# 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.

| | | | | |
|---|---|---|---|---|
| Expression Identifier | $x$ | $\in$ | ExprVar | |
| Module Identifier | $M$ | $\in$ | ModVar | |
| Expression | $e$ | $\in$ | Expr | |
| Expression | $e$ | $\rightarrow$ | $x$ | *identifier, expression* |
| | | $\mid$ | $\lambda x.e$ | *function* |
| | | $\mid$ | $e\ e$ | *application* |
| | | $\mid$ | $e!e$ | *linked expression* |
| | | $\mid$ | $\varepsilon$ | *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 configuration(context, memory and time) and expression with the resulting value and state.

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.

| | | | | |
|---|---|---|---|---|
| Time | $t$ | $\in$ | $\mathbb{T}$ | |
| Context | $C$ | $\in$ | Ctx | |
| Result of expressions | $v$ | $\in$ | $\text{Val} \triangleq \text{Expr} \times \text{Ctx}$ | |
| Result of expressions/modules | $V$ | $\in$ | $\text{Val} + \text{Ctx}$ | |
| Memory | $m$ | $\in$ | $\text{Mem} \triangleq \mathbb{T} \xrightarrow{\text{fin}} \text{Val}$ | |
| Configuration | $s$ | $\in$ | $\text{Config} \triangleq \text{Ctx} \times \text{Mem} \times \mathbb{T}$ | |
| Result | $r$ | $\in$ | $\text{Result} \triangleq (\text{Val} + \text{Ctx}) \times \text{Mem} \times \mathbb{T}$ | |
| Context | $C$ | $\to$ | $[\,]$ | *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* |
| | $v$ | $\to$ | $\langle \lambda x.e, C \rangle$ | *closure* |

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. $\text{tick} \in \mathbb{T} \to \mathbb{T}$ satisfies: $\forall t \in \mathbb{T} : t < \text{tick}\, t$.

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

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

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

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

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

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

## 2.2 The Evaluation Relation

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

$$[\textsc{ExprVar}]\ \frac{t_x = \text{addr}(C, x) \qquad v = m(t_x)}{(x, C, m, t) \Downarrow (v, m, t)} \qquad [\textsc{Fn}]\ \frac{}{(\lambda x.e, C, m, t) \Downarrow (\langle \lambda x.e, C \rangle, m, t)}$$

$$[\textsc{App}]\ \frac{\begin{array}{c}(e_1, C, m, t) \Downarrow (\langle \lambda x.e_\lambda, C_\lambda \rangle, m_\lambda, t_\lambda) \\ (e_2, C, m_\lambda, t_\lambda) \Downarrow (v, m_a, t_a) \\ (e_\lambda, C_\lambda[\lambda x^{t_a}.[\,]], m_a[t_a \mapsto v], \text{tick}\, t_a) \Downarrow (v', m', t')\end{array}}{(e_1\ e_2, C, m, t) \Downarrow (v', m', t')} \qquad [\textsc{Linking}]\ \frac{\begin{array}{c}(e_1, C, m, t) \Downarrow (C', m', t') \\ (e_2, C', m', t') \Downarrow (V, m'', t'')\end{array}}{(e_1 ! e_2, C, m, t) \Downarrow (V, m'', t'')}$$

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.

$$[\textsc{Empty}]\ \frac{}{(\varepsilon, C, m, t) \Downarrow (C, m, t)} \qquad [\textsc{ModVar}]\ \frac{C' = \mathsf{ctx}(C, M)}{(M, C, m, t) \Downarrow (C', m, t)}$$

$$[\textsc{LetE}]\ \frac{(e_1, C, m, t) \Downarrow (v, m', t') \quad (e_2, C[\texttt{let}\ x^{t'}\ [\,]], m'[t' \mapsto v], \mathsf{tick}\, t') \Downarrow (C', m'', t'')}{(\texttt{let}\ x\ e_1\ e_2, C, m, t) \Downarrow (C', m'', t'')} \qquad [\textsc{LetM}]\ \frac{(e_1, C, m, t) \Downarrow (C', m', t') \quad (e_2, C[\texttt{let}\ M\ C'\ [\,]], m', t') \Downarrow (C'', m'', t'')}{(\texttt{let}\ M\ e_1\ e_2, C, m, t) \Downarrow (C'', m'', t'')}$$

The equivalence of the evaluation relation with a reference interpreter is formalized in Coq.

