(Sketches of) Proofs for Modular Analysis

Joonhyup Lee

September 19, 2023

1 Part I: Semantics that Allow Open Code to be Closed Fractionally

We make the following observations:

- Most code that static analyzers deal with is *open code* that uses external values.
- Those external values are defined in a different *scope* from the code of interest.
- The different scopes are organized in term of *modules*.
- The modules are usually bound to *module names*.

Therefore, experts who write realistic analyzers are immediately faced with the problem of *closing* open code. Especially, in the case when external values are not defined in the same language, the semantics of such values must be *modelled*, either by the analysis expert or by the user of the analyzer. Since we cannot possibly model all such cases in one try, attempts to close open code must be a never-ending race of fractional advances.

If we force the analyzers to output results only in the fortunate case that all external values has already been modelled, we end up unnecessarily recomputing each time we fail to close completely. We claim that this is undesirable: the analyzer, upon meeting an open term, may just "cache" what has been computed already and "pick up" from there when that open term is resolved. No doubt, there may already be program analyzers that perform such caching, but how can we model such a computation *mathematically*? Therefore, we aim to define semantics for terms that have been fractionally closed, and prove that closing the *fractionally closed semantics* is equal to the *closed semantics*.

To illustrate what will be proven, say that e is an open term, S_2 is the set of contexts that close e fractionally, and S_1 is another set of contexts that aim to fill in the blanks. What we aim to show is:

$$[\![e]\!](S_1 \rhd S_2) = S_1 \multimap [\![e]\!]S_2$$

when $S_1 \multimap [\![e]\!]S_2$ closes the fractionally closed $[\![e]\!]S_2$ and $[\![e]\!](S_1 \rhd S_2)$ starts from the filled-in states.

In this section, we will define:

1. The *abstract syntax* of a model language with modules that can be bound to names.
2. The *semantics* of the language that is defined regardless of openness.

and sketch how to design an analysis that allows fractional specification.

1.1 Abstract Syntax

The language is basically an extension of untyped lambda calculus with modules and the linking construct. $e_1 \bowtie e_2$ means that e_1 is a module that is evaluated first to a *context*, and that e_2 is evaluated under the exported context.

Identifiers	x, M	$\in$	Var	
Expression	e	$\to$	x	value identifier
		$\mid$	$\lambda x.e$	function
		$\mid$	$e\ e$	application
		$\mid$	$e \bowtie e$	linked expression
		$\mid$	ε	empty module
		$\mid$	M	module identifier
		$\mid$	`let` $x\ e\ e$	binding expression
		$\mid$	`let` $M\ e\ e$	binding module

Figure 1: Abstract syntax of the simple module language.

$$
\begin{array}{rcll}
\text{Environment/Context} & C & \in & \text{Ctx} \\
\text{Value of expressions} & v & \in & \text{Val} \subseteq \text{Expr} \times \text{Ctx} \\
\text{Value of expressions/modules} & V & \in & \text{ValCtx} \triangleq \text{Val} \uplus \text{Ctx} \\
\text{Context} & C & \to & [] \qquad\qquad \text{empty stack} \\
 & & \mid & (x, v) :: C \qquad \text{expression binding} \\
 & & \mid & (M, C) :: C \qquad \text{module binding} \\
\text{Value of expressions} & v & \to & \langle \lambda x.e, C \rangle \qquad \text{closure}
\end{array}
$$

Figure 2: Definition of the semantic domains.

$\boxed{(e, C) \rightsquigarrow V \text{ or } (e', C')}$

$$
[\textsc{ExprID}]\ \frac{v = C(x)}{(x, C) \rightsquigarrow v} \qquad [\textsc{Fn}]\ \frac{}{(\lambda x.e, C) \rightsquigarrow \langle \lambda x.e, C \rangle} \qquad [\textsc{AppL}]\ \frac{}{(e_1\ e_2, C) \rightsquigarrow (e_1, C)} \qquad [\textsc{AppR}]\ \frac{(e_1, C) \rightsquigarrow \langle \lambda x.e_\lambda, C_\lambda \rangle}{(e_1\ e_2, C) \rightsquigarrow (e_2, C)}
$$

$$
[\textsc{AppBody}]\ \frac{\begin{array}{c}(e_1, C) \rightsquigarrow \langle \lambda x.e_\lambda, C_\lambda \rangle \\ (e_2, C) \rightsquigarrow v\end{array}}{(e_1\ e_2, C) \rightsquigarrow (e_\lambda, (x, v) :: C_\lambda)} \qquad [\textsc{App}]\ \frac{\begin{array}{c}(e_1, C) \rightsquigarrow \langle \lambda x.e_\lambda, C_\lambda \rangle \\ (e_2, C) \rightsquigarrow v \\ (e_\lambda, (x, v) :: C_\lambda) \rightsquigarrow v'\end{array}}{(e_1\ e_2, C) \rightsquigarrow v'}
$$

$$
[\textsc{LinkL}]\ \frac{}{(e_1 \bowtie e_2, C) \rightsquigarrow (e_1, C)} \qquad [\textsc{LinkR}]\ \frac{(e_1, C) \rightsquigarrow C'}{(e_1 \bowtie e_2, C) \rightsquigarrow (e_2, C')} \qquad [\textsc{Link}]\ \frac{\begin{array}{c}(e_1, C) \rightsquigarrow C' \\ (e_2, C') \rightsquigarrow V\end{array}}{(e_1 \bowtie e_2, C) \rightsquigarrow V}
$$

$$
[\textsc{Empty}]\ \frac{}{(\varepsilon, C) \rightsquigarrow C} \qquad [\textsc{ModID}]\ \frac{C' = C(M)}{(M, C) \rightsquigarrow C'}
$$

$$
[\textsc{LetEL}]\ \frac{}{(\texttt{let}\ x\ e_1\ e_2, C) \rightsquigarrow (e_1, C)} \qquad [\textsc{LetER}]\ \frac{(e_1, C) \rightsquigarrow v}{(\texttt{let}\ x\ e_1\ e_2, C) \rightsquigarrow (e_2, (x, v) :: C)} \qquad [\textsc{LetE}]\ \frac{\begin{array}{c}(e_1, C) \rightsquigarrow v \\ (e_2, (x, v) :: C) \rightsquigarrow C'\end{array}}{(\texttt{let}\ x\ e_1\ e_2, C) \rightsquigarrow C'}
$$

$$
[\textsc{LetML}]\ \frac{}{(\texttt{let}\ M\ e_1\ e_2, C) \rightsquigarrow (e_1, C)} \qquad [\textsc{LetMR}]\ \frac{(e_1, C) \rightsquigarrow C'}{(\texttt{let}\ M\ e_1\ e_2, C) \rightsquigarrow (e_2, (M, C') :: C)} \qquad [\textsc{LetM}]\ \frac{\begin{array}{c}(e_1, C) \rightsquigarrow C' \\ (e_2, (M, C') :: C) \rightsquigarrow C''\end{array}}{(\texttt{let}\ M\ e_1\ e_2, C) \rightsquigarrow C''}
$$

Figure 3: The concrete one-step transition relation.

1.2 Operational Semantics

We present the operational semantics $\rightsquigarrow$ for our language. The semantic domains are given in Figure 2 and the operational semantics is defined in Figure 3.

Our semantics relate an element ℓ of $\text{Left} \triangleq \text{Expr} \times \text{Ctx}$ with an element ρ of $\text{Right} \triangleq \text{Left} \uplus \text{ValCtx}$. Note that $C(x)$ pops the highest value that is associated with x from the stack C and $C(M)$ pops the highest context associated with M from C. The relation $\rightsquigarrow$ is unorthodox in that unlike normal big-step operaional semantics, the relation $\rightsquigarrow$ relates a configuration not only to its final result but also to intermediate configurations of which its values are required to compute the final result. Why it is defined as such is because defining a *collecting semantics* becomes much simpler.

1.3 Collecting Semantics

To define a semantics that is computable, we must formulate the collecting semantics as a least fixed point of a monotonic function that maps an element of some CPO D to D. In our case, $D \triangleq \wp(\Sigma)$ when $\Sigma \triangleq \rightsquigarrow \uplus \text{Right}$ and $\wp(S)$ is the powerset

of S. The semantics of an expression e starting from initial states in $S \subseteq \mathrm{Ctx}$ is the collection of $\ell \rightsquigarrow \rho$ and ρ derivable from initial configurations (e, C) with $C \in S$. Defining the transfer function is straightforward from the definition of the transition relation.

Definition 1.1 (Transfer function). Given $A \in D$, define

$$\mathsf{Step}(A) \triangleq \left\{ \ell \rightsquigarrow \rho, \rho \middle| \frac{A'}{\ell \rightsquigarrow \rho} \wedge A' \subseteq A \wedge \ell \in A \right\}$$

The Step function is naturally monotonic, as a "cache" A that remembers more about the intermediate proof tree will derive more results than a cache that remembers less. Now, because of Tarski's fixpoint theorem, we can formulate the collecting semantics in fixpoint form.

Definition 1.2 (Collecting semantics). Given $e \in \mathrm{Expr}$ and $S \subseteq \mathrm{Ctx}$, define:

$$[\![e]\!]S \triangleq \mathsf{lfp}(\lambda X.\mathsf{Step}(X) \cup \{(e, C) | C \in S\})$$

Note that the above definition can be defined without qualms for situations when the C in (e, C) does not close e. Then the collecting semantics will store the proof tree only up to the point the first free variable is evaluated.

