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	*identifier, expression*
	$\mid$	$\lambda x.e$	*function*
	$\mid$	$e\ e$	*application*
	$\mid$	$e!e$	*linked expression*
	$\mid$	ε	*empty module*
	$\mid$	M	*identifier, module*
	$\mid$	`let` $x\ e\ e$	*let-binding, expression*
	$\mid$	`let` $M\ e\ e$	*let-binding, module*

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 together with the memory serves 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. Below, the $\mathbb{T}$ set may be any arbitrary set. Later, we constrain $\mathbb{T}$ to meet the constraint that tick always give fresh timestamps.

$$
\begin{array}{lcll}
t & \in & \mathbb{T} & \\
v & \in & \mathrm{Val}\,\mathbb{T} & \\
C & \in & \mathrm{Ctx}\,\mathbb{T} & \\
\sigma & \in & \mathrm{Mem}\,\mathbb{T} \triangleq \mathbb{T} \xrightarrow{\text{fin}} \mathrm{Val}\,\mathbb{T} & \\
C & ::= & [\,] & \textit{hole} \\
 & | & \lambda x^t.C & \textit{function parameter binding} \\
 & | & \texttt{let}\ x^t\ C & \textit{let expression binding} \\
 & | & \texttt{let}\ M\ C\ C & \textit{let module context binding} \\
v & ::= & \langle \lambda x.e, C \rangle & \textit{closure}
\end{array}
$$

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.

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

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.

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

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.

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

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

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

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

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

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

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

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

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

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

To

3 Collecting Semantics

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