# 3 Collecting Semantics

To describe what happens when an expression $e$ is excecuted under the context $C$, memory $m$ and initial time $t$, we need to collect all *reachable states* starting from $(e, C, m, t)$ and all values that are returned by the reachable states. We first define the one-step reachability relation $\rightsquigarrow$.

$$[\textsc{AppL}]\ \frac{}{(e_1\ e_2, C, m, t) \rightsquigarrow (e_1, C, m, t)} \qquad [\textsc{AppR}]\ \frac{(e_1, C, m, t) \Downarrow (\langle \lambda x.e_\lambda, C_\lambda \rangle, m_\lambda, t_\lambda)}{(e_1\ e_2, C, m, t) \rightsquigarrow (e_2, C, m_\lambda, t_\lambda)}$$

$$[\textsc{AppBody}]\ \frac{(e_1, C, m, t) \Downarrow (\langle \lambda x.e_\lambda, C_\lambda \rangle, m_\lambda, t_\lambda) \quad (e_2, C, m_\lambda, t_\lambda) \Downarrow (v, m_a, t_a)}{(e_1\ e_2, C, m, t) \rightsquigarrow (e_\lambda, C_\lambda[\lambda x^{t_a}.[\,]], m_a[t_a \mapsto v], \mathsf{tick}\, t_a)}$$

$$[\textsc{LinkL}]\ \frac{}{(e_1 ! e_2, C, m, t) \rightsquigarrow (e_1, C, m, t)} \qquad [\textsc{LinkR}]\ \frac{(e_1, C, m, t) \Downarrow (C', m', t')}{(e_1 ! e_2, C, m, t) \rightsquigarrow (e_2, C', m', t')}$$

$$[\textsc{LetEL}]\ \frac{}{(\texttt{let}\ x\ e_1\ e_2, C, m, t) \rightsquigarrow (e_1, C, m, t)} \qquad [\textsc{LetER}]\ \frac{(e_1, C, m, t) \Downarrow (v, m', t')}{(\texttt{let}\ x\ e_1\ e_2, C, m, t) \rightsquigarrow (e_2, C[\texttt{let}\ x^{t'}\ [\,]], m'[t' \mapsto v], \mathsf{tick}\, t')}$$

$$[\textsc{LetML}]\ \frac{}{(\texttt{let}\ M\ e_1\ e_2, C, m, t) \rightsquigarrow (e_1, C, m, t)} \qquad [\textsc{LetMR}]\ \frac{(e_1, C, m, t) \Downarrow (C', m', t')}{(\texttt{let}\ M\ e_1\ e_2, C, m, t) \rightsquigarrow (e_2, C[\texttt{let}\ M\ C'\ [\,]], m', t')}$$

The well-definedness of the reachability relation with respect to a reference interpreter is formalized in Coq.

We want to enable separate analysis without making any assumptions on the free variables. That is, there may be stuck states because the value of free variables are not known in the given context. Since we want to collect the most precise information that describes the execution of the program, we have to collect all reachable states. The relation that collects the reachable states $\rightsquigarrow^*$ is defined as the reflexive and transitive closure of $\rightsquigarrow$.

**Definition 3.1** (Collecting semantics). The semantics for an expression $e$ under configuration $s \in \mathrm{Config}$ is an element in $(\mathrm{Expr} \times \mathrm{Config}) \to \wp(\mathrm{Result})_\perp$ defined as:

$$[\![e]\!](s) \triangleq \bigcup_{(e,s) \rightsquigarrow^* (e', s')} [(e', s') \mapsto \{r | (e', s') \Downarrow r\}]$$

That is, the semantics collect all reachable configurations and all results the configurations might return.