1.4 Injection and Linking

We first define what it means to *fill in the blanks* of an individual $V_2 \in \mathrm{ValCtx}$ with a $C_1 \in \mathrm{Ctx}$:

$$V_2\langle C_1 \rangle \triangleq \begin{cases} C_1 & V_2 = [] \\ (x, v\langle C_1 \rangle) :: C\langle C_1 \rangle & V_2 = (x, v) :: C \\ (M, C\langle C_1 \rangle) :: C'\langle C_1 \rangle & V_2 = (M, C) :: C' \\ \langle \lambda x.e, C\langle C_1 \rangle \rangle & V_2 = \langle \lambda x.e, C \rangle \end{cases}$$

This does indeed "fill in the blanks", since:

Claim 1.1 (Fill in the Blanks). For all $C_1, C_2 \in \mathrm{Ctx}$, for each expression variable x,

$$C_2(x) = v \Rightarrow C_2\langle C_1 \rangle(x) = v\langle C_1 \rangle \text{ and } C_2(x) = \bot \Rightarrow C_2\langle C_1 \rangle(x) = C_1(x)$$

and for each module variable M,

$$C_2(M) = C \Rightarrow C_2\langle C_1 \rangle(x) = C\langle C_1 \rangle \text{ and } C_2(M) = \bot \Rightarrow C_2\langle C_1 \rangle(M) = C_1(M)$$

Sketch. Induction on C_2. □

Moreover, filling in the blanks preserves the evaluation relation $\rightsquigarrow$:

Claim 1.2 (Injection Preserves Evaluation). For all $\ell \in \mathrm{Left}$, $\rho \in \mathrm{Right}$, $\ell \rightsquigarrow \rho \Rightarrow \ell\langle C \rangle \rightsquigarrow \rho\langle C \rangle$. More explicity,

$$(\ell, \rho) \in \Sigma \Rightarrow (\ell\langle C \rangle, \rho\langle C \rangle) \in \Sigma$$

Sketch. Induction on $\rightsquigarrow$. □

Thus, we can define $\triangleright$ that injects a *set* of contexts S into a subset A of D and a semantic linking operation $\bowtie$ that does the rest of the computation:

Definition 1.3 (Injection). For $C_1 \in \mathrm{Ctx}, \ell = (e, C_2) \in \mathrm{Left}$, define $\ell\langle C_1 \rangle \triangleq (e, C_2\langle C_1 \rangle)$.
Then for $S \subseteq \mathrm{Ctx}$ and $A \in D$, define:

$$S \triangleright A \triangleq \{\rho\langle C \rangle | C \in S \wedge \rho \in A\} \cup \{\ell\langle C \rangle \rightsquigarrow \rho\langle C \rangle | C \in S \wedge \ell \rightsquigarrow \rho \in A\}$$

Definition 1.4 (Semantic Linking). For $S \subseteq \mathrm{Ctx}$ and $A \in D$, define:

$$S \bowtie A \triangleq \mathsf{lfp}(\lambda X.\mathsf{Step}(X) \cup (S \triangleright A))$$

Thus we reach the main theorem that allows "fractional closures" to be soundly defined:

Claim 1.3 (Advance). For all $e \in \mathrm{Expr}$ and $S_1, S_2 \subseteq \mathrm{Ctx}$,

$$[\![e]\!](S_1 \triangleright S_2) = S_1 \bowtie [\![e]\!]S_2$$

Proof. Let A be $\{(e,C)|C \in S_1 \rhd S_2\}$, and let B be $S_1 \rhd [\![e]\!]S_2$. Note that $A \subseteq B$ by the definition of $[\![e]\!]S_2$.

Also, let X_A be $\mathsf{lfp}(\lambda X.\mathsf{Step}(X) \cup A) = [\![e]\!](S_1 \rhd S_2)$ and let X_B be $\mathsf{lfp}(\lambda X.\mathsf{Step}(X) \cup B) = S_1 \rightarrowtriangle [\![e]\!]S_2$. By the previous lemma, we have that $B \subseteq X_A$.

Then first, X_A is a fixed point of $\lambda X.\mathsf{Step}(X) \cup B$, since

$$X_A = X_A \cup B = (\mathsf{Step}(X_A) \cup A) \cup B = \mathsf{Step}(X_A) \cup (A \cup B) = \mathsf{Step}(X_A) \cup B$$

. Then since X_B is the least fixed point, $X_B \subseteq X_A$.

Also, note that X_B is a pre-fixed point of $\lambda X.\mathsf{Step}(X) \cup A$, since

$$\mathsf{Step}(X_B) \cup A \subseteq \mathsf{Step}(X_B) \cup B = X_B$$

. D is a complete lattice, so by Tarski's fixpoint theorem, X_A is the least of all pre-fixed points. Thus, $X_A \subseteq X_B$.

Since $X_B \subseteq X_A$ and $X_A \subseteq X_B$, we have that $X_A = X_B$. □

1.5 Skeleton of a Static Analysis

Since we have defined a semantics that fully embrace *incomplete computations*, we only have to abstract our semantic operators to obtain a sound static analysis.

We require a CPO $D^\#$ that is Galois connected with D by abstraction α and concretization γ:

$$D = \wp(\Sigma) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} D^\#$$

and semantic operators $\mathsf{Step}^\#$ and $\rhd^\#$ that satisfies:

$$\mathsf{Step} \circ \gamma \subseteq \gamma \circ \mathsf{Step}^\# \qquad \rhd \circ (\gamma, \gamma) \subseteq \gamma \circ \rhd^\#$$

. Then we define $[\![e]\!]^\#$ and $\rightarrowtriangle^\#$ as:

$$[\![e]\!]^\# S^\# \triangleq \mathsf{lfp}(\lambda X^\#.\mathsf{Step}^\#(X^\#) \cup^\# \alpha\{(e,C)|C \in \gamma S^\#\}) \qquad S^\# \rightarrowtriangle^\# A^\# \triangleq \mathsf{lfp}(\lambda X^\#.\mathsf{Step}^\#(X^\#) \cup^\# (S^\# \rhd^\# A^\#))$$

which, by definition and Tarski's fixpoint theorem satisfies:

$$[\![e]\!] \circ \gamma \subseteq \gamma \circ [\![e]\!]^\# \qquad \rightarrowtriangle \circ (\gamma, \gamma) \subseteq \gamma \circ \rightarrowtriangle^\#$$

. Then we can soundly approximate fractional specifications by:

$$\begin{aligned}
S_1 \rightarrowtriangle [\![e]\!]S_2 &\subseteq S_1 \rightarrowtriangle \gamma([\![e]\!]^\# \alpha(S_2)) && (\because [\![e]\!] \subseteq \gamma \circ [\![e]\!]^\# \circ \alpha \text{ and monotonicity of } \rightarrowtriangle) \\
&\subseteq \gamma(\alpha(S_1)) \rightarrowtriangle \gamma([\![e]\!]^\# \alpha(S_2)) && (\because \mathrm{id} \subseteq \gamma \circ \alpha \text{ and monotonicity of } \rightarrowtriangle) \\
&\subseteq \gamma(\alpha(S_1) \rightarrowtriangle^\# [\![e]\!]^\# \alpha(S_2)) && (\because \rightarrowtriangle \circ (\gamma, \gamma) \subseteq \gamma \circ \rightarrowtriangle^\#)
\end{aligned}$$

2 Part II: A Simple Method to Derive Sound Analyzers

All that is left is to present an abstraction for the semantics in the previous section. Since the "depth" of the context C is unbound due to entries (x, v) also containing C in the environment part of v, we need to abstract $S \subseteq \mathrm{Ctx}$ to finitely compute an overapproximation. However, devising such an abstraction is not immediately obvious, since such abstractions must support operations satisfying $\{C(x)|C \in \gamma(C^\#)\} \subseteq \gamma(C^\#(x))$, when the operation $C^\#(x)$ reads abstract closures bound to x from the abstract context, which then again must contain abstract closures.

To break this recursive structure, we employ the common technique of introducing addresses and a memory. Thus, we extend the operational semantics of the previous section to a sematics that involve choosing a *time* domain $\mathbb{T}$ to use as addresses, and an *abstract time domain* $\overline{\mathbb{T}}$ that allow easy abstraction of the semantics via a *single* function $\bar{\alpha} : \mathbb{T} \to \overline{\mathbb{T}}$.

2.1 Semantic Domains

The domains for defining the operational semantics is extended to include the *concrete time* and *memory*. Compared with Figure 2, Figure 4 defines four more sets, $\mathbb{T}$, Mem, State, and Result to streamline the presentation. A $s = (C, m, t) \in \mathrm{State}$ corresponds to a C in Figure 2, as the pair C, m cooperates to represent the recursively defined C in the original representation without memory. Similarly, a $r = (V, m, t) \in \mathrm{Result}$ corresponds to a V in Figure 2, as the pair V, m cooperates to represent the recursively defined V.

Note that a heavy burden has been cast upon the *time* component. The time component is responsible for providing *fresh* addresses to write to in the memory, and it is also an indicator of the execution *history* up to that point. Hence, the policy for incrementing the timestamps of states decides what events are recorded in the timestamps, and the abstraction of this policy must select what events are preserved in the abstract semantics. We name this policy tick in our framework. The *type* of tick can be freely chosen, since it may choose to record any event that occurs during execution, but in this section we choose the type $\mathbb{T} \to \mathbb{T}$, the simplest possible option.

$$
\begin{array}{rcll}
\text{Time} & t & \in & \mathbb{T} \\
\text{Environment/Context} & C & \in & \text{Ctx} \\
\text{Value of expressions} & v & \in & \text{Val} \subseteq \text{Expr} \times \text{Ctx} \\
\text{Value of expressions/modules} & V & \in & \text{ValCtx} \triangleq \text{Val} \uplus \text{Ctx} \\
\text{Memory} & m & \in & \text{Mem} \triangleq \mathbb{T} \xrightarrow{\text{fin}} \text{Val} \\
\text{State} & s & \in & \text{State} \subseteq \text{Ctx} \times \text{Mem} \times \mathbb{T} \\
\text{Result} & r & \in & \text{Result} \subseteq \text{ValCtx} \times \text{Mem} \times \mathbb{T} \\
\text{Context} & C & \to & [] \qquad \text{empty stack} \\
 & & | & (x,t) :: C \qquad \text{expression binding} \\
 & & | & (M,C) :: C \qquad \text{module binding} \\
\text{Value of expressions} & v & \to & \langle \lambda x.e, C \rangle \qquad \text{closure}
\end{array}
$$

Figure 4: Definition of the semantic domains in the concrete case.

