# A Syntax-Guided Framework for Modular Analysis

JOONHYUP LEE

## 1 INTRODUCTION

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 interfaced through *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. The problem is: 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*.

### 1.1 Separate Static Analyis

To illustrate what we mean by the "fractionally closed semantics", we first give a concrete example.

```
(* Module M *)
let x = 1
```

```
(* Module F *)
let fix fact n =
  if n <= 0 then 1
  else n * fact (n - 1)
```

```
(* Client code *)
Include M
Include F
let ret = (F.fact 100) + M.x
```

Above, we have a piece of code that adds an integer `x` exported by the module `M` to the result of 100!. Given this program, a compiler that supports separate compilation produces object files that can be linked with different implementations of the module `M`. What we desire is some sort of semantic object for static analyzers that corresponds to such object files. Since object files represent programs with unresolved variable references, we say that they are fractionally closed.

Defining separate analysis results and linking allows discussion for a wide variety of cases. Say that the client code is analyzed with *only* assuming the implementation for `F.fact`. Thus, the analysis result, if well defined, will contain the information that the unresolved variable `M.x` must

Author's address: Joonhyup Lee.


XXXX-XXXX/2023/10-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

be added to 100!. Later, when the full implementation of the modules are known, we simply link what was missing with the separate analysis results.

Such an approach is useful in two ways:

**Rely-guarantee**

When the client code is linked with another implementation of `F`, check whether `fact` is changed, and if it is not changed, simply inject the rest into the analysis results.

**Incrementality**

If the implementation of `x` is changed, it will not trigger re-analysis of the whole program.

## 2 UNCOVERING MODULARITY IN OPERATIONAL SEMANTICS

First we introduce our model language. The language is basically an extension of untyped lambda calculus with modules and the linking construct.

| | | | | |
|---|---|---|---|---|
| Identifiers | $x, M$ | $\in$ | Var | |
| Expression | $e$ | $\longrightarrow$ | $x \mid \lambda x.e \mid e\ e$ | untyped $\lambda$-calculus |
| | | $\mid$ | $e \rtimes e$ | linked expression |
| | | $\mid$ | $\varepsilon$ | empty module |
| | | $\mid$ | $M$ | module identifier |
| | | $\mid$ | `let` $x\ e\ e$ | expression binding |
| | | $\mid$ | `let` $M\ e\ e$ | module binding |

Fig. 1. Abstract syntax of the simple module language.

### 2.1 Rationale for the design of the simple language

There are no recursive modules, first-class modules, or functors in the simple language that is defined. Also, note that modules and expressions are not separated in the syntax. Why is this so?

The rationale for the exclusion of recursive modules/first-class modules/functors is because we want to be able to determine the *shape* of the environment at every program point before running the program. To enforce this property when function applications might return modules, we need to employ signatures to project the dynamically computed modules onto a statically known shape. Concretely, we need to define signatures $X$ to resolve (1) the variable $M$ in the body of $\lambda M :> X.e$, and (2) the result of a dynamically computed $(e_1\ e_2) :> X$. To simplify the presentation, we first consider the case that does not require signatures.

The rationale for not separating modules and expressions in the syntax is because we want to utilize the linking construct to link both modules to expressions and modules to modules. That is, we want expressions to be parsed as $(m_1 \rtimes m_2) \rtimes e$. $m_1 \rtimes m_2$ links a module with a module, and $(m_1 \rtimes m_2) \rtimes e$ links a module with an expression. Why this is convenient will be clear when we explain separate analysis; we want to link modules with modules as well as expressions.

### 2.2 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 $\ell$ of $\mathsf{Left} \triangleq \mathsf{Expr} \times \mathsf{Ctx}$ with an element $\rho$ of $\mathsf{Right} \triangleq \mathsf{Left} \uplus \mathsf{ValCtx}$. 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, it relates a configuration not only to its final result but also

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

Fig. 2. Definition of the semantic domains.

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

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

$$[\textsc{AppR}]\ \frac{(e_1, C) \rightsquigarrow \langle \lambda x.e_\lambda, C_\lambda \rangle}{(e_1\ e_2, C) \rightsquigarrow (e_2, C)} \qquad [\textsc{AppBody}]\ \frac{(e_1, C) \rightsquigarrow \langle \lambda x.e_\lambda, C_\lambda \rangle \quad (e_2, C) \rightsquigarrow v}{(e_1\ e_2, C) \rightsquigarrow (e_\lambda, (x, v) :: C_\lambda)}$$

$$[\textsc{App}]\ \frac{(e_1, C) \rightsquigarrow \langle \lambda x.e_\lambda, C_\lambda \rangle \quad (e_2, C) \rightsquigarrow v \quad (e_\lambda, (x, v) :: C_\lambda) \rightsquigarrow v'}{(e_1\ e_2, C) \rightsquigarrow v'}$$

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

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

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

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

Fig. 3. The concrete one-step transition relation.

