(Sketches of) Proofs for Modular Analysis

Joonhyup Lee

October 6, 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.

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

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

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

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

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

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

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

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 Left $\triangleq$ Expr $\times$ Ctx with an element ρ of Right $\triangleq$ Left $\uplus$ 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, $D \triangleq \mathscr{P}(\Sigma)$ when $\Sigma \triangleq \rightsquigarrow \uplus$Right and $\mathscr{P}(S)$ 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 $\rightsquigarrow$. When we define $\ell\langle C_1 \rangle$ for $C_1 \in \mathrm{Ctx}, \ell = (e, C_2) \in \mathrm{Left}$ as $(e, C_2\langle C_1 \rangle)$, we have:

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 ∞ that does the rest of the computation:

Definition 1.3 (Injection). 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:

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

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 $S_1, S_2 \subseteq \mathrm{Ctx}$,

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

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:

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

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.

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

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, $\text{tick}(t) \notin \text{supp}(C, m)$ 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 our criteria can be guaranteed by first enforcing $C \leq t$ and $m \leq t$, when

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

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

. Then the criteria that $\text{tick}(t)$ must be fresh 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:

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

as the set of *valid* states that enable tick to behave nicely. It is almost trivial that the set Left × Right, 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 $D \triangleq \mathcal{P}(\Sigma)$ 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:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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:

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

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 $\overline{\alpha}$ as:

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

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

and define $\overline{\alpha}(r)$ 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 $\overline{\alpha}(\ell) \overline{\rightsquigarrow} \overline{\alpha}(\rho)$.

Sketch. Induction on $\rightsquigarrow$. □

Then if we define $D^\# \triangleq \mathscr{P}(\overline{\Sigma})$, 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:

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

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

Then it is straightforward to see that:

Claim 2.3 (Galois Connection). $\mathscr{P}(\Sigma) = D \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} D^\# = \mathscr{P}(\overline{\Sigma})$. 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

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

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

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

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

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

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

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

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

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

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

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

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

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

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:

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

Then we can prove that:

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

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 $\alpha \circ \gamma \subseteq \mathsf{id}$ 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 $\lfloor \bar{C} \rfloor$ to be the signatures that are obtained by stripping all timestamps from C and $\bar{C}$. Moreover, we may define $\lfloor m \rfloor \triangleq \{\lfloor C \rfloor | \exists t : \langle \_, C \rangle = m(t)\}$ and $\lfloor \overline{m} \rfloor \triangleq \{\lfloor \bar{C} \rfloor | \exists \bar{t} : \langle \_, \bar{C} \rangle \in \overline{m}(\bar{t})\}$ 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 and m is the memory component of $\rho = (V, m, t)$ or (e, C, m, t). $\lfloor \bar{\rho} \rfloor$ can be analogously defined.

If we can prove that for all e and $\bar{s}$, there exists a finite set $X \subseteq \mathsf{Sig}$ that bounds $\bigcup_{(e,\bar{s}) \overline{\rightsquigarrow}^* \bar{\rho}} \lfloor \bar{\rho} \rfloor \subseteq X$, we can show that all reachable $\bar{\rho}$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 $f : \overline{\mathsf{Right}} \to \mathcal{P}(\mathsf{Sig})$. We need two lemmas, the first being $\forall \bar{\rho} : \lfloor \bar{\rho} \rfloor \subseteq f(\bar{\rho})$ and the second being $\forall \bar{\ell}, \bar{\rho} : \bar{\ell} \overline{\rightsquigarrow} \bar{\rho} \Rightarrow f(\bar{\rho}) \subseteq f(\bar{\ell})$.

