

(Sketches of) Proofs for Modular Analysis

Joonhyup Lee

October 17, 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:

$$\llbracket e \rrbracket (S_1 \triangleright S_2) = S_1 \times \llbracket e \rrbracket S_2$$

when $S_1 \times \llbracket e \rrbracket S_2$ closes the fractionally closed $\llbracket e \rrbracket S_2$ and $\llbracket e \rrbracket (S_1 \triangleright S_2)$ 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	\rightarrow	x	value identifier
			$\lambda x.e$	function
			$e e$	application
			$e \bowtie e$	linked expression
			ε	empty module
			M	module identifier
			$\text{let } x e e$	binding expression
			$\text{let } M e e$	binding module

Figure 1: Abstract syntax of the simple module language.

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

Figure 2: Definition of the semantic domains.

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

$$\begin{array}{c}
\text{[EXPRID]} \frac{v = C(x)}{(x, C) \rightsquigarrow v} \quad \text{[FN]} \frac{}{(\lambda x.e, C) \rightsquigarrow \langle \lambda x.e, C \rangle} \quad \text{[APPL]} \frac{}{(e_1 e_2, C) \rightsquigarrow (e_1, C)} \\
\\
\text{[APPR]} \frac{(e_1, C) \rightsquigarrow \langle \lambda x.e_\lambda, C_\lambda \rangle}{(e_1 e_2, C) \rightsquigarrow (e_2, C)} \quad \text{[APPBODY]} \frac{(e_1, C) \rightsquigarrow \langle \lambda x.e_\lambda, C_\lambda \rangle \quad (e_2, C) \rightsquigarrow v}{(e_1 e_2, C) \rightsquigarrow (e_\lambda, (x, v) :: C_\lambda)} \quad \text{[APP]} \frac{(e_1, C) \rightsquigarrow \langle \lambda x.e_\lambda, C_\lambda \rangle \quad (e_2, C) \rightsquigarrow v \quad (e_\lambda, (x, v) :: C_\lambda) \rightsquigarrow v'}{(e_1 e_2, C) \rightsquigarrow v'} \\
\\
\text{[LINKL]} \frac{}{(e_1 \bowtie e_2, C) \rightsquigarrow (e_1, C)} \quad \text{[LINKR]} \frac{(e_1, C) \rightsquigarrow C'}{(e_1 \bowtie e_2, C) \rightsquigarrow (e_2, C')} \quad \text{[LINK]} \frac{(e_1, C) \rightsquigarrow C' \quad (e_2, C') \rightsquigarrow V}{(e_1 \bowtie e_2, C) \rightsquigarrow V} \\
\\
\text{[EMPTY]} \frac{}{(\varepsilon, C) \rightsquigarrow C} \quad \text{[MODID]} \frac{C' = C(M)}{(M, C) \rightsquigarrow C'} \\
\\
\text{[LETML]} \frac{}{(\text{let } x e_1 e_2, C) \rightsquigarrow (e_1, C)} \quad \text{[LETMR]} \frac{(e_1, C) \rightsquigarrow v}{(\text{let } x e_1 e_2, C) \rightsquigarrow (e_2, (x, v) :: C)} \quad \text{[LETE]} \frac{(e_1, C) \rightsquigarrow v \quad (e_2, (x, v) :: C) \rightsquigarrow C'}{(\text{let } x e_1 e_2, C) \rightsquigarrow C'} \\
\\
\text{[LETML]} \frac{}{(\text{let } M e_1 e_2, C) \rightsquigarrow (e_1, C)} \quad \text{[LETMR]} \frac{(e_1, C) \rightsquigarrow C'}{(\text{let } M e_1 e_2, C) \rightsquigarrow (e_2, (M, C') :: C)} \quad \text{[LETM]} \frac{(e_1, C) \rightsquigarrow C' \quad (e_2, (M, C') :: C) \rightsquigarrow C''}{(\text{let } M e_1 e_2, C) \rightsquigarrow C''}
\end{array}$$

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 operational 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, $D \triangleq \mathcal{P}(\Sigma)$ when $\Sigma \triangleq \rightsquigarrow \uplus \text{Right}$ and $\mathcal{P}(S)$ is the

powerset of S . The semantics of an expression e starting from initial states in $S \subseteq \text{Ctx}$ 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

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

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 $e \in \text{Expr}$ and $S \subseteq \text{Ctx}$, define:

$$\llbracket e \rrbracket S \triangleq \text{lfp}(\lambda X. \text{Step}(X) \cup \{(e, C) \mid C \in S\})$$

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 $V_2 \in \text{ValCtx}$ with a $C_1 \in \text{Ctx}$:

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

This does indeed “fill in the blanks”, since:

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

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

and for each module variable M ,

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

Sketch. Induction on C_2 . □

Moreover, filling in the blanks preserves the evaluation relation \rightsquigarrow . When we define $\ell \langle C_1 \rangle$ for $C_1 \in \text{Ctx}$, $\ell = (e, C_2) \in \text{Left}$ as $(e, C_2 \langle C_1 \rangle)$, we have:

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

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

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 \times that does the rest of the computation:

Definition 1.3 (Injection). For $S \subseteq \text{Ctx}$ and $A \in D$, define:

$$S \triangleright A \triangleq \{\rho \langle C \rangle \mid C \in S \text{ and } \rho \in A\} \cup \{\ell \langle C \rangle \rightsquigarrow \rho \langle C \rangle \mid C \in S \text{ and } \ell \rightsquigarrow \rho \in A\}$$

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

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

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

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

$$\llbracket e \rrbracket (S_1 \triangleright S_2) = S_1 \times \llbracket e \rrbracket S_2$$

Proof. Let A be $\{(e, C) \mid C \in S_1 \triangleright S_2\}$, and let B be $S_1 \triangleright \llbracket e \rrbracket S_2$. Note that $A \subseteq B$ by the definition of $\llbracket e \rrbracket S_2$.

Also, let X_A be $\text{lfp}(\lambda X. \text{Step}(X) \cup A) = \llbracket e \rrbracket (S_1 \triangleright S_2)$ and let X_B be $\text{lfp}(\lambda X. \text{Step}(X) \cup B) = S_1 \bowtie \llbracket e \rrbracket S_2$. By the previous lemma, we have that $B \subseteq X_A$.

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

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

. 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 $\lambda X. \text{Step}(X) \cup A$, since

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

. 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 $X_A = X_B$. \square

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 :

$$\mathcal{P}(\Sigma) = D \xleftrightarrow[\alpha]{\gamma} D^\#$$

and semantic operators $\text{Step}^\#$ and $\triangleright^\#$ that satisfies:

$$\text{Step} \circ \gamma \subseteq \gamma \circ \text{Step}^\# \quad \triangleright \circ (\gamma, \gamma) \subseteq \gamma \circ \triangleright^\#$$

. Then we define $\llbracket e \rrbracket^\#$ and $\bowtie^\#$ as:

$$\llbracket e \rrbracket^\# S^\# \triangleq \text{lfp}(\lambda X^\#. \text{Step}^\#(X^\#) \cup^\# \alpha\{(e, C) \mid C \in \gamma S^\#\}) \quad S^\# \bowtie^\# A^\# \triangleq \text{lfp}(\lambda X^\#. \text{Step}^\#(X^\#) \cup^\# (S^\# \triangleright^\# A^\#))$$

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

$$\llbracket e \rrbracket \circ \gamma \subseteq \gamma \circ \llbracket e \rrbracket^\# \quad \bowtie \circ (\gamma, \gamma) \subseteq \gamma \circ \bowtie^\#$$

. Then we can soundly approximate fractional specifications by:

$$\begin{aligned} S_1 \bowtie \llbracket e \rrbracket S_2 &\subseteq S_1 \bowtie \gamma(\llbracket e \rrbracket^\# \alpha(S_2)) & (\because \llbracket e \rrbracket \subseteq \gamma \circ \llbracket e \rrbracket^\# \circ \alpha \text{ and monotonicity of } \bowtie) \\ &\subseteq \gamma(\alpha(S_1)) \bowtie \gamma(\llbracket e \rrbracket^\# \alpha(S_2)) & (\because \text{id} \subseteq \gamma \circ \alpha \text{ and monotonicity of } \bowtie) \\ &\subseteq \gamma(\alpha(S_1) \bowtie^\# \llbracket e \rrbracket^\# \alpha(S_2)) & (\because \bowtie \circ (\gamma, \gamma) \subseteq \gamma \circ \bowtie^\#) \end{aligned}$$

2 Part II: Instrumented Semantics to Enable Abstraction

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 $S \subseteq \text{Ctx}$ to finitely compute an overapproximation. However, devising such an abstraction is not immediately obvious, since such abstractions must support operations satisfying $\{C(x) \mid C \in \gamma(C^\#)\} \subseteq \gamma(C^\#(x))$, 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 semantics that involve choosing a *time* domain \mathbb{T} to use as addresses, and an *abstract time domain* ${}^\circ\mathbb{T}$ that allow easy abstraction of the semantics via a *single* function ${}^\circ\alpha : \mathbb{T} \rightarrow {}^\circ\mathbb{T}$.

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 $s = (C, m, t) \in \text{State}$ 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 $r = (V, m, t) \in \text{Result}$ 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 $\mathbb{T} \rightarrow \mathbb{T}$, the simplest possible option.