2.2 Operational Semantics

The operational semantics with memory is defined in Figure 5. One must first note that there is a problem with the definition of $\rightsquigarrow$ as it is. There are no restrictions on tick and the states (C, m, t), thus a write to the address t may overwrite an existing value that may be used for future computations. That is, (1) $t \notin \text{supp}(C, m)$, and (2) $\text{tick}(t) \notin \text{supp}(C, m) \wedge \text{tick}(t) \neq t$ must be guaranteed, when $\text{supp}(C, m)$ is the set of timestamps reachable from (C, m). To enforce this invariant upon all *valid* concrete executions defined by the relation $\rightsquigarrow$, we enforce that there be a *total order* on $\mathbb{T}$. Then the first criteria, that t must be fresh in C, m, is formalized by demanding that $C < t$ and $m < t$, when:

$$
C < t \triangleq \begin{cases} \text{True} & C = [] \\ t' < t \wedge C' < t & C = (x, t') :: C' \\ C' < t \wedge C'' < t & C = (M, C') :: C'' \end{cases}
\qquad
V < t \triangleq \begin{cases} C < t & V = \langle \_, C \rangle \\ C < t & V = C \end{cases}
$$

$$
m < t \triangleq \forall t' \in \text{dom}(m) : t' < t \wedge m(t') < t
$$

. The second criteria, that $\text{tick}(t)$ must also be fresh and must not equal t, is formalized by demanding that:

$$
t < \text{tick}(t)
$$

for all t. This condition is not as restrictive as it seems, as we can conversely think of a tick generating fresh timestamps as *inducing* a total order on $\mathbb{T}$. Also, these constraints match with the physical intuition of causality. Now, to allow only such valid transitions, we define:

$$
\text{State} \triangleq \{(C, m, t) | C < t \wedge m < t\} \qquad \text{Result} \triangleq \{(V, m, t) | V < t \wedge m < t\}
$$

as the set of *valid* states that enable tick to behave nicely. It is almost trivial that the set Left × Result, when Left $\triangleq$ Expr × State and Right $\triangleq$ Left $\uplus$ Result, is *closed* under the inductive definition of $\rightsquigarrow$. That is,

Claim 2.1 (Valid States Transition to Valid States). For all $\ell \in \text{Left}$ and ρ, if $\ell \rightsquigarrow \rho$ according to the inductive rules, $\rho \in \text{Right}$.

Sketch. Induction on the derivation of $\rightsquigarrow$. □

2.3 Collecting Semantics

The definition for the collecting semantics of the language is equal to the collecting semantics in the previous section. That is, when we let $D \triangleq \wp(\Sigma)$ where $\Sigma \triangleq \text{Right} \uplus \rightsquigarrow$,

Definition 2.1 (Transfer function). Given $A \in D$, define

$$
\text{Step}(A) \triangleq \left\{ \ell \rightsquigarrow \rho, \rho \middle| \frac{A'}{\ell \rightsquigarrow \rho} \wedge A' \subseteq A \wedge \ell \in A \right\}
$$

and

Definition 2.2 (Collecting semantics). Given $e \in \text{Expr}$ and $S \subseteq \text{State}$, define:

$$
[\![e]\!]S \triangleq \text{lfp}(\lambda X.\text{Step}(X) \cup \{(e, s) | s \in S\})
$$

$\boxed{(e, C, m, t) \rightsquigarrow (V, m', t') \text{ or } (e', C', m', t')}$

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

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

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

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

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

$$[\textsc{Link}]\ \frac{\begin{array}{c}(e_1, C, m, t) \rightsquigarrow (C', m', t') \\ (e_2, C', m', t') \rightsquigarrow (V, m'', t'')\end{array}}{(e_1 \rtimes e_2, C, m, t) \rightsquigarrow (V, m'', t'')} \qquad [\textsc{Empty}]\ \frac{}{(\varepsilon, C, m, t) \rightsquigarrow (C, m, t)} \qquad [\textsc{ModID}]\ \frac{C' = C(M)}{(M, C, m, t) \rightsquigarrow (C', m, t)}$$

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

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

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

$$[\textsc{LetE}]\ \frac{\begin{array}{c}(e_1, C, m, t) \rightsquigarrow (v, m', t') \\ (e_2, (x, t') :: C, m'[t' \mapsto v], \mathsf{tick}(t')) \rightsquigarrow (C', m'', t'')\end{array}}{(\mathtt{let}\ x\ e_1\ e_2, C, m, t) \rightsquigarrow (C', m'', t'')} \qquad [\textsc{LetM}]\ \frac{\begin{array}{c}(e_1, C, m, t) \rightsquigarrow (C', m', t') \\ (e_2, (M, C') :: C, m', t') \rightsquigarrow (C'', m'', t'')\end{array}}{(\mathtt{let}\ M\ e_1\ e_2, C, m, t) \rightsquigarrow (C'', m'', t'')}$$

Figure 5: The concrete one-step transition relation.

2.4 Abstract Semantics

As promised, we present a way to simply abstract the concrete semantics via a finite abstraction of the time component. For this purpose, we choose a finite *abstract time* domain $\overline{\mathbb{T}}$ that is connected to the concrete time domain via an auxiliary

$$
\begin{array}{rrcll}
\text{Abstract Time} & \dot{\overline{t}} & \in & \overline{\mathbb{T}} & \\
\text{Environment/Context} & \overline{C} & \in & \overline{\text{Ctx}} & \\
\text{Value of expressions} & \dot{\overline{v}} & \in & \overline{\text{Val}} \subseteq \text{Expr} \times \overline{\text{Ctx}} & \\
\text{Value of expressions/modules} & \overline{V} & \in & \overline{\text{ValCtx}} \triangleq \overline{\text{Val}} \uplus \overline{\text{Ctx}} & \\
\text{Abstract Memory} & \overline{m} & \in & \overline{\text{Mem}} \triangleq \overline{\mathbb{T}} \xrightarrow{\text{fin}} \wp(\overline{\text{Val}}) & \\
\text{Abstract State} & \dot{\overline{s}} & \in & \overline{\text{State}} \triangleq \overline{\text{Ctx}} \times \overline{\text{Mem}} \times \overline{\mathbb{T}} & \\
\text{Abstract Result} & \dot{\overline{r}} & \in & \overline{\text{Result}} \triangleq \overline{\text{ValCtx}} \times \overline{\text{Mem}} \times \overline{\mathbb{T}} & \\
\text{Abstract Tick} & \overline{\text{tick}} & \in & \overline{\text{Tick}} \triangleq \overline{\text{State}} \times \text{Var} \times \overline{\text{Val}} \to \overline{\mathbb{T}} & \\
\text{Context} & \overline{C} & \to & [] & \text{empty stack} \\
 & & | & (x, \dot{\overline{t}}) :: \overline{C} & \text{expression binding} \\
 & & | & (M, \overline{C}) :: \overline{C} & \text{module binding} \\
\text{Value of expressions} & \dot{\overline{v}} & \to & \langle \lambda x.e, \overline{C} \rangle & \text{closure}
\end{array}
$$

Figure 6: Definition of the semantic domains in the abstract case.

function $\dot{\overline{\alpha}} : \mathbb{T} \to \overline{\mathbb{T}}$. Since the policy to update the timestamp must also be compatible with respect to $\dot{\overline{\alpha}}$, we require the $\overline{\text{tick}} : \overline{\mathbb{T}} \to \overline{\mathbb{T}}$ function to satisfy $\dot{\overline{\alpha}} \circ \text{tick} = \overline{\text{tick}} \circ \dot{\overline{\alpha}}$.

Then the operational semantics can be abstracted directly, with modifications only in the *update* of the memory and *reads* from the memory. The memory update operation is defined as a weak update, that is:

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

and the read from the memory returns a set of closures with abstract addresses, allowing transitions to any value within that set. The full definition for the abstract version of the operational semantics $\overline{\rightsquigarrow}$ is in Figure 7. $\overline{\rightsquigarrow} \subseteq \overline{\text{Left}} \times \overline{\text{Right}}$, when $\overline{\text{Left}} \triangleq \text{Expr} \times \overline{\text{State}}$ and $\overline{\text{Right}} \triangleq \overline{\text{Left}} \uplus \overline{\text{Result}}$.

We note that the abstract operational semantics is a sound approximation of the concrete semantics in the operational sense, since if we extend $\dot{\overline{\alpha}}$ as:

$$
\dot{\overline{\alpha}}(C) \triangleq \begin{cases} [] & C = [] \\ (x, \dot{\overline{\alpha}}(t)) :: \dot{\overline{\alpha}}(C') & C = (x,t) :: C' \\ (M, \dot{\overline{\alpha}}(C')) :: \dot{\overline{\alpha}}(C'') & C = (M, C') :: C'' \end{cases} \qquad \dot{\overline{\alpha}}(V) \triangleq \begin{cases} \langle e, \dot{\overline{\alpha}}(C) \rangle & V = \langle e, C \rangle \\ \dot{\overline{\alpha}}(C) & V = C \end{cases}
$$

$$
\dot{\overline{\alpha}}(m) \triangleq \lambda \dot{\overline{t}}.\{\dot{\overline{\alpha}}(m(t)) | \dot{\overline{\alpha}}(t) = \dot{\overline{t}} \wedge t \in \text{dom}(m)\}
$$

and define $\dot{\overline{\alpha}}(r)$ for $r \in \text{Result}$ by mapping over each component, we have:

Claim 2.2 (Operational Soundness). For all $\ell \in \text{Left}$ and $\rho \in \text{Right}$, if $\ell \rightsquigarrow \rho$ then $\dot{\overline{\alpha}}(\ell) \overline{\rightsquigarrow} \dot{\overline{\alpha}}(\rho)$.