If we have the lemmas, then for all $\bar{\rho}$ such that $(e, \bar{s}) \overline{\rightsquigarrow}^* \bar{\rho}$, $\lfloor \bar{\rho} \rfloor \subseteq f(\bar{\rho}) \subseteq f(e, \bar{s})$, thus all signatures in $[\![e]\!]^\#S^\#$ can be bound by $X \triangleq \bigcup_{s \in S^\#} f(e, \bar{s})$, 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: Linking in the Semantics with Memory

Now we need to define an injection operation that fills in the blanks of a $r = (V, m, t) \in \mathsf{Result}$ with a $s = (C', m', t') \in \mathsf{State}$. Recall the definition for injection in the semantics without memory. $V\langle C \rangle$ enables access to values that were previously not available in V by filling in the bottom of the stack with C. Thus, we must mimic this by filling in all contexts in r with the context part of s. Also, to retain all information stored in the memory, the memory part of r must be merged with the memory of s.

It is at this point that a problem occurs. When merging the two memories m and m', we may encounter overlapping addresses. Thus, we must require that all reachable addresses from (C, m) does not overlap with reachable addresses in (C', m'). Then again, this requirement may be lifted if we allow linking of semantics that use *different* time domains as addresses. After all, we can only *read* values from C in $V\langle C \rangle$; why not preserve addresses that were used in s before injection and never allow writing to those addresses? Thus, in this section we first define $r_2\langle s_1 \rangle$, when s_1 uses $\mathbb{T}_1$ as addresses and r_2 uses $\mathbb{T}_2$ as addresses. Then $r_2\langle s_1 \rangle$ must live in a version of Result that uses $\mathbb{T}_1 + \mathbb{T}_2$ as addresses. From now on, variables with subscripts 1 or 2 are to be understood to be using $\mathbb{T}_i (i = 1, 2)$ as addresses, and variables with the subscript + are to be understood to be the linked version.

Defining linking between different time domains demand that tick, $\overline{\mathbb{T}}$, $\bar{\alpha}$, and $\overline{\mathsf{tick}}$ also be linked. Concretely, we demand that the linked tick_+ preserves the condition that $\bar{\alpha}_+ \circ \mathsf{tick}_+ = \overline{\mathsf{tick}}_+ \circ \bar{\alpha}_+$. Also, we need to link tick well so that for all valid transitions $\ell_2 \rightsquigarrow \rho_2$ under tick_2, $\ell_2\langle s_1 \rangle \rightsquigarrow \rho_2\langle s_1 \rangle$ is also a valid transition under tick_+. This is to ensure that injection into the collecting semantics is well-defined. For the rest of this section, we define linking for all semantic domains and prove that the requirements laid out in the skeleton for static analysis hold.

3.1 tick_+, $\bar{\alpha}_+$, $\overline{\mathsf{tick}}$

We must first define tick_+, $\bar{\alpha}_+$, and $\overline{\mathsf{tick}}_+$ that satisfies the condition that $t_+ \leq \mathsf{tick}_+(t_+)$ and $\bar{\alpha}_+ \circ \mathsf{tick}_+ = \overline{\mathsf{tick}}_+ \circ \bar{\alpha}_+$. But wait, what must be the order on $\mathbb{T}_1 + \mathbb{T}_2$? We define $\leq_+$ to be the *lexicographic* order on $\mathbb{T}_1 + \mathbb{T}_2$, when an element t_1

of $\mathbb{T}_1$ is lifted to $(0, t_1)$ and an element t_2 of $\mathbb{T}_2$ is lifted to $(1, t_2)$. Then if we define:

$$\mathsf{tick}_+(t) \triangleq \begin{cases} \mathsf{tick}_1(t) & t \in \mathbb{T}_1 \\ \mathsf{tick}_2(t) & t \in \mathbb{T}_2 \end{cases} \qquad \bar{\alpha}_+(t) \triangleq \begin{cases} \bar{\alpha}_1(t) & t \in \mathbb{T}_1 \\ \bar{\alpha}_2(t) & t \in \mathbb{T}_2 \end{cases} \qquad \overline{\mathsf{tick}}_+(\dot{t}) \triangleq \begin{cases} \overline{\mathsf{tick}}_1(\dot{t}) & \dot{t} \in \overline{\mathbb{T}}_1 \\ \overline{\mathsf{tick}}_2(\dot{t}) & \dot{t} \in \overline{\mathbb{T}}_2 \end{cases}$$

it is easy to check that all requirements are satisfied.