**Example 3.1.** The semantics for the non-terminating lambda expression $\Omega = (\lambda x.xx)(\lambda x.xx)$ satisfies:

$$[\![\Omega]\!]([\,], \emptyset, 0)(\Omega, [\,], \emptyset, 0) = \emptyset$$
$$[\![\Omega]\!]([\,], \emptyset, 0)(Y, \_, \_, \_) = \perp$$

since (1) $(\Omega, [\,], \emptyset, 0)$ is reached by $(\Omega, [\,], \emptyset, 0)$ but does not return, and (2) since the $Y$ combinator cannot be reached.

To justify separate analysis, we decompose the collecting semantics of the linking expression into a composition of the semantics of the left and right expressions.

**Definition 3.2** (Linking operator).

$$\mathsf{E}\ e_1\ s \triangleq [\![e_1]\!](s)(e_1, s) \qquad \text{(Exported from } e_1 \text{ under } s\text{)}$$

$$\mathsf{L}\ E\ e_2 \triangleq \bigcup_{s' \in E} [\![e_2]\!](s') \qquad \text{(Reached under exported context)}$$

$$\mathsf{L}^\circ\ E\ e_2 \triangleq \bigcup_{s' \in E} [\![e_2]\!](s')(e_2, s') \qquad \text{(Resolved under exported context)}$$

$$\mathsf{Link}\ e_1\ e_2\ s \triangleq [\![e_1]\!](s) \cup \mathsf{L}\ (\mathsf{E}\ e_1\ s)\ e_2 \cup [(e_1 ! e_2, s) \mapsto \mathsf{L}^\circ\ (\mathsf{E}\ e_1\ s)\ e_2] \qquad \text{(Linking of } e_1 \text{ and } e_2 \text{ under } s\text{)}$$

**Theorem 3.1** (Concrete linking).

$$[\![e_1 ! e_2]\!](s) = \mathsf{Link}\ e_1\ e_2\ s$$

That is, the semantics of the linked expression is the union of the semantics of the exporting expression and the semantics of the consuming expression under the exported configuration.

# 4 Abstract Semantics

The abstract semantics is almost exactly the same as the concrete semantics, except for the fact that the memory domain is now a finite map from the abstract time domain to a *set* of values. Note we do not need to define the $C^\#$, $v^\#$, $V^\#$ components, as they are *exactly* their concrete counterparts. They are simply $C$, $v$, $V$, parametrized by a different $\mathbb{T}$.

| | | | |
|---|---|---|---|
| Abstract time | $t^\#$ | $\in$ | $\mathbb{T}^\#$ |
| Context | $C^\#$ | $\in$ | Ctx |
| Result of expressions | $v^\#$ | $\in$ | Val |
| Result of expressions/modules | $V^\#$ | $\in$ | Val + Ctx |
| Abstract Memory | $m^\#$ | $\in$ | $\mathrm{Mem}^\# \triangleq \mathbb{T}^\# \xrightarrow{\text{fin}} \wp(\mathrm{Val})$ |
| Abstract Configuration | $s^\#$ | $\in$ | $\mathrm{Config}^\# \triangleq \mathrm{Ctx} \times \mathrm{Mem}^\# \times \mathbb{T}^\#$ |
| Abstract Result | $r^\#$ | $\in$ | $\mathrm{Result}^\# \triangleq (\mathrm{Val} + \mathrm{Ctx}) \times \mathrm{Mem}^\# \times \mathbb{T}^\#$ |

## 4.1 Big-Step Evaluation

First the abstract evaluation relation $\Downarrow^\#$ is defined. Note that the update for the memory is now a weak update. That is,

**Definition 4.1** (Weak update). Given $m^\# \in \mathrm{Mem}^\#$, $t^\# \in \mathbb{T}^\#$, $v^\# \in \mathrm{Val}$, we define $m^\#[t^\# \mapsto^\# v^\#]$ as:

$$m^\#[t^\# \mapsto^\# v^\#](t'^\#) \triangleq \begin{cases} m^\#(t^\#) \cup \{v^\#\} & (t'^\# = t^\#) \\ m^\#(t'^\#) & (\text{otherwise}) \end{cases}$$

Also, for the abstract time, we do not enforce the existence of an ordering on the timestamps, but we do need a policy for performing the tick operation. Since we want to utilize the information when the binding is performed, the $\mathrm{tick}^\#$ function takes in more information than simply the previous abstract time.