Sketch. Induction on $\rightsquigarrow$. □

Then if we define $D^\# \triangleq \wp(\overline{\Sigma})$, when $\overline{\Sigma} \triangleq \overline{\text{Right}} \uplus \overline{\rightsquigarrow}$, we can establish a Galois connection between D and $D^\#$. The abstraction and concretization functions are given by:

Definition 2.3 (Abstraction and Concretization). Define $\alpha : D \to D^\#$ and $\gamma : D^\# \to D$ by:

$$
\alpha(A) \triangleq \{\dot{\overline{\alpha}}(\ell) \overline{\rightsquigarrow} \dot{\overline{\alpha}}(\rho) | \ell \rightsquigarrow \rho \in A\} \cup \{\dot{\overline{\alpha}}(\rho) | \rho \in A\}
$$

$$
\gamma(A^\#) \triangleq \{\ell \rightsquigarrow \rho | \dot{\overline{\alpha}}(\ell) \overline{\rightsquigarrow} \dot{\overline{\alpha}}(\rho) \in A^\#\} \cup \{\rho | \dot{\overline{\alpha}}(\rho) \in A^\#\}
$$

Then it is straightforward to see that:

Claim 2.3 (Galois Connection). $\wp(\Sigma) = D \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} D^\# = \wp(\overline{\Sigma})$. That is, $\forall A \in D, A^\# \in D^\# : \alpha(A) \subseteq A^\# \Leftrightarrow A \subseteq \gamma(A^\#)$.

Sketch. Straightforward from the definitions of α and γ. □

The definition for the abstract fixpoint semantics is naturally connected soundly with the collecting semantics.

Definition 2.4 (Abstract transfer function). Given $A^\# \in D^\#$, define

$$
\text{Step}^\#(A^\#) \triangleq \left\{ \dot{\overline{\ell}} \overline{\rightsquigarrow} \overline{\rho}, \overline{\rho} \middle| \frac{A'^\#}{\dot{\overline{\ell}} \overline{\rightsquigarrow} \overline{\rho}} \wedge A'^\# \subseteq A^\# \wedge \dot{\overline{\ell}} \in A^\# \right\}
$$

$\boxed{(e, \overline{C}, \overline{m}, \overline{t}) \overline{\rightsquigarrow} (\overline{V}, \overline{m'}, \overline{t'}) \text{ or } (e', \overline{C'}, \overline{m'}, \overline{t'})}$

$$[\textsc{ExprID}]\ \frac{\overline{t_x} = \mathsf{addr}(\overline{C}, x) \qquad \overline{v} \in \overline{m}(\overline{t_x})}{(x, \overline{C}, \overline{m}, \overline{t}) \overline{\rightsquigarrow} (\overline{v}, \overline{m}, \overline{t})} \qquad [\textsc{Fn}]\ \frac{}{(\lambda x.e, \overline{C}, \overline{m}, \overline{t}) \overline{\rightsquigarrow} (\langle \lambda x.e, \overline{C} \rangle, \overline{m}, \overline{t})}$$

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

$$[\textsc{AppBody}]\ \frac{\begin{array}{c}(e_1, \overline{C}, \overline{m}, \overline{t}) \overline{\rightsquigarrow} (\langle \lambda x.e_\lambda, \overline{C_\lambda} \rangle, \overline{m_\lambda}, \overline{t_\lambda}) \\ (e_2, \overline{C}, \overline{m_\lambda}, \overline{t_\lambda}) \overline{\rightsquigarrow} (v, \overline{m_a}, \overline{t_a})\end{array}}{(e_1\ e_2, \overline{C}, \overline{m}, \overline{t}) \overline{\rightsquigarrow} (e_\lambda, (x, \overline{t_a}) :: \overline{C_\lambda}, \overline{m_a}[\overline{t_a \mapsto v}], \overline{\mathsf{tick}(t_a)})}$$

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

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

$$[\textsc{Link}]\ \frac{\begin{array}{c}(e_1, \overline{C}, \overline{m}, \overline{t}) \overline{\rightsquigarrow} (\overline{C'}, \overline{m'}, \overline{t'}) \\ (e_2, \overline{C'}, \overline{m'}, \overline{t'}) \overline{\rightsquigarrow} (\overline{V}, \overline{m''}, \overline{t''})\end{array}}{(e_1 \bowtie e_2, \overline{C}, \overline{m}, \overline{t}) \overline{\rightsquigarrow} (\overline{V}, \overline{m''}, \overline{t''})} \qquad [\textsc{Empty}]\ \frac{}{(\varepsilon, \overline{C}, \overline{m}, \overline{t}) \overline{\rightsquigarrow} (\overline{C}, \overline{m}, \overline{t})}$$

$$[\textsc{ModID}]\ \frac{\overline{C'} = \mathsf{ctx}(\overline{C}, M)}{(M, \overline{C}, \overline{m}, \overline{t}) \overline{\rightsquigarrow} (\overline{C'}, \overline{m}, \overline{t})} \qquad [\textsc{LetEL}]\ \frac{}{(\mathtt{let}\ x\ e_1\ e_2, \overline{C}, \overline{m}, \overline{t}) \overline{\rightsquigarrow} (e_1, \overline{C}, \overline{m}, \overline{t})}$$

$$[\textsc{LetER}]\ \frac{(e_1, \overline{C}, \overline{m}, \overline{t}) \overline{\rightsquigarrow} (\overline{v}, \overline{m'}, \overline{t'})}{(\mathtt{let}\ x\ e_1\ e_2, \overline{C}, \overline{m}, \overline{t}) \overline{\rightsquigarrow} (e_2, (x, \overline{t'}) :: \overline{C}, \overline{m'}[\overline{t' \mapsto v}], \overline{\mathsf{tick}(t')})}$$

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

$$[\textsc{LetE}]\ \frac{\begin{array}{c}(e_1, \overline{C}, \overline{m}, \overline{t}) \overline{\rightsquigarrow} (\overline{v}, \overline{m'}, \overline{t'}) \\ (e_2, (x, \overline{t'}) :: \overline{C}, \overline{m'}[\overline{t' \mapsto v}], \overline{\mathsf{tick}(t')}) \overline{\rightsquigarrow} (\overline{C'}, \overline{m''}, \overline{t''})\end{array}}{(\mathtt{let}\ x\ e_1\ e_2, \overline{C}, \overline{m}, \overline{t}) \overline{\rightsquigarrow} (\overline{C'}, \overline{m''}, \overline{t''})}$$

$$[\textsc{LetM}]\ \frac{\begin{array}{c}(e_1, \overline{C}, \overline{m}, \overline{t}) \overline{\rightsquigarrow} (\overline{C'}, \overline{m'}, \overline{t'}) \\ (e_2, (M, \overline{C'}) :: \overline{C}, \overline{m'}, \overline{t'}) \overline{\rightsquigarrow} (\overline{C''}, \overline{m''}, \overline{t''})\end{array}}{(\mathtt{let}\ M\ e_1\ e_2, \overline{C}, \overline{m}, \overline{t}) \overline{\rightsquigarrow} (\overline{C''}, \overline{m''}, \overline{t''})}$$

Figure 7: The abstract one-step transition relation.

Definition 2.5 (Abstract semantics). Given $e \in \text{Expr}$ and $S^\# \subseteq \overline{\text{State}}$, define:

$$[\![e]\!]^\# S^\# \triangleq \mathsf{lfp}(\lambda X^\#.\mathsf{Step}^\#(X^\#) \cup \{(e, \overline{s}) | \overline{s} \in S^\#\})$$

Then we can prove that:

Claim 2.4 (Soundness of $\mathsf{Step}^\#$). $\mathsf{Step} \circ \gamma \subseteq \gamma \circ \mathsf{Step}^\#$

Proof. From operational soundness, $\alpha \circ \mathsf{Step} \subseteq \mathsf{Step}^\# \circ \alpha$. Thus, $\alpha \circ \mathsf{Step} \circ \gamma \subseteq \mathsf{Step}^\# \circ \alpha \circ \gamma$, since γ is monotonic. Since $\alpha \circ \gamma \subseteq \mathsf{id}$ by Galois connection and $\mathsf{Step}^\#$ is monotonic, we have that $\alpha \circ \mathsf{Step} \circ \gamma \subseteq \mathsf{Step}^\#$. By Galois connection, this is equivalent to $\mathsf{Step} \circ \gamma \subseteq \gamma \circ \mathsf{Step}^\#$. □

2.5 Computability of the Abstract Semantics

Now we can say that $[\![e]\!]^\#\alpha(S)$ is a sound abstraction of $[\![e]\!]S$. However, is it true that $[\![e]\!]^\#\alpha(S)$ is finitely computable? Note that conceptually, all reachable configurations are derived from some closed expression e evaluated from the empty context $[]$ and empty memory $\emptyset$. We claim that in such situations, when the abstract semantics is computed from a finite set $S^\#$ of initial states, the resulting computation $[\![e]\!]^\#S^\#$ has finite cardinality.

Since $\overline{\mathbb{T}}$ is finite, all we have to prove is that all reachable *signatures* are finite. What we mean by a *signature* is a context that is stripped of all timestamps. Explicitly, we mean an element of an inductively defined set Sig given by $X \to [] \mid x :: X \mid (M, X) :: X$. Then we may inductively define $\lfloor C \rfloor$ and $\lfloor \overline{C} \rfloor$ to be the signatures that are obtained by stripping all timestamps from C and $\overline{C}$. Moreover, we may define $\lfloor m \rfloor \triangleq \{\lfloor C \rfloor | \exists t : \langle \_, C \rangle = m(t)\}$ and $\lfloor \overline{m} \rfloor \triangleq \{\lfloor \overline{C} \rfloor | \exists \dot{t} : \langle \_, \overline{C} \rangle \in \overline{m}(\dot{t})\}$ to be all signatures in a memory. Finally, we may define $\lfloor \rho \rfloor$ as the union of $\lfloor C \rfloor$ and $\lfloor m \rfloor$, when C is the context component of ρ and m is the memory component of ρ. $\lfloor \overline{\rho} \rfloor$ can be analogously defined.