3.2 Injection

Now we define injection between $s_1 \in \mathsf{State}_1$ and $r_2 \in \mathsf{Result}_2$.

$$V_2\langle C_1\rangle \triangleq \begin{cases} C_1 & V_2 = [] \\ (x,t) :: C\langle C_1\rangle & V_2 = (x,t) :: C \\ (M, C\langle C_1\rangle) :: C'\langle C_1\rangle & V_2 = (M,C) :: C' \\ \langle \lambda x.e, C_2\langle C_1\rangle\rangle & V_2 = \langle \lambda x.e, C_2\rangle \end{cases} \qquad m_2\langle C_1\rangle \triangleq \bigcup_{t \in \mathsf{dom}(m_2)} [t \mapsto m_2(t)\langle C_1\rangle]$$

$$r_2\langle s_1\rangle \triangleq (V_2\langle C_1\rangle, m_1 \cup m_2\langle C_1\rangle, t_2)$$

Figure 8: Definition of the injection operator. In the definition of $r_2\langle s_1\rangle$, $s_1 = (C_1, m_1, t_1)$ and $r_2 = (V_2, m_2, t_2)$.

As is expected, injecting s_1 into r_2 involves injecting C_1 in every context in r_2 and merging the memories. This definition is exactly what we were searching for, since it respects all requirements laid out in the introduction to this section. First, the time-boundedness of $r_2\langle s_1\rangle$ is guaranteed with respect to the ordering $\leq_+$, thus $r_2\langle s_1\rangle$ is safely in Result_+. Also, if we define $\ell_2\langle s_1\rangle$ when $\ell_2 = (e, s_2)$ by $(e, s_2\langle s_1\rangle)$, we can show that injection preserves valid transitions. That is,

Claim 3.1 (Injection Preserves Evaluation). For all $s_1 \in \mathsf{State}_1$, $\ell_2 \in \mathsf{Left}_2$ and $\rho_2 \in \mathsf{Right}_2$, if $\ell_2 \rightsquigarrow \rho_2$ under tick_2, then $\ell_2\langle s_1\rangle \rightsquigarrow \rho_2\langle s_1\rangle$ under tick_+.

Sketch. Induction on $\rightsquigarrow$ under tick_2. We use the equality $m[t \mapsto v]\langle C\rangle = m\langle C\rangle[t \mapsto v\langle C\rangle]$ and other such equalities that allow commutativity of injection and semantic operations. □

Thus we can define $\triangleright$ and $\between$ that satisfies the desired property.

Definition 3.1 (Injection). For $S_1 \subseteq \mathsf{State}_1$ and $A_2 \in D_2$, define:

$$S_1 \triangleright A_2 \triangleq \{\rho_2\langle s_1\rangle | s_1 \in S_1 \wedge \rho_2 \in A_2\} \cup \{\ell_2\langle s_1\rangle \rightsquigarrow \rho_2\langle s_1\rangle | s_1 \in S_1 \wedge \ell_2 \rightsquigarrow \rho_2 \in A_2\}$$

Definition 3.2 (Semantic Linking). For $S_1 \subseteq \mathsf{State}_1$ and $A_2 \in D_2$, define:

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

Claim 3.2 (Advance). For all $e \in \mathsf{Expr}$ and $S_1 \subseteq \mathsf{State}_1$, $S_2 \subseteq \mathsf{State}_2$,

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

3.3 Checking Soundness for Analysis

We can define injection and linking in the abstract semantics in the same way as the concrete semantics. Only the definition of $\bar{m}_2\langle \bar{C}_1\rangle$ in Figure 8 has to be adapted to account for the fact that $\bar{m}_2(\dot{t})$ is now a *set* of closures and thus injection of $\bar{C}_1$ must be mapped over all elements. Then we can show that:

