(C'', M) & (C = \texttt{let}\ M'\ C'\ C'' \land \mathsf{ctx}(C'', M) \neq \bot) \\ \mathsf{ctx}(C'', M) & (C = \texttt{let}\ M'\ C'\ C'' \land M' \neq M) \\ \mathsf{ctx}(C', M) & (C = \lambda x^t.C') \\ \mathsf{ctx}(C', M) & (C = \texttt{let}\ x^t\ C') \end{cases}$$

Now we are in a position to define the big-step evaluation relation.

$$[\textsc{ExprVar}]\ \frac{p_x = \mathsf{addr}(C, x) \qquad p_x \neq \mathsf{nil}}{(x, C), (\sigma, t) \Downarrow \sigma(p_x), (\sigma, t)}$$

$$[\textsc{Fn}]\ \frac{}{(\lambda x.e, C), (\sigma, t) \Downarrow (\lambda x.e, C), (\sigma, t)}$$

$$[\textsc{App}]\ \frac{\begin{array}{c}(e_1, C), (\sigma, t) \Downarrow (\lambda x.e_\lambda, C_\lambda), (\sigma_\lambda, t_\lambda) \\ (e_2, C), (\sigma_\lambda, t_\lambda) \Downarrow (v, C_a), (\sigma_a, t_a) \\ (e_\lambda, C_\lambda[\lambda x^{t_a}.[]]), (\sigma_a[t_a :: \mathsf{level}(C_\lambda) \mapsto (v, C_a)], t_a + 1) \Downarrow (v', C'), (\sigma', t')\end{array}}{(e_1\ e_2, C), (\sigma, t) \Downarrow (v', C'), (\sigma', t')}$$

$$[\textsc{Linking}]\ \frac{\begin{array}{c}(m, C), (\sigma, t) \Downarrow C', (\sigma', t') \\ (e, C'), (\sigma', t') \Downarrow (v, C''), (\sigma'', t'')\end{array}}{(m!e, C), (\sigma, t) \Downarrow (v, C''), (\sigma'', t'')}$$

$$[\textsc{Empty}]\ \frac{}{(\varepsilon, C), (\sigma, t) \Downarrow C, (\sigma, t)}$$

$$[\textsc{ModVar}]\ \frac{C' = \mathsf{ctx}(C, M) \qquad C' \neq \bot}{(M, C), (\sigma, t) \Downarrow C', (\sigma, t)}$$

$$[\textsc{LetE}]\ \frac{\begin{array}{c}(e, C), (\sigma, t) \Downarrow (v, C'), (\sigma', t') \\ (m, C[\mathtt{let}\ x^{t'}\ []]), (\sigma_v[t'\ ::\ \mathsf{level}(C) \mapsto (v, C')], t' + 1) \Downarrow C'', (\sigma'', t'')\end{array}}{(\mathtt{let}\ x\ e\ m, C), (\sigma, t) \Downarrow C'', (\sigma'', t'')}$$

$$[\textsc{LetM}]\ \frac{\begin{array}{c}(m_1, C), (\sigma, t) \Downarrow C', (\sigma', t') \\ (m_2, C[\mathtt{let}\ M\ C'\ []]), (\sigma'[t'\ ::\ \mathsf{level}(C') \mapsto (p', C')], t' + 1) \Downarrow C'', (\sigma'', t'')\end{array}}{(\mathtt{let}\ M\ m_1\ m_2, C), (\sigma, t) \Downarrow C'', (\sigma'', t'')}$$

To

3 Collecting Semantics

Now we make the semantics of a module m and the semantics of an expression e explicit.