If we can prove that for all e and $\dot{s}$, $\bigcup_{(e,\dot{s}) \overline{\rightsquigarrow}^* \overline{\rho}} \lfloor \overline{\rho} \rfloor \subseteq X$, when X is a finite set containing all reachable signatures from $(e, \dot{s})$, we can show that all reachable $\overline{\rho}$s are finite, since $\overline{\mathbb{T}}$ is finite. It turns out, since the signature of the modules that are pushed into the stack C can be accurately inferred from the definition of the operational semantics, we can *compute* such an X. Thus we have:

Claim 2.5 (Computability of the Abstract Semantics). If $S^\#$ is finite, $[\![e]\!]^\#S^\#$ is finite.

Sketch. By existence of a function that computes all reachable signatures and by induction on $\overline{\rightsquigarrow}$.

To elaborate, let this function be called f. We need two lemmas, the first being $\forall \overline{\rho} : \lfloor \overline{\rho} \rfloor \subseteq f(\overline{\rho})$ and the second being $\forall \dot{\ell}, \overline{\rho} : \dot{\ell} \overline{\rightsquigarrow} \overline{\rho} \Rightarrow f(\overline{\rho}) \subseteq f(\dot{\ell})$.

If we have the lemmas, then for all $\overline{\rho}$ such that $(e, \dot{s}) \overline{\rightsquigarrow}^* \overline{\rho}$, $\lfloor \overline{\rho} \rfloor \subseteq f(\rho) \subseteq f(\ell)$, thus all signatures in $[\![e]\!]^\#S^\#$ can be bound by $X \triangleq \{f(e, \dot{s}) | (\dot{s}, \_) \in S^\#\}$, which is finite. The lemmas are formalized in Coq. □

We stress that this is a nontrivial result, since our definition of C allows contexts to be pushed onto the stack. The result holds only because the shape of the stack can be deterministically inferred according to the semantics of our language. Thus, in languages which can be annotated with explicit signatures, this property will hold, even with features such as functors or first-class modules.

3 Part III: Formalizing Linking in the Semantics with Memory

In this section, we define how linking between different time domains is performed. First, we assume two time domains $(\mathbb{T}_1, \leq_1, \overline{\mathbb{T}}_1, \overline{\alpha}_1)$ and $(\mathbb{T}_2, \leq_2, \overline{\mathbb{T}}_2, \overline{\alpha}_2)$. Then we can define $(\mathbb{T}_+, \leq_+, \overline{\mathbb{T}}_+, \overline{\alpha}_+)$ by:

$$\mathbb{T}_+ \triangleq 2\mathbb{Z} \times \mathbb{T}_1 \uplus (2\mathbb{Z}+1) \times \mathbb{T}_2 \quad t_+ \leq_+ t'_+ \triangleq \text{lexicographic} \quad \overline{\mathbb{T}}_+ \triangleq \overline{\mathbb{T}}_1 + \overline{\mathbb{T}}_2 \quad \overline{\alpha}_+(t_+) \triangleq \begin{cases} (0, \overline{\alpha}_1(t_1)) & t_+ = (2n, t_1) \\ (1, \overline{\alpha}_2(t_2)) & t_+ = (2n+1, t_2) \end{cases}$$

when $X + Y$ is the separated sum of X and Y. A $\overline{\alpha}_+^{-1} \in \mathbb{T}_+ \times \overline{\mathbb{T}}_+ \to \mathbb{T}_+$ that satisfies the two conditions can be constructed by using some $\overline{\alpha}_1^{-1}$, $\overline{\alpha}_2^{-1}$, $0_1 \in \mathbb{T}_1$ and $0_2 \in \mathbb{T}_2$:

$$\overline{\alpha}_+^{-1}(t_+, \dot{t}_+) \triangleq \begin{cases} (2n, \overline{\alpha}_1^{-1}(t_1, \dot{t}_1)) & t_+ = (2n, t_1) \wedge \dot{t}_+ = \dot{t}_1 \\ (2n+1, \overline{\alpha}_2^{-1}(t_2, \dot{t}_2)) & t_+ = (2n+1, t_2) \wedge \dot{t}_+ = \dot{t}_2 \\ (2n+1, \overline{\alpha}_2^{-1}(0_2, \dot{t}_2)) & t_+ = (2n, t_1) \wedge \dot{t}_+ = \dot{t}_2 \\ (2n+2, \overline{\alpha}_1^{-1}(0_1, \dot{t}_1)) & t_+ = (2n+1, t_2) \wedge \dot{t}_+ = \dot{t}_1 \end{cases}$$

Thus $(\mathbb{T}_+, \leq_+, \overline{\mathbb{T}}_+, \overline{\alpha}_+)$ satisfies our requirements.

The rest of this section explains how, given two initial conditions $S_1 \subseteq \mathsf{State}_1 \times \mathsf{Tick}_1$ and $S_2 \subseteq \mathsf{State}_2 \times \mathsf{Tick}_2$, the semantics starting from the *injected* initial state $S_1 \rhd S_2$ is equal to S_1 *linked* with the semantics computed in advance from S_2. That is, we want to prove:

$$[\![e]\!](S_1 \rhd S_2) = S_1 \bowtie [\![e]\!]S_2$$

and

$$[\![e]\!]^\#(S_1^\# \rhd^\# S_2^\#) = S_2^\# \bowtie^\# [\![e]\!]^\# S_2^\#$$

Before descending into details, we want to clarify that all C, m, t in this section are assumed to be living in the linked time domain. Thus, when we have a $C \in \mathrm{Ctx}_1$, we write C to also mean an element of Ctx_+, with all timestamps equal to its original in Ctx_1 in the second coordinate and with the first coordinate fixed to 0. Likewise, for a $C \in \mathrm{Ctx}_2$, we write C to also mean an element of Ctx_+, with all timestamp equal to its original in Ctx_2 in the second coordinate and with the second coordinate fixed to 1. The same assumption applies for elements in $\mathrm{Mem}_1, \mathrm{Mem}_2, \mathbb{T}_1, \mathbb{T}_2$.

3.1 Definitions

3.1.1 Injection and Deletion

We want to define what it means to *inject* an external $S_1 \subseteq \mathrm{State}_1 \times \mathrm{Tick}_1$ into an assumed $S_2 \subseteq \mathrm{State}_2 \times \mathrm{Tick}_2$. Naturally, we must first define elementwise injection $\triangleright$ between $(s_1, \mathrm{tick}_1) \in S_1$ and $(s_2, \mathrm{tick}_2) \in S_2$ and map this over all pairs in $S_1 \times S_2$. What properties must $(s_+, \mathrm{tick}_+) = (s_1, \mathrm{tick}_1) \triangleright (s_2, \mathrm{tick}_2)$ satisfy?

Consider the case when we did not assume anything, that is, when $s_2 = ([], \emptyset, 0)$. Then first, we expect that $s_+ \cong s_1$. Second, the tick_+ function under s_+ must preserve the transitions made by tick_2 under s_2. That is, if $(e, s_2) \rightsquigarrow^*_{\mathrm{tick}_2} (e', s_2')$, then $(s_+', \mathrm{tick}_+') = (s_1, \mathrm{tick}_1) \triangleright (s_2', \mathrm{tick}_2)$ must satisfy $\mathrm{tick}_+ = \mathrm{tick}_+'$ and $(e, s_+) \rightsquigarrow^*_{\mathrm{tick}_+} (e', s_+')$. This is because we want all transitions after injecting the exported states into the semantics calculated in advance to be valid transitions.

As the first step in defining $\triangleright$, we first define the injection operator for contexts, when $C_2\langle C_1\rangle$ "fills in the blank" in C_2 with C_1. The deletion operator $C_2\langle C_1\rangle^{-1}$, which "digs out" C_1 from C_2, is also defined. This definition can be extended

$$C_2\langle C_1\rangle \triangleq \begin{cases} C_1 & C_2 = [] \\ (x,t) :: C\langle C_1\rangle & C_2 = (x,t) :: C \\ (M, C\langle C_1\rangle) :: C'\langle C_1\rangle & C_2 = (M,C) :: C' \end{cases} \quad C_2\langle C_1\rangle^{-1} \triangleq \begin{cases} [] & \bar{\alpha}_+(C_2) = \bar{\alpha}_+(C_1) \\ [] & C_2 = [] \\ (x,t) :: C\langle C_1\rangle^{-1} & C_2 = (x,t) :: C \\ (M, C\langle C_1\rangle^{-1}) :: C'\langle C_1\rangle^{-1} & C_2 = (M,C) :: C' \end{cases}$$

Figure 8: Definition of the injection operator $C_2\langle C_1\rangle$ and the deletion operator $C_2\langle C_1\rangle^{-1}$. Cases are ordered by precedence. For example, we check for $\bar{\alpha}_+(C_2) = \bar{\alpha}_+(C_1)$ first when deleting C_1 from C_2.

to injection and deletion between abstract contexts as well.

Naturally, we expect tick_+ to increment timestamps in $\mathbb{T}_+$ by using tick_1 for timestamps in $2\mathbb{Z} \times \mathbb{T}_1$, and by using tick_2 for timestamps in $(2\mathbb{Z}+1) \times \mathbb{T}_2$. However, the context and memory will contain timestamps originating from both $\mathbb{T}_1$ and $\mathbb{T}_2$. Therefore, we need to define the filter operations $C.1$ and $C.2$ which selects only timestamps from the time domain of interest. This definition can be extended to filter out abstract timestamps as well.

$$C.i \triangleq \begin{cases} [] & C = [] \\ (x,t') :: C'.i & C = (x,t) :: C' \wedge t = (2n-1+i, t') \\ C'.i & C = (x,t) :: C' \wedge t = (2n+i, t') \\ (M, C'.i) :: C''.i & C = (M, C') :: C'' \end{cases} \qquad V.i \triangleq \begin{cases} C.i & V = C \\ \langle e, C.i\rangle & V = \langle e, C\rangle \end{cases}$$

$$m.i \triangleq \bigcup_{(2n-1+i,t) \in \mathrm{dom}(m)} \{t \mapsto m(t).i\}$$

Figure 9: Definition for the filter operations $(i = 1, 2)$.