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

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

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

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

. 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:

$$\text{State} \triangleq \{(C, m, t) \mid C \leq t \text{ and } m \leq t\} \quad \text{Result} \triangleq \{(V, m, t) \mid V \leq t \text{ and } m \leq t\}$$

as the set of *valid* states that enable tick to behave nicely. It is almost trivial that the set $\text{Left} \times \text{Right}$, when $\text{Left} \triangleq \text{Expr} \times \text{State}$ and $\text{Right} \triangleq \text{Left} \uplus \text{Result}$, 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

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

and

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

$$\llbracket e \rrbracket S \triangleq \text{lfp}(\lambda X. \text{Step}(X) \cup \{(e, s) \mid s \in S\})$$

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

$$\begin{array}{c}
\text{[EXPRID]} \frac{t_x = C(x) \quad v = m(t_x)}{(x, C, m, t) \rightsquigarrow (v, m, t)} \quad \text{[FN]} \frac{}{(\lambda x.e, C, m, t) \rightsquigarrow (\langle \lambda x.e, C \rangle, m, t)} \\
\\
\text{[APPL]} \frac{}{(e_1 \ e_2, C, m, t) \rightsquigarrow (e_1, C, m, t)} \quad \text{[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)} \\
\\
\text{[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, \text{tick}(t_a)) :: C_\lambda, m_a[\text{tick}(t_a) \mapsto v], \text{tick}(t_a))} \\
\\
\text{[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, \text{tick}(t_a)) :: C_\lambda, m_a[\text{tick}(t_a) \mapsto v], \text{tick}(t_a)) \rightsquigarrow (v', m', t') \end{array}}{(e_1 \ e_2, C, m, t) \rightsquigarrow (v', m', t')} \\
\\
\text{[LINKL]} \frac{}{(e_1 \rtimes e_2, C, m, t) \rightsquigarrow (e_1, C, m, t)} \quad \text{[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')} \\
\\
\text{[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'')} \quad \text{[EMPTY]} \frac{}{(\varepsilon, C, m, t) \rightsquigarrow (C, m, t)} \quad \text{[MODID]} \frac{C' = C(M)}{(M, C, m, t) \rightsquigarrow (C', m, t)} \\
\\
\text{[LETML]} \frac{}{(\text{let } x \ e_1 \ e_2, C, m, t) \rightsquigarrow (e_1, C, m, t)} \quad \text{[LETMR]} \frac{(e_1, C, m, t) \rightsquigarrow (C', m', t')}{(\text{let } M \ e_1 \ e_2, C, m, t) \rightsquigarrow (e_2, (M, C') :: C, m', t')} \\
\\
\text{[LETTER]} \frac{(e_1, C, m, t) \rightsquigarrow (v, m', t')}{(\text{let } x \ e_1 \ e_2, C, m, t) \rightsquigarrow (e_2, (x, \text{tick}(t')) :: C, m'[\text{tick}(t') \mapsto v], \text{tick}(t'))} \\
\\
\text{[LETML]} \frac{}{(\text{let } M \ e_1 \ e_2, C, m, t) \rightsquigarrow (e_1, C, m, t)} \quad \text{[LETMR]} \frac{(e_1, C, m, t) \rightsquigarrow (C', m', t')}{(\text{let } M \ e_1 \ e_2, C, m, t) \rightsquigarrow (e_2, (M, C') :: C, m', t')} \\
\\
\text{[LETE]} \frac{\begin{array}{c} (e_1, C, m, t) \rightsquigarrow (v, m', t') \\ (e_2, (x, \text{tick}(t')) :: C, m'[\text{tick}(t') \mapsto v], \text{tick}(t')) \rightsquigarrow (C', m'', t'') \end{array}}{(\text{let } x \ e_1 \ e_2, C, m, t) \rightsquigarrow (C', m'', t'')} \quad \text{[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}}{(\text{let } M \ e_1 \ e_2, C, m, t) \rightsquigarrow (C'', m'', t'')}
\end{array}$$

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 ${}^\circ\mathbb{T}$ that is connected to the concrete time domain via an

Abstract Time	${}^\circ t$	\in	${}^\circ \mathbb{T}$	
Environment/Context	${}^\circ C$	\in	${}^\circ \text{Ctx}$	
Value of expressions	${}^\circ v$	\in	${}^\circ \text{Val} \subseteq \text{Expr} \times {}^\circ \text{Ctx}$	
Value of expressions/modules	${}^\circ V$	\in	${}^\circ \text{ValCtx} \triangleq {}^\circ \text{Val} \uplus {}^\circ \text{Ctx}$	
Abstract Memory	${}^\circ m$	\in	${}^\circ \text{Mem} \triangleq {}^\circ \mathbb{T} \xrightarrow{\text{fin}} \mathcal{P}({}^\circ \text{Val})$	
Abstract State	${}^\circ s$	\in	${}^\circ \text{State} \triangleq {}^\circ \text{Ctx} \times {}^\circ \text{Mem} \times {}^\circ \mathbb{T}$	
Abstract Result	${}^\circ r$	\in	${}^\circ \text{Result} \triangleq {}^\circ \text{ValCtx} \times {}^\circ \text{Mem} \times {}^\circ \mathbb{T}$	
Abstract Tick	${}^\circ \text{tick}$	\in	${}^\circ \text{Tick} \triangleq {}^\circ \text{State} \times \text{Var} \times {}^\circ \text{Val} \rightarrow {}^\circ \mathbb{T}$	
Context	${}^\circ C$	\rightarrow	\square	empty stack
			$ (x, {}^\circ t) :: {}^\circ C$	expression binding
			$ (M, {}^\circ C) :: {}^\circ C$	module binding
Value of expressions	${}^\circ v$	\rightarrow	$\langle \lambda x.e, {}^\circ C \rangle$	closure

