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.

$$
\begin{array}{rcll}
x & \in & \text{ExprVar} & \\
M & \in & \text{ModVar} & \\
e & \in & \text{Expr} & \\
e & ::= & x & \textit{identifier, expression} \\
 & | & \lambda x.e & \textit{function} \\
 & | & e\ e & \textit{application} \\
 & | & e!e & \textit{linked expression} \\
 & | & \varepsilon & \textit{empty module} \\
 & | & M & \textit{identifier, module} \\
 & | & \texttt{let}\ x\ e\ e & \textit{let-binding, expression} \\
 & | & \texttt{let}\ M\ e\ e & \textit{let-binding, module}
\end{array}
$$

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.

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

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). $(\mathbb{T}, \leq, \mathrm{tick})$ is a *concrete time* when

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

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

   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.

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

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

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

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

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

2.2 The Evaluation Relation

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

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

$$[\textsc{App}]\ \dfrac{\begin{array}{c}(e_1,C,\sigma,t)\Downarrow(\langle\lambda x.e_\lambda,C_\lambda\rangle,\sigma_\lambda,t_\lambda)\\(e_2,C,\sigma_\lambda,t_\lambda)\Downarrow(v,\sigma_a,t_a)\\(e_\lambda,C_\lambda[\lambda x^{t_a}.[]],\sigma_a[t_a\mapsto v],\mathsf{tick}\,C\,\sigma_a\,t_a\,x\,v)\Downarrow(v',\sigma',t')\end{array}}{(e_1\,e_2,C,\sigma,t)\Downarrow(v',\sigma',t')}\qquad[\textsc{Linking}]\ \dfrac{\begin{array}{c}(e_1,C,\sigma,t)\Downarrow(C',\sigma',t')\\(e_2,C',\sigma',t')\Downarrow(V,\sigma'',t'')\end{array}}{(e_1!e_2,C,\sigma,t)\Downarrow(V,\sigma'',t'')}$$

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.

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

$$[\textsc{LetE}]\ \dfrac{\begin{array}{c}(e_1,C,\sigma,t)\Downarrow(v,\sigma',t')\\(e_2,C[\mathtt{let}\,x^{t'}\,[]],\sigma'[t'\mapsto v],\mathsf{tick}\,C\,\sigma'\,t'\,x\,v)\Downarrow(C',\sigma'',t'')\end{array}}{(\mathtt{let}\,x\,e_1\,e_2,C,\sigma,t)\Downarrow(C',\sigma'',t'')}\qquad[\textsc{LetM}]\ \dfrac{\begin{array}{c}(e_1,C,\sigma,t)\Downarrow(C',\sigma',t')\\(e_2,C[\mathtt{let}\,M\,C'\,[]],\sigma',t')\Downarrow(C'',\sigma'',t'')\end{array}}{(\mathtt{let}\,M\,e_1\,e_2,C,\sigma,t)\Downarrow(C'',\sigma'',t'')}$$

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 σ and initial time t, we need to collect all *reachable states* starting from (e,C,σ,t) and all values that are returned by the reachable states. We first define the reachability relation $\rightsquigarrow$.

$$[\textsc{Refl}]\ \dfrac{}{(e,C,\sigma,t)\rightsquigarrow(e,C,\sigma,t)}\qquad[\textsc{AppL}]\ \dfrac{(e_1,C,\sigma,t)\rightsquigarrow(e',C',\sigma',t')}{(e_1\,e_2,C,\sigma,t)\rightsquigarrow(e',C',\sigma',t')}$$