Note that $m.i$ cannot be well-defined for memories that contain addresses with overlapping second coordinates. That is, in the case when m can be accessed with both $(0, t_1)$ and $(2, t_1)$, $m.1$ is not well-defined. In this case, we leave $m.i$ as undefined. This means that the mapping $m \mapsto m.i$ is a partial function that is defined for only memories with distinct second coordinates. Luckily, in the case that m is derived from injection, all timestamps in $2\mathbb{Z} \times \mathbb{T}_1$ will be of the form $(0, \_)$, and all timestamps in $(2\mathbb{Z}+1) \times \mathbb{T}_2$ will be of the form $(1, \_)$. Thus, $m.i$ will always be defined for our purposes. Of course, $\overline{m.i}$ is always defined, since the only even first coordinate is 0, and the only odd first coordinate is 1.

Moving on, we extend $\_\langle\_\rangle$ and $\_\langle\_\rangle^{-1}$:

Definition 3.1 (Filling in the Blanks). Let $s_1 = (C_1, m_1, t_1) \in \mathrm{State}_1$ and $r_2 = (V_2, m_2, t_2) \in \mathrm{Result}_2$. Then we define:

$$V_2\langle C_1\rangle \triangleq \begin{cases} C_2\langle C_1\rangle & V_2 = C_2 \\ \langle e, C_2\langle C_1\rangle\rangle & V_2 = \langle e, C_2\rangle \end{cases} \qquad \begin{aligned} m_2\langle C_1\rangle &\triangleq \bigcup_{t \in \mathrm{dom}(m_2)} \{t \mapsto m_2(t)\langle C_1\rangle\} \\ r_2\langle s_1\rangle &\triangleq (V_2\langle C_1\rangle, m_1 \cup m_2\langle C_1\rangle, t_2) \end{aligned}$$

$$V_2\langle C_1\rangle^{-1} \triangleq \begin{cases} C_2\langle C_1\rangle^{-1} & V_2 = C_2 \\ \langle e, C_2\langle C_1\rangle^{-1}\rangle & V_2 = \langle e, C_2\rangle \end{cases} \quad m_2\langle C_1\rangle^{-1} \triangleq \bigcup_{t \in \mathrm{dom}(m_2)} \left\{t \mapsto m_2(t)\langle C_1\rangle^{-1}\right\}$$

Naturally, these definitions extend to their abstract counterparts.

Note that $r_2\langle s_1\rangle \in \text{Result}_+$. That is, it is time-bounded. Now we only have to define tick_+ which preserves transitions made before injection. The definition for $\triangleright$ is:

Definition 3.2 (Elementwise Concrete Injection). Let $s_1 = (C_1, m_1, t_1) \in \text{State}_1$, $\text{tick}_1 \in \text{Tick}_1$, $r_2 = (V_2, m_2, t_2) \in \text{Result}_2$, and $\text{tick}_2 \in \text{Tick}_2$. We define $(s_1, \text{tick}_1) \triangleright (r_2, \text{tick}_2) \triangleq (r_2\langle s_1\rangle, \text{tick}_+) \in \text{Result}_+ \times \text{Tick}_+$, when tick_+ is given by:

$$\text{tick}_+(C, m, t, x, v) \triangleq \begin{cases} (2n, \text{tick}_1(C.1, m.1, t_1, x, v.1)) & t = (2n, t_1) \\ (2n+1, \text{tick}_2(C\langle C_1\rangle^{-1}.2, m\langle C_1\rangle^{-1}.2, t_2, x, v\langle C_1\rangle^{-1}.2)) & t = (2n+1, t_2) \\ \bar{\alpha}_+^{-1}(t, \overline{\text{tick}}_+(\bar{\alpha}_+(C), \bar{\alpha}_+(m), \bar{\alpha}_+(t), x, \bar{\alpha}_+(v))) & m.i = \bot \end{cases}$$

where

$$\overline{\text{tick}}_+(\bar{C}, \bar{m}, \dot{t}, x, \bar{v}) \triangleq \begin{cases} (0, \bar{\alpha}_1(\text{tick}_1)(\bar{C}.1, \bar{m}.1, \dot{t}_1, x, \bar{v}.1)) & \dot{t} = (0, \dot{t}_1) \\ (1, \bar{\alpha}_2(\text{tick}_2)(\bar{C}\langle\bar{\alpha}_1(C_1)\rangle^{-1}.2, \bar{m}\langle\bar{\alpha}_1(C_1)\rangle^{-1}.2, \dot{t}_2, x, \bar{v}\langle\bar{\alpha}_1(C_1)\rangle^{-1}.2)) & \dot{t} = (1, \dot{t}_2) \end{cases}$$

satisfies $\bar{\alpha}_+ \circ \text{tick}_+ = \overline{\text{tick}}_+ \circ \bar{\alpha}_+$.

Definition 3.3 (Elementwise Abstract Injection). Let $\dot{s}_1 = (\bar{C}_1, \bar{m}_1, \dot{t}_1) \in \overline{\text{State}}_1$, $\overline{\text{tick}}_1 \in \overline{\text{Tick}}_1$, $\dot{r}_2 = (\bar{V}_2, \bar{m}_2, \dot{t}_2) \in \overline{\text{Result}}_2$, and $\overline{\text{tick}}_2 \in \overline{\text{Tick}}_2$. We define $(\dot{s}_1, \overline{\text{tick}}_1) \bar{\triangleright} (\dot{r}_2, \overline{\text{tick}}_2) \triangleq (\dot{r}_2\langle\dot{s}_1\rangle, \overline{\text{tick}}_+) \in \overline{\text{Result}}_+ \times \overline{\text{Tick}}_+$, when $\overline{\text{tick}}_+$ is given by:

$$\overline{\text{tick}}_+(\bar{C}, \bar{m}, \dot{t}, x, \bar{v}) \triangleq \begin{cases} (0, \overline{\text{tick}}_1(\bar{C}.1, \bar{m}.1, \dot{t}_1, x, \bar{v}.1)) & \dot{t} = (0, \dot{t}_1) \\ (1, \overline{\text{tick}}_2(\bar{C}\langle\bar{C}_1\rangle^{-1}.2, \bar{m}\langle\bar{C}_1\rangle^{-1}.2, \dot{t}_2, x, \bar{v}\langle\bar{C}_1\rangle^{-1}.2)) & \dot{t} = (1, \dot{t}_2) \end{cases}$$

Since tick_+ digs out C_1 from the memory and context, timestamps produced by tick_+ after injection will look at only the parts before injection. Thus, it will produce the same timestamps that were produced by tick_2 under S_2. This is why injection into transitions computed in advance are valid as transitions beginning from injected states.

3.1.2 Semantic Linking

$$(s_1, \text{tick}_1) \triangleright (\rho_2, \text{tick}_2) \triangleq \begin{cases} (r_+, \text{tick}_+) & \rho_2 = r_2 \wedge (r_+, \text{tick}_+) = (s_1, \text{tick}_1) \triangleright (r_2, \text{tick}_2) \\ ((e, s_+), \text{tick}_+) & \rho_2 = \ell_2 = (e, s_2) \wedge (s_+, \text{tick}_+) = (s_1, \text{tick}_1) \triangleright (s_2, \text{tick}_2) \end{cases}$$

$$(s_1, \text{tick}_1) \triangleright (\ell_2 \rightsquigarrow_{\text{tick}_2} \rho_2) \triangleq \ell_+ \rightsquigarrow_{\text{tick}_+} \rho_+$$
$$\text{where } (\ell_+, \text{tick}_+) = (s_1, \text{tick}_1) \triangleright (\ell_2, \text{tick}_2) \wedge (\rho_+, \text{tick}_+) = (s_1, \text{tick}_1) \triangleright (\rho_2, \text{tick}_2)$$

Figure 10: Extension of $\triangleright$ to define injection into an element of D_2.

Definition 3.4 (Concrete Injection). Let $S_1 \subseteq \text{State}_1 \times \text{Tick}_1$ and $A_2 \subseteq D_2$. Then we define:

$$S_1 \triangleright A_2 \triangleq \{p \triangleright q | p \in S_1 \wedge q \in A_2\}$$

Definition 3.5 (Abstract Injection). Let $S_1^\# \subseteq \overline{\text{State}}_1 \times \overline{\text{Tick}}_1$ and $A_2^\# \subseteq \overline{D}_2$. Then we define:

$$S_1^\# \bar{\triangleright}^\# A_2^\# \triangleq \{\bar{p} \bar{\triangleright} \bar{q} | \bar{p} \in S_1^\# \wedge \bar{q} \in A_2^\#\}$$

Now we need to define the semantic linking operator $\bowtie$. More specifically, we must define $S_1 \bowtie A_2$, when $S_1 \subseteq \text{State}_1 \times \text{Tick}_1$ and $A_2 \subseteq D_2$. Remember that A_2 is the separately computed semantics, and S_1 is what was missing. Thus, we must first inject all $(s_1, \text{tick}_1) \in S_1$ into $(\rho_2, \text{tick}_2), \ell_2 \rightsquigarrow_{\text{tick}_2} \rho_2 \in A_2$. Next, since we have gained new information about the external environment, we must collect more that can be gleaned from S_1. Thus, the definition of semantic linking is as follows:

Definition 3.6 (Concrete Linking). Let $S_1 \subseteq \text{State}_1 \times \text{Tick}_1$ and $A_2 \subseteq D_2$. Then:

$$S_1 \bowtie A_2 \triangleq \text{lfp}(\lambda X.\text{Step}(X) \cup (S_1 \triangleright A_2))$$

Definition 3.7 (Abstract Linking). Let $S_1^\# \subseteq \overline{\text{State}}_1 \times \overline{\text{Tick}}_1$ and $A_2^\# \subseteq \overline{D}_2$. Then:

$$S_1^\# \bowtie^\# A_2^\# \triangleq \text{lfp}(\lambda X^\#.\text{Step}^\#(X^\#) \cup (S_1^\# \bar{\triangleright}^\# A_2^\#))$$

3.2 Proof Sketches

Lemma 3.1 (Concrete Advance, Operational). For all $(s_1, \mathrm{tick}_1) \in \mathrm{State}_1 \times \mathrm{Tick}_1$, $\ell_2 \in \mathrm{Left}_2$, $\mathrm{tick}_2 \in \mathrm{Tick}_2$, $\rho_2 \in \mathrm{Right}_2$, If we let $(\ell_+, \mathrm{tick}_+) = (s_1, \mathrm{tick}_1) \rhd (\ell_2, \mathrm{tick}_2)$ and $(\rho_+, \mathrm{tick}_+) = (s_1, \mathrm{tick}_1) \rhd (\rho_2, \mathrm{tick}_2)$, we have:

$$\ell_2 \rightsquigarrow_{\mathrm{tick}_2} \rho_2 \Rightarrow \ell_+ \rightsquigarrow_{\mathrm{tick}_+} \rho_+$$

Thus $S_1 \rhd [\![e]\!] S_2 \subseteq [\![e]\!](S_1 \rhd S_2)$.