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

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

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:

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

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 ${}^\circ \rightsquigarrow$ is in Figure 7. ${}^\circ \rightsquigarrow \subseteq {}^\circ \text{Left} \times {}^\circ \text{Right}$, when ${}^\circ \text{Left} \triangleq \text{Expr} \times {}^\circ \text{State}$ and ${}^\circ \text{Right} \triangleq \text{Left} \uplus {}^\circ \text{Result}$.

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

$${}^\circ \alpha(C) \triangleq \begin{cases} \square & C = \square \\ (x, {}^\circ \alpha(t)) :: {}^\circ \alpha(C') & C = (x, t) :: C' \\ (M, {}^\circ \alpha(C')) :: {}^\circ \alpha(C'') & C = (M, C') :: C'' \end{cases} \quad {}^\circ \alpha(V) \triangleq \begin{cases} \langle \lambda x.e, {}^\circ \alpha(C) \rangle & V = \langle \lambda x.e, C \rangle \\ {}^\circ \alpha(C) & V = C \end{cases}$$

$${}^\circ \alpha(m) \triangleq \lambda {}^\circ t. \{ {}^\circ \alpha(m(t)) \mid {}^\circ \alpha(t) = {}^\circ t \text{ and } t \in \text{dom}(m) \}$$

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

Sketch. Induction on \rightsquigarrow . □

Then if we define $D^\# \triangleq \mathcal{P}({}^\circ \Sigma)$, when ${}^\circ \Sigma \triangleq {}^\circ \text{Right} \uplus {}^\circ \rightsquigarrow$, 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 $\alpha : D \rightarrow D^\#$ and $\gamma : D^\# \rightarrow D$ by:

$$\alpha(A) \triangleq \{ {}^\circ \alpha(\ell) {}^\circ \rightsquigarrow {}^\circ \alpha(\rho) \mid \ell \rightsquigarrow \rho \in A \} \cup \{ {}^\circ \alpha(\rho) \mid \rho \in A \}$$

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

Then it is straightforward to see that:

Claim 2.3 (Galois Connection). $\mathcal{P}(\Sigma) = D \xrightleftharpoons[\alpha]{\gamma} D^\# = \mathcal{P}({}^\circ \Sigma)$. 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

$$\text{Step}^\#(A^\#) \triangleq \left\{ {}^\circ \ell {}^\circ \rightsquigarrow {}^\circ \rho, {}^\circ \rho \left| \frac{A'^\#}{{}^\circ \ell {}^\circ \rightsquigarrow {}^\circ \rho} \text{ and } A'^\# \subseteq A^\# \text{ and } {}^\circ \ell \in A^\# \right. \right\}$$