**Definition 4.2** (Abstract time). $(\mathbb{T}^\#, \mathrm{tick}^\#)$ is an *abstract time* when $\mathrm{tick}^\# \in \mathrm{Ctx} \to \mathrm{Mem}^\# \to \mathbb{T}^\# \to \mathrm{ExprVar} \to \mathrm{Val} \to \mathbb{T}^\#$ is the policy for advancing the timestamp.

In our semantics, $\mathrm{tick}^\#\ C^\#\ m^\#\ t^\#\ x\ v^\#$ is performed when $v^\#$ is bound to $x$ under configuration $(C^\#, m^\#, t^\#)$.

$$[\textsc{ExprVar}]\ \frac{{t_x}^\# = \mathrm{addr}(C^\#, x) \qquad v^\# \in m^\#({t_x}^\#)}{(x, C^\#, m^\#, t^\#) \Downarrow^\# (v^\#, m^\#, t^\#)} \qquad [\textsc{Fn}]\ \frac{}{(\lambda x.e, C^\#, m^\#, t^\#) \Downarrow^\# (\langle \lambda x.e, C^\# \rangle, m^\#, t^\#)}$$

$$[\textsc{App}]\ \frac{\begin{array}{c}(e_1, C^\#, m^\#, t^\#) \Downarrow^\# (\langle \lambda x.e_\lambda, {C_\lambda}^\# \rangle, {m_\lambda}^\#, {t_\lambda}^\#) \\ (e_2, C^\#, {m_\lambda}^\#, {t_\lambda}^\#) \Downarrow^\# (v^\#, {m_a}^\#, {t_a}^\#) \\ ({e_\lambda}, {C_\lambda}^\#[\lambda x^{{t_a}^\#}.[]], {m_a}^\#[{t_a}^\# \mapsto^\# v^\#], \mathrm{tick}^\#\ C^\#\ {m_a}^\#\ {t_a}^\#\ x\ v^\#) \Downarrow^\# (v'^\#, m'^\#, t'^\#)\end{array}}{(e_1\ e_2, C^\#, m^\#, t^\#) \Downarrow^\# (v'^\#, m'^\#, t'^\#)}$$

$$[\textsc{Linking}]\ \frac{\begin{array}{c}(e_1, C^\#, m^\#, t^\#) \Downarrow^\# (C'^\#, m'^\#, t'^\#) \\ (e_2, C'^\#, m'^\#, t'^\#) \Downarrow^\# (V^\#, m''^\#, t''^\#)\end{array}}{(e_1 ! e_2, C^\#, m^\#, t^\#) \Downarrow^\# (V^\#, m''^\#, t''^\#)}$$

$$[\textsc{Empty}]\ \frac{}{(\varepsilon, C^\#, m^\#, t^\#) \Downarrow^\# (C^\#, m^\#, t^\#)} \qquad [\textsc{ModVar}]\ \frac{C'^\# = \mathrm{ctx}(C^\#, M)}{(M, C^\#, m^\#, t^\#) \Downarrow^\# (C'^\#, m^\#, t^\#)}$$

$$[\textsc{LetE}]\ \frac{(e_1, C^\#, m^\#, t^\#)\Downarrow^\#(v^\#, m'^\#, t'^\#) \quad (e_2, C^\#[\texttt{let}\ x^{t'^\#}\ [\,]], m'^\#[t'^\# \mapsto^\# v^\#], \mathsf{tick}^\#\ C^\#\ m'^\#\ t'^\#\ x\ v^\#)\Downarrow^\#(C'^\#, m''^\#, t''^\#)}{(\texttt{let}\ x\ e_1\ e_2, C^\#, m^\#, t^\#)\Downarrow^\#(C'^\#, m''^\#, t''^\#)}$$

$$[\textsc{LetM}]\ \frac{(e_1, C^\#, m^\#, t^\#)\Downarrow^\#(C'^\#, m'^\#, t'^\#) \quad (e_2, C^\#[\texttt{let}\ M\ C'^\#\ [\,]], m'^\#, t'^\#)\Downarrow^\#(C''^\#, m''^\#, t''^\#)}{(\texttt{let}\ M\ e_1\ e_2, C^\#, m^\#, t^\#)\Downarrow^\#(C''^\#, m''^\#, t''^\#)}$$