Claim 3.3 (Injection Preserves Abstract Evaluation). For all $\dot{s}_1 \in \overline{\mathsf{State}}_1$, $\dot{\ell}_2 \in \overline{\mathsf{Left}}_2$ and $\bar{\rho}_2 \in \overline{\mathsf{Right}}_2$, if $\dot{\ell}_2 \overline{\rightsquigarrow} \bar{\rho}_2$ under $\overline{\mathsf{tick}}_2$, then $\dot{\ell}_2\langle \dot{s}_1\rangle \overline{\rightsquigarrow} \bar{\rho}_2\langle \dot{s}_1\rangle$ under $\overline{\mathsf{tick}}_+$.

Sketch. Induction on $\overline{\rightsquigarrow}$ under $\overline{\mathsf{tick}}_2$. □

and thus we can define:

Definition 3.3 (Abstract Injection). For $S_1^\# \subseteq \overline{\mathsf{State}}_1$ and $\bar{A}_2 \in D_2^\#$, define:

$$S_1^\# \bar{\triangleright}^\# A_2^\# \triangleq \{\bar{\rho}_2\langle \dot{s}_1\rangle | \dot{s}_1 \in S_1^\# \wedge \bar{\rho}_2 \in A_2^\#\} \cup \{\dot{\ell}_2\langle \dot{s}_1\rangle \overline{\rightsquigarrow} \bar{\rho}_2\langle \dot{s}_1\rangle | \dot{s}_1 \in S_1^\# \wedge \dot{\ell}_2 \overline{\rightsquigarrow} \bar{\rho}_2 \in A_2^\#\}$$

Definition 3.4 (Abstract Linking). For $S_1^\# \subseteq \overline{\mathsf{State}}_1$ and $A_2^\# \in D_2^\#$, define:

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

so that the *best possible result* is achieved:

Claim 3.4 (Abstract Advance). For all $e \in \text{Expr}$ and $S_1^\# \subseteq \overline{\text{State}}_1$, $S_2^\# \subseteq \overline{\text{State}}_2$,

$$[\![e]\!]^\#(S_1^\# \triangleright^\# S_2^\#) = S_1^\# \leadsto^\# [\![e]\!]^\# S_2^\#$$

Since we have that

$$\alpha_+(S_1 \triangleright A_2) = \alpha_1(S_1) \triangleright^\# \alpha_2(A_2)$$

due to $\alpha_+(r_2\langle s_1 \rangle) = \alpha(r_2)\langle \alpha_1(s_1) \rangle$, the above claim directly leads to overapproximation by:

$$\begin{aligned}
S_1 \leadsto [\![e]\!] S_2 &= [\![e]\!](S_1 \triangleright S_2) && (\because \text{Advance}) \\
&\subseteq \gamma_+([\![e]\!]^\# \alpha_+(S_1 \triangleright S_2)) && (\because \text{Galois connection}) \\
&= \gamma_+([\![e]\!]^\#(\alpha_1(S_1) \triangleright^\# \alpha_2(S_2))) && (\because \alpha_+(S_1 \triangleright A_2) = \alpha_1(S_1) \triangleright^\# \alpha_2(A_2)) \\
&= \gamma_+(\alpha_1(S_1) \leadsto^\# [\![e]\!]^\# \alpha_2(S_2)) && (\because \text{Abstract advance})
\end{aligned}$$

4 Part IV: A Criteria for Reusing Analysis Results

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 $(\mathbb{T}, \leq, \overline{\mathbb{T}}, \overline{\alpha})$, the analysis overapproximates a *compatible* notion of execution.