Sketch. By induction on $\rightsquigarrow_{\mathrm{tick}_2}$. We use the fact that $m[t \mapsto v]\langle C \rangle = m\langle C \rangle[t \mapsto v\langle C \rangle]$. □

Lemma 3.2 (Concrete Advance). Let $S_1 \subseteq \mathrm{State}_1 \times \mathrm{Tick}_1$ and $S_2 \subseteq \mathrm{State}_2 \times \mathrm{Tick}_2$. Then:

$$[\![e]\!](S_1 \rhd S_2) = S_1 \bowtie [\![e]\!] S_2$$

Sketch. Straightforward from the previous lemma. □

Lemma 3.3 (Abstract Advance, Operational). For all $(\dot{s}_1, \overline{\mathrm{tick}}_1) \in \overline{\mathrm{State}}_1 \times \overline{\mathrm{Tick}}_1$, $\dot{\ell}_2 \in \overline{\mathrm{Left}}_2$, $\overline{\mathrm{tick}}_2 \in \overline{\mathrm{Tick}}_2$, $\bar{\rho}_2 \in \overline{\mathrm{Right}}_2$, If we let $(\dot{\ell}_+, \overline{\mathrm{tick}}_+) = (\dot{s}_1, \overline{\mathrm{tick}}_1) \bar{\rhd} (\dot{\ell}_2, \overline{\mathrm{tick}}_2)$ and $(\bar{\rho}_+, \overline{\mathrm{tick}}_+) = (\dot{s}_1, \overline{\mathrm{tick}}_1) \bar{\rhd} (\bar{\rho}_2, \overline{\mathrm{tick}}_2)$, we have:

$$\dot{\ell}_2 \overline{\rightsquigarrow}_{\mathrm{tick}_2} \bar{\rho}_2 \Rightarrow \dot{\ell}_+ \overline{\rightsquigarrow}_{\mathrm{tick}_+} \bar{\rho}_+$$

Thus $S_1^\# \bar{\rhd}^\# [\![e]\!]^\# S_2^\# \subseteq [\![e]\!]^\# (S_1^\# \bar{\rhd}^\# S_2^\#)$.

Sketch. By induction on $\overline{\rightsquigarrow}_{\mathrm{tick}_2}$. We use the fact that $\bar{m}[\dot{t} \overline{\mapsto} \bar{v}]\langle \bar{C} \rangle = \bar{m}\langle \bar{C} \rangle[\dot{t} \overline{\mapsto} \bar{v}\langle \bar{C} \rangle]$. □

Lemma 3.4 (Abstract Advance). Let $S_1^\# \subseteq \overline{\mathrm{State}}_1 \times \overline{\mathrm{Tick}}_1$ and $S_2^\# \subseteq \overline{\mathrm{State}}_2 \times \overline{\mathrm{Tick}}_2$. Then:

$$[\![e]\!]^\# (S_1^\# \bar{\rhd}^\# S_2^\#) = S_1^\# \bar{\bowtie}^\# [\![e]\!]^\# S_2^\#$$

Sketch. Straightforward from the previous lemma. □

4 Results on Equivalence

In this section, we define what it means for semantics that use different timestamps to be *equivalent*. Our framework hinges heavily on the definition of equivalence, since when we link semantics that use two different timestamps, we end up with a semantics that uses a totally different $\mathbb{T}$ and tick. Even in the case without linking, we need to justify why no matter our choice of $(\mathbb{T}, \leq, \overline{\mathbb{T}}, \bar{\alpha})$, the analysis overapproximates a *compatible* notion of execution.

In this section, we assume a pair of semantics, each parametrized with $(\mathbb{T}, \leq, \overline{\mathbb{T}}, \bar{\alpha})$ and $(\mathbb{T}', \leq', \overline{\mathbb{T}}', \bar{\alpha}')$.

4.1 Definitions

We first define what it means for two states $s \in \mathrm{State}$ and $s' \in \mathrm{State}'$ to be equivalent. Note that $s = (C, m, t)$ and $s' = (C', m', t')$ for some contexts C, C', some memories m, m', and some times t, t'. The choice of t and t' is "not special" in the sense that as long as they bound the context and memory, tick will continue producing fresh addresses. Thus, the notion of equivalence is defined by how the "information extractable" from the C and m components are "same".

Note that the information that is extractable are only accessed through a sequence of names x and M. Thus, one may imagine access "paths" with names on the edges and information sources(C and t) on the nodes. Then the definition of equivalence may be given as equivalence on all access paths with same labels on edges. Taking this a step further, one may imagine an "access graph" that collects all reachable C and t as the nodes of the graph with directed edges connecting the nodes corresponding to accesses by names or the expression part of closures in memory.

$$\begin{array}{rcl@{\qquad}rcl}
\mathsf{Step}_m(G) & \triangleq & \{C \xrightarrow{x} t, t | C \in G \wedge t = C(x)\} & \overline{\mathsf{Step}}_{\bar{m}}(\bar{G}) & \triangleq & \{\bar{C} \xrightarrow{x} \dot{t}, \dot{t} | \bar{C} \in \bar{G} \wedge \dot{t} = \mathsf{addr}(\bar{C}, x)\} \\
& \cup & \{C \xrightarrow{M} C', C' | C \in G \wedge C' = C(M)\} & & \cup & \{\bar{C} \xrightarrow{M} \bar{C}', \bar{C}' | \bar{C} \in \bar{G} \wedge \bar{C}' = \mathsf{ctx}(\bar{C}, M)\} \\
& \cup & \{t \xrightarrow{e} C, C | t \in G \wedge \langle e, C \rangle = m(t)\} & & \cup & \{\dot{t} \xrightarrow{e} \bar{C}, \bar{C} | \dot{t} \in \bar{G} \wedge \langle e, \bar{C} \rangle \in \bar{m}(\dot{t})\} \\
\underline{C, m} & \triangleq & \mathsf{lfp}(\lambda X. \mathsf{Step}_m(X) \cup \{C\}) & \underline{\bar{C}, \bar{m}} & \triangleq & \mathsf{lfp}(\lambda \overline{X}. \overline{\mathsf{Step}}_{\bar{m}}(\overline{X}) \cup \{\bar{C}\})
\end{array}$$

The graphs $\underline{C, m}$ and $\underline{\bar{C}, \bar{m}}$ are *rooted labelled directed graph*s, with the initial context as the root. Then the definition of equivalence between states simply means that the access graphs are isomorphic.

Definition 4.1 (Equivalent Concrete States). Let $s = (C, m, t) \in \text{State}$ and $s' = (C', m', t') \in \text{State}'$. We say s is *equivalent* to s' and write $s \cong s'$ when there exists a $\varphi : \text{Ctx} \uplus \mathbb{T} \to \text{Ctx}' \uplus \mathbb{T}'$ and $\varphi^{-1} : \text{Ctx}' \uplus \mathbb{T}' \to \text{Ctx} \uplus \mathbb{T}$ such that:

1. $\varphi(C) = C'$ and $\varphi^{-1}(C') = C$
2. $\forall \text{node}_1, \text{node}_2 : \text{node}_1 \xrightarrow{\text{lbl}} \text{node}_2 \in \underline{C, m} \Rightarrow \varphi(\text{node}_1) \xrightarrow{\text{lbl}} \varphi(\text{node}_2) \in \underline{C', m'}$
3. $\forall \text{node}'_1, \text{node}'_2 : \text{node}'_1 \xrightarrow{\text{lbl}} \text{node}'_2 \in \underline{C', m'} \Rightarrow \varphi^{-1}(\text{node}'_1) \xrightarrow{\text{lbl}} \varphi^{-1}(\text{node}'_2) \in \underline{C, m}$
4. $\forall \text{node}, \text{node}' : \text{node} \in \underline{C, m} \Rightarrow \varphi^{-1}(\varphi(\text{node})) = \text{node}$ and $\text{node}' \in \underline{C', m'} \Rightarrow \varphi(\varphi^{-1}(\text{node}')) = \text{node}'$

Definition 4.2 (Equivalent Abstract States). Let $\bar{s} = (\overline{C}, \overline{m}, \bar{t}) \in \overline{\text{State}}$ and $\bar{s'} = (\overline{C}', \overline{m}', \bar{t'}) \in \overline{\text{State}}'$. We say $\bar{s}$ is *equivalent* to $\bar{s'}$ and write $\bar{s} \bar{\cong} \bar{s'}$ when there exists a $\bar{\varphi} : \overline{\text{Ctx}} \uplus \overline{\mathbb{T}} \to \overline{\text{Ctx}}' \uplus \overline{\mathbb{T}}'$ and $\bar{\varphi}^{-1} : \overline{\text{Ctx}}' \uplus \overline{\mathbb{T}}' \to \overline{\text{Ctx}} \uplus \overline{\mathbb{T}}$ such that:

1. $\bar{\varphi}(\overline{C}) = \overline{C}'$ and $\bar{\varphi}^{-1}(\overline{C}') = \overline{C}$
2. $\forall \text{node}_1, \text{node}_2 : \text{node}_1 \xrightarrow{\text{lbl}} \text{node}_2 \in \underline{\overline{C}, \overline{m}} \Rightarrow \bar{\varphi}(\text{node}_1) \xrightarrow{\text{lbl}} \bar{\varphi}(\text{node}_2) \in \underline{\overline{C}', \overline{m}'}$
3. $\forall \text{node}'_1.\text{node}'_2 : \text{node}'_1 \xrightarrow{\text{lbl}} \text{node}'_2 \in \underline{\overline{C}', \overline{m}'} \Rightarrow \bar{\varphi}^{-1}(\text{node}'_1) \xrightarrow{\text{lbl}} \bar{\varphi}^{-1}(\text{node}'_2) \in \underline{\overline{C}, \overline{m}}$
4. $\forall \text{node}, \text{node}' : \text{node} \in \underline{\overline{C}, \overline{m}} \Rightarrow \bar{\varphi}^{-1}(\bar{\varphi}(\text{node})) = \text{node}$ and $\text{node}' \in \underline{\overline{C}', \overline{m}'} \Rightarrow \bar{\varphi}(\bar{\varphi}^{-1}(\text{node}')) = \text{node}'$

We also define equivalence between results $r = (V, m.t) \in \text{Result}$ and $r' = (V', m', t') \in \text{Result}'$ by the conjunction of the equality between the expression part of V and the isomorphism between access graphs. Also, equivalence between $\ell = (e, s) \in \text{Left}$ and $\ell' = (e', s') \in \text{Left}'$ can be similarly defined as $e = e' \wedge s \cong s'$. Thus equivalence between $\rho \in \text{Right}$ and $\rho' \in \text{Right}'$ is defined, as either $\rho \in \text{Left}$ or $\rho \in \text{Result}$.

Then the equivalence between elements of $\wp(D)$ and $\wp(D')$, the CPOs, can be defined as:

Definition 4.3 (Equivalence between Elements of $\wp(D)$ and $\wp(D')$). Let $A \subseteq D$ and $A' \subseteq D'$. We say that A and A' are equivalent and write $A \cong A'$ iff:

1. $\forall \ell, \text{tick}, \rho : \ell \rightsquigarrow_{\text{tick}} \rho \in A \Rightarrow \exists \ell', \text{tick}', \rho' : \ell' \rightsquigarrow_{\text{tick}'} \rho' \in A' \wedge \ell \cong \ell' \wedge \rho \cong \rho'$
2. $\forall \ell', \text{tick}', \rho' : \ell' \rightsquigarrow_{\text{tick}'} \rho' \in A' \Rightarrow \exists \ell, \text{tick}, \rho : \ell \rightsquigarrow_{\text{tick}} \rho \in A \wedge \ell \cong \ell' \wedge \rho \cong \rho'$
3. $\forall \rho, \text{tick} : (\rho, \text{tick}) \in A \Rightarrow \exists \rho', \text{tick}' : (\rho', \text{tick}') \in A' \wedge \rho \cong \rho'$
4. $\forall \rho', \text{tick}' : (\rho', \text{tick}') \in A' \Rightarrow \exists \rho, \text{tick} : (\rho, \text{tick}) \in A \wedge \rho \cong \rho'$

Likewise, we can define equivalence between elements of $\wp(\overline{D})$ and $\wp(\overline{D}')$.

Definition 4.4 (Equivalence between Elements of $\wp(\overline{D})$ and $\wp(\overline{D}')$). Let $A^\# \subseteq \overline{D}$ and $A'^\# \subseteq \overline{D}'$. We say that $A^\#$ and $A'^\#$ are equivalent and write $A^\# \bar{\cong}^\# A'^\#$ iff:

1. $\forall \bar{\ell}, \overline{\text{tick}}, \bar{\rho} : \bar{\ell} \overline{\rightsquigarrow}_{\overline{\text{tick}}} \bar{\rho} \in A^\# \Rightarrow \exists \bar{\ell}', \overline{\text{tick}}', \bar{\rho}' : \bar{\ell}' \overline{\rightsquigarrow}_{\overline{\text{tick}}'} \bar{\rho}' \in A'^\# \wedge \bar{\ell} \bar{\cong} \bar{\ell}' \wedge \bar{\rho} \bar{\cong} \bar{\rho}'$
2. $\forall \bar{\ell}', \overline{\text{tick}}', \bar{\rho}' : \bar{\ell}' \overline{\rightsquigarrow}_{\overline{\text{tick}}'} \bar{\rho}' \in A'^\# \Rightarrow \exists \bar{\ell}, \overline{\text{tick}}, \bar{\rho} : \bar{\ell} \overline{\rightsquigarrow}_{\overline{\text{tick}}} \bar{\rho} \in A^\# \wedge \bar{\ell} \bar{\cong} \bar{\ell}' \wedge \bar{\rho} \bar{\cong} \bar{\rho}'$
3. $\forall \bar{\rho}, \overline{\text{tick}} : (\bar{\rho}, \overline{\text{tick}}) \in A^\# \Rightarrow \exists \bar{\rho}', \overline{\text{tick}}' : (\bar{\rho}', \overline{\text{tick}}') \in A' \wedge \bar{\rho} \bar{\cong} \bar{\rho}'$
4. $\forall \bar{\rho}', \overline{\text{tick}}' : (\bar{\rho}', \overline{\text{tick}}') \in A'^\# \Rightarrow \exists \bar{\rho}, \overline{\text{tick}} : (\bar{\rho}, \overline{\text{tick}}) \in A \wedge \bar{\rho} \bar{\cong} \bar{\rho}'$

4.2 Proof Sketches

4.2.1 Evaluation Preserves Equivalence

To prove if we actually did define equivalence sensibly, we must show that the operational semantics preserves equivalence. That is, we need to prove that starting from equivalent configurations, we end up in equivalent configurations.

Claim 4.1 (Evaluation Preserves Equivalence). For all $\ell \in \text{Left}$, $\text{tick} \in \text{Tick}$, $\rho \in \text{Right}$, $\ell' \in \text{Left}'$, $\text{tick}' \in \text{Tick}'$,

$$\ell \rightsquigarrow_{\text{tick}} \rho \wedge \ell \cong \ell' \Rightarrow \exists \rho' : \ell' \rightsquigarrow_{\text{tick}'} \rho' \wedge \rho \cong \rho'$$

Thus, if $S \subseteq \text{State} \times \text{Tick}$ and $S' \subseteq \text{State}' \times \text{Tick}'$ are equivalent, $[\![e]\!]S \cong [\![e]\!]S'$.

Sketch. To perform induction on $\rightsquigarrow_{\mathsf{tick}}$, we need to strengthen the claim. For convenience, we write $(\rho, \varphi) \cong (\rho', \varphi^{-1})$ to emphasize that the graph isomorphism is given by φ.

Then we can strengthen the claim to a claim about graph isomorphisms.

$$\begin{array}{l}\forall \ell, \mathsf{tick}, \rho, \ell', \mathsf{tick}', \varphi, \varphi^{-1} : \ell \rightsquigarrow_{\mathsf{tick}} \rho \wedge (\ell, \varphi) \cong (\ell', \varphi^{-1}) \Rightarrow \\ \exists \rho', \phi, \phi^{-1} : \ell' \rightsquigarrow_{\mathsf{tick}'} \rho' \wedge (\rho, \phi) \cong (\rho', \phi^{-1}) \wedge \varphi(n)|_{n<t} = \phi(n)|_{n<t} \\ \text{when } t \text{ is the time component of } \ell\end{array}$$

The important part here is that ϕ is an *extension* of φ in the sense that it agrees with φ with nodes in $\underline{\ell}$. Thus the induction hypothesis will push through. □

4.2.2 Concretization Preserves Equivalence

For the definition of equivalence to be compatible with analysis, we want to show that if the abstract initial states are equivalent, so are the concretization of those states. If this is true, we can obtain an overapproximation of an equivalent semantics by $[\![e]\!]\gamma(S^\#) \cong [\![e]\!]\gamma'({S'}^\#) \subseteq \gamma'([\![e]\!]^\# {S'}^\#)$ from $S^\# \cong^\# {S'}^\#$. The first $\cong$ is from the fact that evaluation preserves equivalence and concretization preserves equivalence. The second $\subseteq$ is from the fact that $\mathsf{Step}^\#$ is a sound approximation of Step. The claim we want to prove is: If $S^\# \cong^\# {S'}^\#$, then $\gamma(S^\#) \cong \gamma'({S'}^\#)$.

Claim 4.2 (Concretization Preserves Equivalence). For all $S^\# \subseteq \overline{\mathsf{State}} \times \overline{\mathsf{Tick}}$ and ${S'}^\# \subseteq \overline{\mathsf{State}}' \times \overline{\mathsf{Tick}}'$, $S^\# \cong^\# {S'}^\#$ implies $\gamma(S^\#) \cong \gamma'({S'}^\#)$.

Sketch. We want to prove:

$$\forall s \in \mathsf{State}, \bar{s}' \in \overline{\mathsf{State}}' : \bar{\alpha}(s) \bar{\cong} \bar{s}' \Rightarrow \exists s' \in \overline{\mathsf{State}}' : s \cong s' \wedge \bar{\alpha}'(s') = \bar{s}'$$

If this is true, $\forall s \in \gamma(s^\#) : \exists s' \in \gamma({s'}^\#) : s \cong s'$.

Proving the same statement in the opposite side leads to $\forall s' \in \gamma({s'}^\#) : \exists s \in \gamma(s^\#) : s \cong s'$, so that $\gamma(S^\#) \cong \gamma'({S'}^\#)$. The proof of the above statement involves constructing such a s' by traversal over reachable subparts n of s, (1) translating n to a reachable part $\bar{n}'$ of $\bar{s}'$ by $\bar{\varphi} \circ \bar{\alpha}$(when $\bar{\varphi}$ is the isomorphism between $\bar{\alpha}(s)$ and $\bar{s}'$) and (2) lifting $\bar{n}'$ to a reachable part n' of s' while tabulating the graph isomorphism φ between s and s'. Finally, (3) the unreachable parts of s' are filled in with dummy values that translate to the unfilled parts of $\bar{s}'$. □