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

$$\begin{array}{c}
\text{[EXPRID]} \frac{\circ t_x = \circ C(x) \quad \circ v \in \circ m(\circ t_x)}{(x, \circ C, \circ m, \circ t) \circ \rightsquigarrow (\circ v, \circ m, \circ t)} \quad \text{[FN]} \frac{}{(\lambda x. e, \circ C, \circ m, \circ t) \circ \rightsquigarrow (\langle \lambda x. e, \circ C \rangle, \circ m, \circ t)} \\
\\
\text{[APPL]} \frac{}{(e_1 e_2, \circ C, \circ m, \circ t) \circ \rightsquigarrow (e_1, \circ C, \circ m, \circ t)} \quad \text{[APPR]} \frac{(e_1, \circ C, \circ m, \circ t) \circ \rightsquigarrow (\langle \lambda x. e_\lambda, \circ C_\lambda \rangle, \circ m_\lambda, \circ t_\lambda)}{(e_1 e_2, \circ C, \circ m, \circ t) \circ \rightsquigarrow (e_2, \circ C, \circ m_\lambda, \circ t_\lambda)} \\
\\
\text{[APPBODY]} \frac{(e_1, \circ C, \circ m, \circ t) \circ \rightsquigarrow (\langle \lambda x. e_\lambda, \circ C_\lambda \rangle, \circ m_\lambda, \circ t_\lambda) \quad (e_2, \circ C, \circ m_\lambda, \circ t_\lambda) \circ \rightsquigarrow (\circ v, \circ m_a, \circ t_a)}{(e_1 e_2, \circ C, \circ m, \circ t) \circ \rightsquigarrow (e_\lambda, (x, \circ \text{tick}(\circ t_a)) :: \circ C_\lambda, \circ m_a[\circ \text{tick}(\circ t_a) \mapsto \circ v], \circ \text{tick}(\circ t_a))} \\
\\
\text{[APP]} \frac{(e_1, \circ C, \circ m, \circ t) \circ \rightsquigarrow (\langle \lambda x. e_\lambda, \circ C_\lambda \rangle, \circ m_\lambda, \circ t_\lambda) \quad (e_2, \circ C, \circ m_\lambda, \circ t_\lambda) \circ \rightsquigarrow (\circ v, \circ m_a, \circ t_a)}{(e_\lambda, (x, \circ \text{tick}(\circ t_a)) :: \circ C_\lambda, \circ m_a[\circ \text{tick}(\circ t_a) \mapsto \circ v], \circ \text{tick}(\circ t_a)) \circ \rightsquigarrow (\circ v', \circ m', \circ t')} \\
\\
\text{[LINKL]} \frac{}{(e_1 \bowtie e_2, \circ C, \circ m, \circ t) \circ \rightsquigarrow (e_1, \circ C, \circ m, \circ t)} \quad \text{[LINKR]} \frac{(e_1, \circ C, \circ m, \circ t) \circ \rightsquigarrow (\circ C', \circ m', \circ t')}{(e_1 \bowtie e_2, \circ C, \circ m, \circ t) \circ \rightsquigarrow (e_2, \circ C', \circ m', \circ t')} \\
\\
\text{[LINK]} \frac{(e_1, \circ C, \circ m, \circ t) \circ \rightsquigarrow (\circ C', \circ m', \circ t') \quad (e_2, \circ C', \circ m', \circ t') \circ \rightsquigarrow (\circ V, \circ m'', \circ t'')}{(e_1 \bowtie e_2, \circ C, \circ m, \circ t) \circ \rightsquigarrow (\circ V, \circ m'', \circ t'')} \quad \text{[EMPTY]} \frac{}{(\varepsilon, \circ C, \circ m, \circ t) \circ \rightsquigarrow (\circ C, \circ m, \circ t)} \\
\\
\text{[MODID]} \frac{\circ C' = \circ C(M)}{(M, \circ C, \circ m, \circ t) \circ \rightsquigarrow (\circ C', \circ m, \circ t)} \quad \text{[LETEL]} \frac{}{(\text{let } x e_1 e_2, \circ C, \circ m, \circ t) \circ \rightsquigarrow (e_1, \circ C, \circ m, \circ t)} \\
\\
\text{[LETERR]} \frac{(e_1, \circ C, \circ m, \circ t) \circ \rightsquigarrow (\circ v, \circ m', \circ t')}{(\text{let } x e_1 e_2, \circ C, \circ m, \circ t) \circ \rightsquigarrow (e_2, (x, \circ \text{tick}(\circ t')) :: \circ C, \circ m'[\circ \text{tick}(\circ t') \mapsto \circ v], \circ \text{tick}(\circ t'))} \\
\\
\text{[LETML]} \frac{}{(\text{let } M e_1 e_2, \circ C, \circ m, \circ t) \circ \rightsquigarrow (e_1, \circ C, \circ m, \circ t)} \quad \text{[LETMR]} \frac{(e_1, \circ C, \circ m, \circ t) \circ \rightsquigarrow (\circ C', \circ m', \circ t')}{(\text{let } M e_1 e_2, \circ C, \circ m, \circ t) \circ \rightsquigarrow (e_2, (M, \circ C') :: \circ C, \circ m', \circ t')} \\
\\
\text{[LETE]} \frac{(e_1, \circ C, \circ m, \circ t) \circ \rightsquigarrow (\circ v, \circ m', \circ t') \quad (e_2, (x, \circ \text{tick}(\circ t')) :: \circ C, \circ m'[\circ \text{tick}(\circ t') \mapsto \circ v], \circ \text{tick}(\circ t')) \circ \rightsquigarrow (\circ C', \circ m'', \circ t'')}{(\text{let } x e_1 e_2, \circ C, \circ m, \circ t) \circ \rightsquigarrow (\circ C', \circ m'', \circ t'')} \\
\\
\text{[LETM]} \frac{(e_1, \circ C, \circ m, \circ t) \circ \rightsquigarrow (\circ C', \circ m', \circ t') \quad (e_2, (M, \circ C') :: \circ C, \circ m', \circ t') \circ \rightsquigarrow (\circ C'', \circ m'', \circ t'')}{(\text{let } M e_1 e_2, \circ C, \circ m, \circ t) \circ \rightsquigarrow (\circ C'', \circ m'', \circ t'')}
\end{array}$$

Figure 7: The abstract one-step transition relation.

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

$$\llbracket e \rrbracket^\# S^\# \triangleq \text{Ifp}(\lambda X^\#. \text{Step}^\#(X^\#) \cup \{(e, \circ s) \mid \circ s \in S^\#\})$$

Then we can prove that:

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

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

2.5 Finiteness of the Abstract Semantics

Now we can say that $\llbracket e \rrbracket^\# \alpha(S)$ is a sound abstraction of $\llbracket e \rrbracket S$. However, is it true that $\llbracket e \rrbracket^\# \alpha(S)$ is finitely computable?