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

4.1 Motivation

Assume that we have a cached analysis result that uses the abstract timestamps $\{0, 1\}$. Unfortunately,

4.2 Definitions

We first define what it means for two states $s \in \text{State}$ and $s' \in \text{State}'$ 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. Also, given a $\varphi \in \mathbb{T} \to \mathbb{T}'$, we can define how access paths that use timestamps in $\mathbb{T}$ are translated to access paths in $\mathbb{T}'$. The definitions are given in 9. From now on, we shall write Path for the set of access paths that use timestamps in $\mathbb{T}$, and

$$\begin{array}{rcll}
p & \to & \epsilon & \text{(empty path)} \\
& | & \xrightarrow{x} t\ p & \text{(address access)} \\
& | & \xrightarrow{M} p & \text{(module access)} \\
& | & \xrightarrow{\lambda x.e} p & \text{(value access)}
\end{array}
\qquad
\varphi(p) \triangleq \begin{cases}
\epsilon & p = \epsilon \\
\xrightarrow{x} \varphi(t)\ \varphi(p') & p = \xrightarrow{x} t\ p' \\
\xrightarrow{M} \varphi(p') & p = \xrightarrow{M} p' \\
\xrightarrow{\lambda x.e} \varphi(p') & p = \xrightarrow{\lambda x.e} p'
\end{cases}$$

Figure 9: Definition for access paths and how translation of timestamps are mapped over access paths.

Path' for the set of access paths that use timestamps in $\mathbb{T}'$.

Then given an access path, we can define a predicate $\text{valid} \in (\text{Ctx} \uplus \mathbb{T}) \times \text{Mem} \times \text{Path} \to \text{Prop}$. Given $r \in \text{Ctx} \uplus \mathbb{T}, m \in \text{Mem}$, $p \in \text{Path}$, $\text{valid}(r, m, p)$ is true iff all access edges in the path are valid, when starting from r.

$$\text{valid}(r, m, p) \triangleq \begin{cases}
\text{True} & p = \epsilon \\
t = C(x) \wedge \text{valid}(t, m, p') & r = C \wedge p = \xrightarrow{x} t\ p' \\
\exists C_M\ :\ C_M = C(M) \wedge \text{valid}(C_M, m, p') & r = C \wedge p = \xrightarrow{M} p' \\
\exists C\ :\ \langle \lambda x.e, C \rangle = m(t) \wedge \text{valid}(C, m, p') & r = t \wedge p = \xrightarrow{\lambda x.e} p' \\
\text{False} & \text{otherwise}
\end{cases}$$

Likewise, we can define a predicate $\overline{\text{valid}} \in (\overline{\text{Ctx}} \uplus \overline{\mathbb{T}}) \times \overline{\text{Mem}} \times \overline{\text{Path}} \to \text{Prop}$. Given $\bar{r} \in \overline{\text{Ctx}} \uplus \overline{\mathbb{T}}, \bar{m} \in \overline{\text{Mem}}, \bar{p} \in \overline{\text{Path}}$, $\overline{\text{valid}}(\bar{r}, \bar{m}, \bar{p})$ is true iff all access edges in the path are valid, when starting from $\bar{r}$.

$$\overline{\text{valid}}(\bar{r}, \bar{m}, \bar{p}) \triangleq \begin{cases} \text{True} & \bar{p} = \epsilon \\ \dot{t} = \overline{C}(x) \wedge \overline{\text{valid}}(\dot{t}, \bar{m}, \overline{p'}) & \bar{r} = \overline{C} \wedge \bar{p} = \xrightarrow{x} \dot{t}\ \overline{p'} \\ \exists \overline{C_M} : \overline{C_M} = \overline{C}(M) \wedge \overline{\text{valid}}(\overline{C_M}, \bar{m}, \overline{p'}) & \bar{r} = \overline{C} \wedge \bar{p} = \xrightarrow{M} \overline{p'} \\ \exists \overline{C} : \langle \lambda x.e, \overline{C} \rangle \in \bar{m}(\dot{t}) \wedge \overline{\text{valid}}(\overline{C}, \bar{m}, \overline{p'}) & \bar{r} = \dot{t} \wedge \bar{p} = \xrightarrow{\lambda x.e} \overline{p'} \\ \text{False} & \text{otherwise} \end{cases}$$

Now we can give a straightforward definitions of equivalence.