$$[\textsc{AppR}]\ \dfrac{\begin{array}{c}(e_1,C,\sigma,t)\Downarrow(\langle\lambda x.e_\lambda,C_\lambda\rangle,\sigma_\lambda,t_\lambda)\\(e_2,C,\sigma_\lambda,t_\lambda)\rightsquigarrow(e',C',\sigma',t')\end{array}}{(e_1\,e_2,C,\sigma,t)\rightsquigarrow(e',C',\sigma',t')}\qquad[\textsc{AppBody}]\ \dfrac{\begin{array}{c}(e_1,C,\sigma,t)\Downarrow(\langle\lambda x.e_\lambda,C_\lambda\rangle,\sigma_\lambda,t_\lambda)\\(e_2,C,\sigma_\lambda,t_\lambda)\Downarrow(v,\sigma_a,t_a)\\(e_\lambda,C_\lambda[\lambda x^{t_a}.[]],\sigma_a[t_a\mapsto v],\mathsf{tick}\,C\,\sigma_a\,t_a\,x\,v)\rightsquigarrow(e',C',\sigma',t')\end{array}}{(e_1\,e_2,C,\sigma,t)\rightsquigarrow(e',C',\sigma',t')}$$

$$[\textsc{LinkL}]\ \dfrac{(e_1,C,\sigma,t)\rightsquigarrow(e',C',\sigma',t')}{(e_1!e_2,C,\sigma,t)\rightsquigarrow(e',C',\sigma',t')}\qquad[\textsc{LinkR}]\ \dfrac{\begin{array}{c}(e_1,C,\sigma,t)\Downarrow(C',\sigma',t')\\(e_2,C',\sigma',t')\rightsquigarrow(e'',C'',\sigma'',t'')\end{array}}{(e_1!e_2,C,\sigma,t)\rightsquigarrow(e'',C'',\sigma'',t'')}$$

$$[\textsc{LetEL}]\ \dfrac{(e_1,C,\sigma,t)\rightsquigarrow(e',C',\sigma',t')}{(\mathtt{let}\,x\,e_1\,e_2,C,\sigma,t)\rightsquigarrow(e',C',\sigma',t')}\qquad[\textsc{LetER}]\ \dfrac{\begin{array}{c}(e_1,C,\sigma,t)\Downarrow(v,\sigma',t')\\(e_2,C[\mathtt{let}\,x^{t'}\,[]],\sigma'[t'\mapsto v],\mathsf{tick}\,C\,\sigma'\,t'\,x\,v)\rightsquigarrow(e'',C'',\sigma'',t'')\end{array}}{(\mathtt{let}\,x\,e_1\,e_2,C,\sigma,t)\rightsquigarrow(e'',C'',\sigma'',t'')}$$

$$[\textsc{LetML}]\ \dfrac{(e_1,C,\sigma,t)\rightsquigarrow(e',C',\sigma',t')}{(\mathtt{let}\,M\,e_1\,e_2,C,\sigma,t)\rightsquigarrow(e',C',\sigma',t')}\qquad[\textsc{LetMR}]\ \dfrac{\begin{array}{c}(e_1,C,\sigma,t)\Downarrow(C',\sigma',t')\\(e_2,C[\mathtt{let}\,M\,C'\,[]],\sigma',t')\rightsquigarrow(e'',C'',\sigma'',t'')\end{array}}{(\mathtt{let}\,M\,e_1\,e_2,C,\sigma,t)\rightsquigarrow(e'',C'',\sigma'',t'')}$$

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 both the stuck and resolved states.

Definition 3.1 (Reachable stuck states).

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

Definition 3.2 (Reachable resolved states).

$$\underline{e}^{\circ}(C,\sigma,t)\triangleq\{((e',C',\sigma',t'),(V,\sigma'',t''))|(e,C,\sigma,t)\rightsquigarrow(e',C',\sigma',t')\land(e',C'\sigma',t')\Downarrow(V,\sigma'',t'')\}$$

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

$$[\![e]\!](C,\sigma,t)\triangleq(\underline{e}^{\circ}(C,\sigma,t),\underline{e}^{\times}(C,\sigma,t))$$

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

Lemma 3.1 (Separation of linking, resolved part).