## 4.2 Big-Step Reachability

The abstract one-step reachability relation $\rightsquigarrow^\#$ is defined as:

$$[\textsc{AppL}]\ \frac{}{(e_1\ e_2, C^\#, m^\#, t^\#)\rightsquigarrow^\#(e_1, C^\#, m^\#, t^\#)} \qquad [\textsc{AppR}]\ \frac{(e_1, C^\#, m^\#, t^\#)\Downarrow^\#(\langle\lambda x.e_\lambda, {C_\lambda}^\#\rangle, {m_\lambda}^\#, {t_\lambda}^\#)}{(e_1\ e_2, C^\#, m^\#, t^\#)\rightsquigarrow^\#(e_2, C^\#, {m_\lambda}^\#, {t_\lambda}^\#)}$$

$$[\textsc{AppBody}]\ \frac{(e_1, C^\#, m^\#, t^\#)\Downarrow^\#(\langle\lambda x.e_\lambda, {C_\lambda}^\#\rangle, {m_\lambda}^\#, {t_\lambda}^\#) \quad (e_2, C^\#, {m_\lambda}^\#, {t_\lambda}^\#)\Downarrow^\#(v^\#, {m_a}^\#, {t_a}^\#)}{(e_1\ e_2, C^\#, m^\#, t^\#)\rightsquigarrow^\#(e_\lambda, {C_\lambda}^\#[\lambda x^{{t_a}^\#}.[\,]], {m_a}^\#[{t_a}^\# \mapsto^\# v^\#], \mathsf{tick}^\#\ C^\#\ {m_a}^\#\ {t_a}^\#\ x\ v^\#)}$$

$$[\textsc{LinkL}]\ \frac{}{(e_1 ! e_2, C^\#, m^\#, t^\#)\rightsquigarrow^\#(e_1, C^\#, m^\#, t^\#)} \qquad [\textsc{LinkR}]\ \frac{(e_1, C^\#, m^\#, t^\#)\Downarrow^\#(C'^\#, m'^\#, t'^\#)}{(e_1 ! e_2, C^\#, m^\#, t^\#)\rightsquigarrow^\#(e_2, C'^\#, m'^\#, t'^\#)}$$

$$[\textsc{LetEL}]\ \frac{}{(\texttt{let}\ x\ e_1\ e_2, C^\#, m^\#, t^\#)\rightsquigarrow^\#(e_1, C^\#, m^\#, t^\#)}$$

$$[\textsc{LetER}]\ \frac{(e_1, C^\#, m^\#, t^\#)\Downarrow^\#(v^\#, m'^\#, t'^\#)}{(\texttt{let}\ x\ e_1\ e_2, C^\#, m^\#, t^\#)\rightsquigarrow^\#(e_2, C^\#[\texttt{let}\ x^{t'^\#}\ [\,]], m'^\#[t'^\# \mapsto^\# v^\#], \mathsf{tick}^\#\ C^\#\ m'^\#\ t'^\#\ x\ v^\#)}$$

$$[\textsc{LetML}]\ \frac{}{(\texttt{let}\ M\ e_1\ e_2, C^\#, m^\#, t^\#)\rightsquigarrow^\#(e_1, C^\#, m^\#, t^\#)} \quad [\textsc{LetMR}]\ \frac{(e_1, C^\#, m^\#, t^\#)\Downarrow^\#(C'^\#, m'^\#, t'^\#)}{(\texttt{let}\ M\ e_1\ e_2, C^\#, m^\#, t^\#)\rightsquigarrow^\#(e_2, C^\#[\texttt{let}\ M\ C'^\#\ [\,]], m'^\#, t'^\#)}$$