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 $\varphi : \mathbb{T} \to \mathbb{T}'$ and $\varphi^{-1} : \mathbb{T}' \to \mathbb{T}$ such that:

1. For all $p \in \text{Path}$, if $\text{valid}(C, m, p)$ then $\text{valid}(C', m', \varphi(p))$ and $p = \varphi^{-1}(\varphi(p))$.
2. For all $p' \in \text{Path}'$, if $\text{valid}(C', m', p')$ then $\text{valid}(C, m, \varphi^{-1}(p'))$ and $p' = \varphi(\varphi^{-1}(p'))$.

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

1. For all $\bar{p} \in \overline{\text{Path}}$, if $\overline{\text{valid}}(\overline{C}, \overline{m}, \bar{p})$ then $\overline{\text{valid}}(\overline{C'}, \overline{m'}, \bar{\varphi}(p))$ and $\bar{p} = \bar{\varphi}^{-1}(\bar{\varphi}(\bar{p}))$.
2. For all $\overline{p'} \in \overline{\text{Path}'}$, if $\overline{\text{valid}}(\overline{C'}, \overline{m'}, \overline{p'})$ then $\text{valid}(\overline{C}, \overline{m}, \bar{\varphi}^{-1}(p'))$ and $\overline{p'} = \bar{\varphi}(\bar{\varphi}^{-1}(\overline{p'}))$.

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 equivalence between . 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 D and D', the CPOs, can be defined as:

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

1. $\forall \ell, \rho : \ell \rightsquigarrow \rho \in A \Rightarrow \exists \ell', \rho' : \ell' \rightsquigarrow \rho' \in A' \wedge \ell \cong \ell' \wedge \rho \cong \rho'$
2. $\forall \ell', \rho' : \ell' \rightsquigarrow \rho' \in A' \Rightarrow \exists \ell, \rho : \ell \rightsquigarrow \rho \in A \wedge \ell \cong \ell' \wedge \rho \cong \rho'$
3. $\forall \rho : \rho \in A \Rightarrow \exists \rho' : \rho' \in A' \wedge \rho \cong \rho'$
4. $\forall \rho' : \rho' \in A' \Rightarrow \exists \rho : \rho \in A \wedge \rho \cong \rho'$

Likewise, we can define equivalence between elements of $D^\#$ and $D'^\#$.

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

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

4.3 Proof Sketches

4.3.1 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 $\ell \in \text{Left}$, $\rho \in \text{Right}$, $\ell' \in \text{Left}'$,

$$\ell \rightsquigarrow \rho \wedge \ell \cong \ell' \Rightarrow \exists \rho' : \ell' \rightsquigarrow \rho' \wedge \rho \cong \rho'$$

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

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

Then we can strengthen the claim.

$$\begin{aligned}&\forall \ell, \rho, \ell', \varphi, \varphi^{-1} : \ell \rightsquigarrow \rho \wedge (\ell, \varphi) \cong (\ell', \varphi^{-1}) \Rightarrow \\ &\exists \rho', \phi, \phi^{-1} : \ell' \rightsquigarrow \rho' \wedge (\rho, \phi) \cong (\rho', \phi^{-1}) \wedge \varphi|_{\leq t} = \phi|_{\leq t} \\ &\text{when } t \text{ is the time component of } \ell\end{aligned}$$

The important part here is that ϕ is an *extension* of φ. Thus the induction hypothesis will push through. □

4.3.2 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 $\text{Step}^\#$ 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 $S^\# \subseteq \overline{\text{State}}$ and $S'^\# \subseteq \overline{\text{State}}'$, $S^\# \cong^\# S'^\#$ implies $\gamma(S^\#) \cong \gamma'(S'^\#)$.

Sketch. We want to prove:

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

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 $\bar{\varphi}$ gives the equivalence 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 $\bar{s'}$. □