$$\underline{e_1!e_2}^\circ(C,\sigma,t) = \underline{e_1}^\circ(C,\sigma,t) \cup \bigcup_{(e_1,C,\sigma,t)\Downarrow(C',\sigma',t')} (\underline{e_2}^\circ(C',\sigma',t') \cup \underbrace{\{((e_1!e_2,C,\sigma,t),(V,\sigma'',t''))|(e_2,C',\sigma',t')\Downarrow(V,\sigma'',t'')\})}_{\text{The final evaluation}}$$

Lemma 3.2 (Separation of linking, stuck part).

$$\begin{aligned}\underline{e_1!e_2}^\times(C,\sigma,t) =&\underline{e_1}^\times(C,\sigma,t) \cup \bigcup_{(e_1,C,\sigma,t)\Downarrow(C',\sigma',t')} \underline{e_2}^\times(C',\sigma',t')\\ &\cup \{(e_1!e_2,C,\sigma,t)|\forall C',\sigma',t' \ : \ (e_1,C,\sigma,t)\Downarrow(C',\sigma',t') \Rightarrow (e_2,C',\sigma',t') \in \underline{e_2}^\times(C',\sigma',t')\}\end{aligned}$$

Now we only need a way to calculate $[\![e_2]\!](C',\sigma',t')$ by utilizing the separately calculated $[\![e_2]\!]([],\varnothing,t_0)$. The problem here is that when we calculate the two modules separately, we cannot enforce the order of the time component of each component. The order is important, because the *equivalence* of the concrete semantics depends on the fact that the tick function always generates a fresh timestamp larger than the times in the context and in the memory. The notion of equivalence between concrete configurations and the notion of soundness between the concrete and abstract configurations is essential in developing a formal guarantee that our modular separate analysis does generate some sound abstract state.

Definition 3.4 (Domain of a configuration). We define the domain of a configuration given by C,σ,t to be

$$\mathsf{Dom}(C,\sigma,t) \triangleq \{t'|t' \in C \lor t' \in \mathsf{Dom}(\sigma) \lor t' = t\}$$

Definition 3.5 (Equivalence between concrete configurations).

1. Let $(\mathbb{T},\leq,\mathsf{tick})$ and $(\mathbb{T}',\leq',\mathsf{tick}')$ be two concrete times.
2. Let (C,σ,t) be a configuration in the first time domain and (C',σ',t') be a configuration in the second time domain.

The two configurations are equivalent if there exists a strictly increasing function $f \in \mathsf{Dom}(C,\sigma,t) \to \mathsf{Dom}(C',\sigma',t')$ that translates (C,σ,t) into (C',σ',t').

$$\begin{aligned}(C,\sigma,t) \equiv (C',\sigma',t') \triangleq \exists f \in \mathsf{Dom}(C,\sigma,t) \to \mathsf{Dom}(C',\sigma',t') :&\forall t_1,t_2 \ : \ t_1 < t_2 \Rightarrow f(t_1) < f(t_2) \land\\ &f(C) = C' \land f(t) = t' \land f\circ\sigma = \sigma'\circ f\end{aligned}$$

We can prove that this relation indeed defines an equivalence relation, since such a function f automatically becomes a bijection between the two domains.

Lemma 3.3 (Preservation of equivalence).

1. Let $(\mathbb{T},\leq,\mathsf{tick})$ and $(\mathbb{T}',\leq',\mathsf{tick}')$ be two concrete times.
2. Let (C,σ,t) be a configuration in the first time domain and (C',σ',t') be a configuration in the second time domain.
3. Let all timestamps in C and σ be strictly less than t.
4. Let $(C,\sigma,t) \equiv (C',\sigma',t')$.

Then for each e, if $(e,C,\sigma,t) \Downarrow (V,\sigma'',t'')$, there exists V',σ''',t''' such that $(e,C',\sigma',t') \Downarrow (V',\sigma''',t''')$ and $(V,\sigma'',t'') \equiv (V',\sigma''',t''')$, when the equivalence between values are defined similarly to the equivalence between configurations. Likewise, if $(e,C,\sigma,t) \rightsquigarrow (e',C'',\sigma'',t'')$, there exists C''',σ''',t''' such that $(e,C',\sigma',t') \rightsquigarrow (e',C''',\sigma''',t''')$ and $(C'',\sigma'',t'') \equiv (C''',\sigma''',t''')$.