Since $^\circ\mathbb{T}$ 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 $X \rightarrow [] \mid x :: X \mid (M, X) :: X$. Then we may inductively define $\lfloor C \rfloor$ and $\lfloor ^\circ C \rfloor$ to be the signatures that are obtained by stripping all timestamps from C and $^\circ C$. Moreover, we may define $\lfloor m \rfloor \triangleq \{ \lfloor C \rfloor \mid \exists t : \langle _, C \rangle = m(t) \}$ and $\lfloor ^\circ m \rfloor \triangleq \{ \lfloor ^\circ C \rfloor \mid \exists t : \langle _, ^\circ C \rangle \in ^\circ m(t) \}$ 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) . $\lfloor ^\circ \rho \rfloor$ can be analogously defined.

If we can prove that for all e and $^\circ s$, there exists a finite set $X \subseteq \text{Sig}$ that bounds $\bigcup_{(e, ^\circ s) \circ \rightsquigarrow^* \rho} \lfloor ^\circ \rho \rfloor \subseteq X$, we can show that all reachable $^\circ \rho$ s are finite, since $^\circ\mathbb{T}$ 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 (Finiteness of the Abstract Semantics). If $S^\#$ is finite, $\llbracket e \rrbracket^\# S^\#$ is finite.

Sketch. By existence of a function that computes all reachable signatures and by induction on $^\circ \rightsquigarrow$.

To elaborate, let this function be called $f : ^\circ\text{Right} \rightarrow \mathcal{P}(\text{Sig})$. We need two lemmas, the first being $\forall ^\circ \rho : \lfloor ^\circ \rho \rfloor \subseteq f(^ \circ \rho)$ and the second being $\forall ^\circ \ell, ^\circ \rho : ^\circ \ell \rightsquigarrow^* \rho \Rightarrow f(^ \circ \rho) \subseteq f(^ \circ \ell)$.

If we have the lemmas, then for all $^\circ \rho$ such that $(e, ^\circ s) \circ \rightsquigarrow^* \rho$, $\lfloor ^\circ \rho \rfloor \subseteq f(^ \circ \rho) \subseteq f(e, ^\circ s)$, thus all signatures in $\llbracket e \rrbracket^\# S^\#$ can be bound by $X \triangleq \bigcup_{s \in S^\#} f(e, ^\circ s)$, which is finite. The lemmas are proven in Coq. \square

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 with static scoping, this property will hold, even with features such as functors or first-class modules.

3 Part III: Linking in the Instrumented Semantics

Now we need to define an injection operation that fills in the blanks of a $r = (V, m, t) \in \text{Result}$ with a $s = (C', m', t') \in \text{State}$. 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 , $^\circ\mathbb{T}$, $^\circ\alpha$, and $^\circ\text{tick}$ also be linked. Concretely, we demand that the linked tick_+ preserves the condition that $^\circ\alpha_+ \circ \text{tick}_+ = ^\circ\text{tick}_+ \circ ^\circ\alpha_+$. 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_+ , $^\circ\alpha_+$, $^\circ\text{tick}$

We must first define tick_+ , $^\circ\alpha_+$, and $^\circ\text{tick}_+$ that satisfies the condition that $t_+ \leq \text{tick}_+(t_+)$ and $^\circ\alpha_+ \circ \text{tick}_+ = ^\circ\text{tick}_+ \circ ^\circ\alpha_+$. 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:

$$\text{tick}_+(t) \triangleq \begin{cases} \text{tick}_1(t) & t \in \mathbb{T}_1 \\ \text{tick}_2(t) & t \in \mathbb{T}_2 \end{cases} \quad ^\circ\alpha_+(t) \triangleq \begin{cases} ^\circ\alpha_1(t) & t \in \mathbb{T}_1 \\ ^\circ\alpha_2(t) & t \in \mathbb{T}_2 \end{cases} \quad ^\circ\text{tick}_+(^ \circ t) \triangleq \begin{cases} ^\circ\text{tick}_1(^ \circ t) & ^ \circ t \in ^\circ\mathbb{T}_1 \\ ^\circ\text{tick}_2(^ \circ t) & ^ \circ t \in ^\circ\mathbb{T}_2 \end{cases}$$

it is easy to check that all requirements are satisfied.