## 4.3 Soundness

From the relations above, we can define the abstract semantics:

**Definition 4.3** (Abstract semantics). The semantics for an expression $e$ under configuration $s^\# \in \mathrm{Config}^\#$ is an element in $(\mathrm{Expr} \times \mathrm{Config}^\#) \to \wp(\mathrm{Result}^\#)_\perp$ defined as:

$$[\![e]\!]^\#(s^\#) \triangleq \bigsqcup_{(e,s^\#)\rightsquigarrow^{\#*}(e',s'^\#)} [(e', s'^\#) \mapsto \{r^\# | (e', s'^\#)\Downarrow^\# r^\#\}]$$

We need to present how to compute the abstract semantics by equating the semantics with the least fixed point of some transfer function.

**Definition 4.4** (Transfer function). Given an element $a^\#$ of $(\mathrm{Expr} \times \mathrm{Config}^\#) \to \wp(\mathrm{Result}^\#)_\perp$,

- Define $\Downarrow^\#_{a^\#}$ and $\rightsquigarrow^\#_{a^\#}$ by replacing all assumptions of the form $s^\#\Downarrow^\# r^\#$ to $r^\# \in a^\#(s^\#)$ in $\Downarrow^\#$ and $\rightsquigarrow^\#$.

We define the transfer function $F^\#$ by:

$$F^\#(a^\#) \triangleq a^\# \sqcup \bigsqcup_{\substack{(e,s^\#) \in \mathrm{dom}(a^\#) \\ (e,s^\#)\rightsquigarrow^\#_{a^\#}(e',s'^\#)}} [(e', s'^\#) \mapsto \{r^\# | (e', s'^\#)\Downarrow^\#_{a^\#} r^\#\}]$$

**Lemma 4.1** (Abstract semantics as a fixpoint).

$$[\![e]\!]^\#(s^\#) = \mathrm{lfp}(\lambda a^\#.F^\#([(e, s^\#) \mapsto \emptyset] \sqcup a^\#))$$

The usual thing to do now is to connect the collecting semantics $[\![e]\!]$ with the abstract semantics $[\![e]\!]^\#$ via a Galois connection. However, we do not enforce the existence of an explicit abstraction and concretization function. Instead, we define a notion of soundness between abstract and concrete results and prove that for any $\mathsf{tick}^\#$, the abstract semantics overapproximate the collecting semantics if it starts from a sound configuration.

**Definition 4.5** ($\alpha$-soundness between results).

- Let $(V, m, t) \in \text{Result}$ and $(V^\#, m^\#, t^\#) \in \text{Result}^\#$. We do not assume that $\mathbb{T}$ and $\mathbb{T}^\#$ are concrete/abstract times.
- Let $\alpha : \mathbb{T} \to \mathbb{T}^\#$, and extend $\alpha$ to a function in $\text{Ctx} \to \text{Ctx}$ by mapping $\alpha$ over all timestamps.
- Extend $\alpha$ to a function in $(\text{Ctx} + \text{Val}) \to (\text{Ctx} + \text{Val})$ accordingly.
- Extend $\alpha$ to a function in $\text{Mem} \to \text{Mem}^\#$ by defining

$$\alpha(m) \triangleq \bigsqcup_{t \in \text{dom}(m)} [\alpha(t) \mapsto \{\alpha(m(t))\}]$$

We say that $(V^\#, m^\#, t^\#)$ is an *$\alpha$-sound approximation* of $(V, m, t)$ when $\alpha(V) = V^\#$, $\alpha(m) \sqsubseteq m^\#$, and $\alpha(t) = t^\#$.

**Definition 4.6** (Soundness between semantics).

- Let $a \in (\text{Expr} \times \text{Config}) \to \wp(\text{Result})_\bot$ and $a^\# \in (\text{Expr} \times \text{Config}^\#) \to \wp(\text{Result}^\#)_\bot$.