Now we need to define a concrete time on $\mathbb{T}+\mathbb{T}'$, when the first time domain exports C',σ' to the second time domain. What we want to prove that is: $(C',\sigma',t') \equiv (C',\sigma',t_0)$, when the first configuration is in $\mathbb{T}$ and the second configuration is in $\mathbb{T}+\mathbb{T}'$, with t_0 being the initial time in $\mathbb{T}'$. Then the reachable states from (e_2,C',σ',t_0) under the added domain are equivalent to the reachable states from (e_2,C',σ',t'). Thus, because we can obtain a subset of the reachable states from (e_2,C',σ',t_0) by simply *injecting* C' and σ' into the reachable states obtained from $[\![e_2]\!]([],\varnothing,t_0)$, we can compute a set *equivalent to* $[\![e_2]\!](C',\sigma',t')$ starting from the *separately analyzed* results.

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.

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

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

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

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

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

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

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

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.

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

$$\text{filter}(\sigma, \mathbb{T}) \triangleq \lambda t \in \mathbb{T}.\text{let}\ \langle \lambda x.e, C\rangle \coloneqq \sigma(t)\ \text{in}\ \langle \lambda x.e, \text{filter}(C, \mathbb{T})\rangle$$

Lemma 3.4 (Linking of time domains).

1. Let $(\mathbb{T}, \leq, \text{tick})$ and $(\mathbb{T}', \leq', \text{tick}')$ be two concrete times.
2. Let $C_0 \in \text{Ctx}\ \mathbb{T}$ be the injected context.
3. Define the $\leq_+$ order by
$$t \leq_+ t' \triangleq \begin{cases} t \leq t' & (t, t' \in \mathbb{T}) \\ t \leq' t' & (t, t' \in \mathbb{T}') \\ \text{True} & (t \in \mathbb{T}, t' \in \mathbb{T}') \\ \text{False} & (t \in \mathbb{T}', t' \in \mathbb{T}) \end{cases}$$
4. Define the tick_+ function so that the timestamps produced from the *injected* context and state are exactly the timestamps produced before injection.

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

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

Lemma 3.5 (Preservation of timestamps under linked time).

1. Let $(\mathbb{T}, \leq, \text{tick})$ and $(\mathbb{T}', \leq', \text{tick}')$ be two concrete times.
2. Let $C_0 \in \text{Ctx}\ \mathbb{T}$ be the injected context.

3. Let $\sigma_0 \in \mathrm{Mem}\ \mathbb{T}$ be the injected memory, and let

$$\sigma; \sigma_0 \triangleq \lambda t. \begin{cases} \sigma(t) & (t \in \mathbb{T}') \\ \sigma_0(t) & (t \in \mathbb{T}) \end{cases}$$

for $\sigma \in \mathrm{Mem}\ \mathbb{T}'$.

Now, if $(e, C, \sigma, t) \Downarrow (V, \sigma', t')$ under $\mathbb{T}'$, then $(e, C_0\langle C\rangle, C_0\langle\sigma\rangle; \sigma_0, t) \Downarrow (C_0\langle V\rangle, C_0\langle\sigma\rangle; \sigma_0, t')$ under $\mathbb{T} + \mathbb{T}'$.
Likewise, if $(e, C, \sigma, t) \rightsquigarrow (e', C', \sigma', t')$ under $\mathbb{T}'$, then $(e, C_0\langle C\rangle, C_0\langle\sigma\rangle; \sigma_0, t) \rightsquigarrow (e', C_0\langle C'\rangle, C_0\langle\sigma\rangle; \sigma_0, t')$ under $\mathbb{T} + \mathbb{T}'$.

Lemma 3.6 (Separate analysis).

4 Abstract Semantics

Definition 4.1 (Soundness).

Lemma 4.1 (Equivalence preserves soundness).