3.2 Injection

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

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

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

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 $s_1 \in \text{State}_1$, $\ell_2 \in \text{Left}_2$ and $\rho_2 \in \text{Right}_2$, 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. \square

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

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

$$S_1 \triangleright A_2 \triangleq \{\rho_2\langle s_1 \rangle \mid s_1 \in S_1 \text{ and } \rho_2 \in A_2\} \cup \{\ell_2\langle s_1 \rangle \rightsquigarrow \rho_2\langle s_1 \rangle \mid s_1 \in S_1 \text{ and } \ell_2 \rightsquigarrow \rho_2 \in A_2\}$$

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

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

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

$$\llbracket e \rrbracket (S_1 \triangleright S_2) = S_1 \times \llbracket e \rrbracket S_2$$

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 ${}^\circ m_2\langle {}^\circ C_1 \rangle$ in Figure 8 has to be adapted to account for the fact that ${}^\circ m_2({}^\circ t)$ is now a *set* of closures and thus injection of ${}^\circ C_1$ must be mapped over all elements. Then we can show that:

Claim 3.3 (Injection Preserves Abstract Evaluation). For all ${}^\circ s_1 \in {}^\circ \text{State}_1$, ${}^\circ \ell_2 \in {}^\circ \text{Left}_2$ and ${}^\circ \rho_2 \in {}^\circ \text{Right}_2$, if ${}^\circ \ell_2 \rightsquigarrow {}^\circ \rho_2$ under ${}^\circ \text{tick}_2$, then ${}^\circ \ell_2\langle {}^\circ s_1 \rangle \rightsquigarrow {}^\circ \rho_2\langle {}^\circ s_1 \rangle$ under ${}^\circ \text{tick}_+$.

Sketch. Induction on ${}^\circ \rightsquigarrow$ under ${}^\circ \text{tick}_2$. \square

and thus we can define:

Definition 3.3 (Abstract Injection). For $S_1^\# \subseteq {}^\circ \text{State}_1$ and $A_2^\# \in D_2^\#$, define:

$$S_1^\# \triangleright^\# A_2^\# \triangleq \{{}^\circ \rho_2\langle {}^\circ s_1 \rangle \mid {}^\circ s_1 \in S_1^\# \text{ and } {}^\circ \rho_2 \in A_2^\#\} \cup \{{}^\circ \ell_2\langle {}^\circ s_1 \rangle \rightsquigarrow {}^\circ \rho_2\langle {}^\circ s_1 \rangle \mid {}^\circ s_1 \in S_1^\# \text{ and } {}^\circ \ell_2 \rightsquigarrow {}^\circ \rho_2 \in A_2^\#\}$$

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

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

so that the *best possible result* is achieved:

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

$$\llbracket e \rrbracket^\# (S_1^\# \triangleright^\# S_2^\#) = S_1^\# \times^\# \llbracket e \rrbracket^\# S_2^\#$$

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:

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

4 Part IV: A Criteria for Reusing Analysis Results

4.1 Motivation

Assume that we have a cached analysis result of a program fragment e under an abstract state $^\circ s$ that uses timestamps in $\{0, 1\}$. We want to analyze e under another initial state $^\circ s'$ that uses abstract timestamps in $\{a, b\}$. The problem is: what is the criteria for reusing the cached analysis results?

Based on the previous sections, we want to find a $^\circ s''$ that satisfies $^\circ s \langle^\circ s''\rangle = ^\circ s'$, so that linking $^\circ s''$ into the cached results result in the semantics that started from $^\circ s'$. However, equality is not possible because the timestamps that are used are different. Thus, in this section, we define what it means for semantics that use different timestamps to be *equivalent*. The definition of equivalence need to satisfy two desired properties, namely:

1. If $^\circ s$ and $^\circ s'$ are equivalent, all $s \in \gamma(\{^\circ s\})$ must have an equivalent $s' \in \gamma'(\{^\circ s'\})$ and vice versa.
2. If s and s' are equivalent, all transitions made by \rightsquigarrow starting from (e, s) and (e, s') must go to equivalent states.

These two properties ensure that if we find a $^\circ s''$ such that $^\circ s \langle^\circ s''\rangle$ is *equivalent* to $^\circ s'$, linking $^\circ s''$ with the cached results will result in an overapproximation of something *equivalent* to the semantics that started from $\gamma'(\{^\circ s'\})$.

4.2 Definitions

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

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 reachable timestamps on the vertices. Also, given a $\varphi \in \mathbb{T} \rightarrow \mathbb{T}'$, 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 Figure 9. From now on, we shall write Path for the set of access paths that use timestamps in \mathbb{T} , and Path' for the set

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

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

of access paths that use timestamps in \mathbb{T}' .

Then given an access path, we can define a predicate $\text{valid} \in (\text{Ctx} \uplus \mathbb{T}) \times \text{Mem} \times \text{Path} \rightarrow \text{Prop}$. 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 .