We say that $a^\#$ is a sound approximation of $a$ and write $a \lesssim a^\#$ if:

$$\forall e \in \text{Expr}, s \in \text{Config}, r \in \text{Result} : r \in a(e, s) \Rightarrow \exists \alpha, \alpha', s^\#, r^\# : \alpha(s) \sqsubseteq s^\# \wedge \alpha'(r) \sqsubseteq r^\# \in a^\#(e, s^\#)$$

That is, $a^\#$ is sound with respect to $a$ iff for every $(e, s), r$ pair in $a$, there exists an $\alpha$-sound pair in $a^\#$.

**Lemma 4.2** (Preservation of soundness, relation version).

- Let $s \in \text{Config}$ and $s^\# \in \text{Config}^\#$.
- Let all timestamps in the $C$ and $m$ component of $s$ be strictly less than the $t$ component.
- Let $s^\#$ be an $\alpha$-sound approximation of $s$ for some $\alpha$.

Then for all $e$,

1. If $(e, s) \Downarrow r$, then there exists $\alpha', r^\#$ such that $(e, s^\#) \Downarrow^\# r^\#$ and $\alpha'(s) \sqsubseteq s^\#$ and $\alpha'(r) \sqsubseteq r^\#$.
2. If $(e, s) \rightsquigarrow (e', s')$, then there exists $\alpha', s'^\#$ such that $(e, s^\#) \rightsquigarrow^\# (e', s'^\#)$ and $\alpha'(s) \sqsubseteq s^\#$ and $\alpha'(s') \sqsubseteq s'^\#$.

**Lemma 4.3** (Preservation of soundness).

- Let $s \in \text{Config}$ and $s^\# \in \text{Config}^\#$.
- Let all timestamps in the $C$ and $m$ component of $s$ be strictly less than the $t$ component.
- Let $s^\#$ be an $\alpha$-sound approximation of $s$ for some $\alpha$.

Then for all $e$, $[\![e]\!](s) \lesssim [\![e]\!]^\#(s^\#)$.

What's remarkable is that we did not put any constraint on the tick and $\mathsf{tick}^\#$ functions. Moreover, we can guarantee that $[\![e]\!]^\#(s^\#)$ can be computed.

**Theorem 4.1** (Finiteness of time implies finiteness of abstraction). If $\mathbb{T}^\#$ is finite,

$$\forall e, s^\# : |[\![e]\!]^\#(s^\#)| < \infty$$

Now for separate analysis, we need to define an abstract time on $\mathbb{T}_1^\# + \mathbb{T}_2^\#$, when the first time domain exports $s'^\#$ to the second time domain.

Before elaborating on how to link the time domains, we need to define the injection and deletion operators that inject the exported context into the separately analyzed results. The notation for injecting $C_1$ into $C_2$ is $C_1\langle C_2 \rangle$, similar to the plugin operator defined above.

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

$$C_1\langle C_2\rangle \triangleq C_1[\mathsf{map\_inject}(C_1, C_2)]$$

We extend the injection operator to map over all closures in a memory so that $C\langle m^\#\rangle$ can be defined naturally.

$$\mathsf{delete\_prefix}(C_1, C_2) \triangleq \begin{cases} \mathsf{delete\_prefix}(C_1', C_2') & ((C_1, C_2) = (\lambda x^t.C_1', \lambda x^t.C_2')) \\ \mathsf{delete\_prefix}(C_1', C_2') & ((C_1, C_2) = (\texttt{let}\ x^t\ C_1', \texttt{let}\ x^t\ C_2')) \\ \mathsf{delete\_prefix}(C_1', C_2') & ((C_1, C_2) = (\texttt{let}\ M\ C'\ C_1', \texttt{let}\ M\ C'\ C_2')) \\ C_2 & (\text{otherwise}) \end{cases}$$

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

$$\mathsf{delete}(C_1, C_2) \triangleq \mathsf{delete\_map}(C_1, \mathsf{delete\_prefix}(C_1, C_2))$$

The deletion operation has the expected property that $\mathsf{delete}(C, C\langle C'\rangle) = C'$.

Finally, before delving into the definition of the linked time domain, we need to define a filter function that filters the context and memory by membership in each time domain.

$$\mathsf{filter}(C, \mathbb{T}) \triangleq \begin{cases} [] & (C = []) \\ \lambda x^t.\mathsf{filter}(C', \mathbb{T}) & (C = \lambda x^t.C' \wedge t \in \mathbb{T}) \\ \texttt{let}\ x^t\ \mathsf{filter}(C', \mathbb{T}) & (C = \texttt{let}\ x^t\ C' \wedge t \in \mathbb{T}) \\ \texttt{let}\ M\ \mathsf{filter}(C', \mathbb{T})\ \mathsf{filter}(C'', \mathbb{T}) & (C = \texttt{let}\ M\ C'\ C'') \\ \mathsf{filter}(C', \mathbb{T}) & (C = \lambda x^t.C' \wedge t \notin \mathbb{T}) \\ \mathsf{filter}(C', \mathbb{T}) & (C = \texttt{let}\ x^t\ C' \wedge t \notin \mathbb{T}) \end{cases}$$

$$\mathsf{filter}(v, \mathbb{T}) \triangleq \langle \lambda x.e, \mathsf{filter}(C, \mathbb{T})\rangle \quad (v = \langle \lambda x.e, C\rangle)$$

$$\mathsf{filter}(m^\#, \mathbb{T}) \triangleq \lambda t \in \mathbb{T}.\{\mathsf{filter}(v, \mathbb{T}) | v \in m^\#(t)\}$$

**Lemma 4.4** (Linking of time domains).

- Let $(\mathbb{T}, \leq, \mathsf{tick})$ and $(\mathbb{T}', \leq', \mathsf{tick}')$ be two concrete times.
- Let $C_0 \in \mathsf{Ctx}$ be the injected context.
- Define the $\mathsf{tick}_+$ function so that the timestamps produced from the *injected* context and state are exactly the timestamps produced before injection.

$$\mathsf{tick}_+(C, m, t, x, v) \triangleq \begin{cases} \mathsf{tick}(\mathsf{filter}(C, \mathbb{T}), \mathsf{filter}(m, \mathbb{T}), t, x, \mathsf{filter}(v, \mathbb{T})) & (t \in \mathbb{T}) \\ \mathsf{tick}'(\mathsf{filter}(\mathsf{delete}(C_0, C), \mathbb{T}'), \mathsf{filter}(\mathsf{delete}(C_0, m), \mathbb{T}'), t, x, \mathsf{filter}(\mathsf{delete}(C_0, v), \mathbb{T}')) & (t \in \mathbb{T}') \end{cases}$$

Then $(\mathbb{T} + \mathbb{T}', \mathsf{tick}_+)$ is a concrete time.

**Lemma 4.5** (Preservation of timestamps under linked time).

- Let $(\mathbb{T}, \leq, \mathsf{tick})$ and $(\mathbb{T}', \leq', \mathsf{tick}')$ be two concrete times.
- Let $C_0 \in \mathsf{Ctx}$ be the injected context.
- Let $m_0 \in \mathsf{Mem}$ be the injected memory, and let

$$m; m_0 \triangleq \lambda t. \begin{cases} m(t) & (t \in \mathbb{T}') \\ m_0(t) & (t \in \mathbb{T}) \end{cases}$$

for $m \in \mathsf{Mem}$.

Now, if $(e, C, m, t) \Downarrow (V, m', t')$ under $\mathbb{T}'$, then $(e, C_0\langle C\rangle, C_0\langle m\rangle; m_0, t) \Downarrow (C_0\langle V\rangle, C_0\langle m\rangle; m_0, t')$ under $\mathbb{T} + \mathbb{T}'$.

Likewise, if $(e, C, m, t) \rightsquigarrow^* (e', C', m', t')$ under $\mathbb{T}'$, then $(e, C_0\langle C\rangle, C_0\langle m\rangle; m_0, t) \rightsquigarrow^* (e', C_0\langle C'\rangle, C_0\langle m\rangle; m_0, t')$ under $\mathbb{T} + \mathbb{T}'$.