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

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

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

Now we can give 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 $s \sim s'$ when there exists a $\varphi : \mathbb{T} \rightarrow \mathbb{T}'$ and $\varphi^{-1} : \mathbb{T}' \rightarrow \mathbb{T}$ 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 (Weakly Equivalent Abstract States). Let $\circ s = (\circ C, \circ m, \circ t) \in \circ \text{State}$ and $\circ s' = (\circ C', \circ m', \circ t') \in \circ \text{State}'$. We say $\circ s$ is *weakly equivalent* to $\circ s'$ when there exists a $\circ \varphi : \circ \mathbb{T} \rightarrow \circ \mathbb{T}'$ and $\circ \varphi^{-1} : \circ \mathbb{T}' \rightarrow \circ \mathbb{T}$ such that:

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

The reason that the above definition is called “weak equivalence” is because

$$[(x, 0)], \{0 \mapsto \{\langle \lambda z.z, [(x, 1)] \rangle, \langle \lambda z.z, [(y, 2)] \rangle\}, 1 \mapsto \{\langle \lambda z.z, [] \rangle\}\}$$

and

$$[(x, 0)], \{0 \mapsto \{\langle \lambda z.z, [(x, 1); (y, 2)] \rangle\}, 1 \mapsto \{\langle \lambda z.z, [] \rangle\}\}$$

become weakly equivalent. However, the concretizations of the two states are not equivalent in the concrete sense. Therefore, the definition needs to be strengthened to require that for each reachable address $\circ t$, all abstract closures that are stored in that address have a weakly equivalent counterpart in the other memory.

Definition 4.3 (Equivalent Abstract States). Let $\circ s = (\circ C, \circ m, \circ t) \in \circ \text{State}$ and $\circ s' = (\circ C', \circ m', \circ t') \in \circ \text{State}'$. We say $\circ s$ is *equivalent* to $\circ s'$ and write $\circ s \sim \circ s'$ when there exists a $\circ \varphi : \circ \mathbb{T} \rightarrow \circ \mathbb{T}'$ and $\circ \varphi^{-1} : \circ \mathbb{T}' \rightarrow \circ \mathbb{T}$ such that:

1. $\circ s$ and $\circ s'$ are weakly equivalent by $\circ \varphi, \circ \varphi^{-1}$.
2. For each $\circ t$ reachable from $\circ s$, if $\langle \lambda x.e, \circ C \rangle \in \circ m(\circ t)$, there exists a $\circ C'$ such that $\langle \lambda x.e, \circ C' \rangle \in \circ m'(\circ \varphi(\circ t))$ and $\circ C, \circ C'$ are weakly equivalent by $\circ \varphi, \circ \varphi^{-1}$ under the empty memory.
3. The same holds for each $\circ t'$ reachable from $\circ s'$.

We also define equivalence between results $(V, m, t) \in \text{Result}$ and $(V', m', t') \in \text{Result}'$ by the conjunction of the equality between the expression parts of V and V' and equivalence between the context parts. Also, equivalence between $\ell = (e, s) \in \text{Left}$ and $\ell' = (e', s') \in \text{Left}'$ can be similarly defined as $e = e'$ and $s \sim s'$.

Then the equivalence between elements of D and D' , the CPOs, can be defined by lifting the definition of \sim . That is, when $A \in D$ and $A' \in D'$, A and A' are defined to be equivalent iff for each element of A , there exists an equivalent element of A' and vice versa. Equivalence between $\circ A \in \circ D$ and $\circ A' \in \circ D'$ can be defined analogously.

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

$$\ell \rightsquigarrow \rho \text{ and } \ell \sim \ell' \Rightarrow \exists \rho' : \ell' \rightsquigarrow \rho' \text{ and } \rho \sim \rho'$$

Thus, if $S \subseteq \text{State}$ and $S' \subseteq \text{State}'$ are equivalent, $\llbracket e \rrbracket S \sim \llbracket e \rrbracket S'$.

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

Then we can strengthen the claim.

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

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

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 $\llbracket e \rrbracket \gamma(S^\#) \sim \llbracket e \rrbracket \gamma'(S'^\#) \subseteq \gamma'(\llbracket e \rrbracket^\# S'^\#)$ from $S^\# \sim^\# S'^\#$. The first \sim 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 .

Claim 4.2 (Concretization Preserves Equivalence). For all $S^\# \subseteq {}^\circ\text{State}$ and $S'^\# \subseteq {}^\circ\text{State}'$,

$$S^\# \sim^\# S'^\# \Rightarrow \gamma(S^\#) \sim \gamma'(S'^\#)$$

Sketch. We want to prove:

$$\forall s \in \text{State}, {}^\circ s' \in {}^\circ\text{State}' : {}^\circ\alpha(s) \sim {}^\circ s' \Rightarrow \exists s' \in \text{State}' : s \sim s' \text{ and } {}^\circ\alpha'(s') = {}^\circ s'$$

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

Proving the same statement in the opposite side leads to $\forall s' \in \gamma(s'^\#) : \exists s \in \gamma(s^\#) : s \sim s'$, so that $\gamma(S^\#) \sim \gamma'(S'^\#)$. 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 ${}^\circ n'$ of ${}^\circ s'$ by ${}^\circ\varphi \circ {}^\circ\alpha$ (when ${}^\circ\varphi$ gives the equivalence between ${}^\circ\alpha(s)$ and ${}^\circ s'$) and (2) lifting ${}^\circ n'$ 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 ${}^\circ s'$. \square