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.

### 2.3 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 $\Sigma \triangleq \rightsquigarrow \uplus \mathsf{Right}$ and $\mathcal{P}(S)$ is the powerset of $S$. The semantics of an expression $e$ starting from initial states in $S \subseteq \mathsf{Ctx}$ is the collection of $\ell \rightsquigarrow \rho$ and $\rho$ derivable from initial configurations $(e, C)$ with $C \in S$. Defining the transfer function is straightforward from the definition of the transition relation.

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

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

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. In fact, we can prove that it is continuous, as it preserves the least upper bound of chains. Now, because of Tarski's fixpoint theorem, we can formulate the collecting semantics in fixpoint form.

**Definition 2.2** (Collecting semantics)**.** Given $e \in \mathsf{Expr}$ and $S \subseteq \mathsf{Ctx}$, 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.

### 2.4 Semantic Linking

Now we present a natural notion of *semantic linking* that, given a (1) (possibly incomplete) proof tree of an expression $e$ under some initial context $C_1$ and (2) some external context $C_2$, gives the meaning of $e$ under the *linked* context of $C_1$ and $C_2$. Thus, it will be clear how analysis results obtained locally can be reused to obtain the meaning of the whole program, all at the level of the operational semantics.

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

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

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

**Lemma 2.1** (Fill in the Blanks)**.** *For all $C_1, C_2 \in \mathsf{Ctx}$, 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 $C_1 \in \mathsf{Ctx}, \ell = (e, C_2) \in \mathsf{Left}$ as $(e, C_2\langle C_1 \rangle)$, we have:

**Lemma 2.2** (Injection Preserves Evaluation)**.** *For all $\ell \in \mathsf{Left}$, $\rho \in \mathsf{Right}$, $\ell \rightsquigarrow \rho \Rightarrow \ell\langle C \rangle \rightsquigarrow \rho\langle C \rangle$. 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 an element $A$ of $D$ and a semantic linking operation $\infty$ that does the rest of the computation:

**Definition 2.3** (Injection)**.** For $S \subseteq \mathsf{Ctx}$ and $A \in D$, define:

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

**Definition 2.4** (Semantic Linking). For $S \subseteq \mathsf{Ctx}$ and $A \in D$, define:

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

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

**Theorem 2.1** (Advance). *For all* $e \in \mathsf{Expr}$ *and* $S_1, S_2 \subseteq \mathsf{Ctx}$,

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

Proof. Let $A$ be $\{(e, C) | C \in S_1 \triangleright S_2\}$, and let $B$ be $S_1 \triangleright [\![e]\!]S_2$. Note that $A \subseteq B$ by the definition of $[\![e]\!]S_2$. Also, let $X_A$ be $\mathsf{lfp}(\lambda X.\mathsf{Step}(X) \cup A) = [\![e]\!](S_1 \triangleright S_2)$ and let $X_B$ be $\mathsf{lfp}(\lambda X.\mathsf{Step}(X) \cup B) = S_1 \between [\![e]\!]S_2$. Since injection preserves evaluation, 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 of $\lambda X.\mathsf{Step}(X) \cup A$. Since $X_B$ is a pre-fixed point, $X_A \subseteq X_B$.

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

## 2.5 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 $\alpha$ and concretization $\gamma$:

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

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

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

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

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

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

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

Then we can soundly approximate fractional specifications by:

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

## 3 INSTRUMENTED SEMANTICS

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 $S \subseteq \mathsf{Ctx}$ to finitely compute an overapproximation. However, devising such an abstraction is not immediately obvious.

Consider an abstract context $C^\#$ that overapproximates $\gamma(C^\#) \subseteq \mathsf{Ctx}$. For a variable $x$, we expect that $\gamma(C^\#(x)) \supseteq \{C(x) | C \in \gamma(C^\#)\}$, when the operation $C^\#(x)$ reads the abstract closure bound to $x$ from $C^\#$. Thus, there is a recursive structure, where reading from an abstract context must return an abstract closure again containing an abstract context.

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.

### 3.1 Semantic Domains

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

Fig. 4. Definition of the instrumented semantic domains.

The domains for defining the operational semantics is extended to include the *time* and *memory*. Compared with Figure 2, Figure 4 defines four more sets, $\mathbb{T}$, Mem, State, and Result. A $s = (C, m, t) \in \mathsf{State}$ corresponds to a $C$ in Figure 2, as the pair $(C, m)$ cooperates to represent the recursively defined $C$ in the original representation. Similarly, a $r = (V, m, t) \in \mathsf{Result}$ 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.

### 3.2 Operational Semantics

An excerpt of the instrumented operational semantics is given 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, $\mathsf{tick}(t) \notin \mathsf{supp}(C, m)$ must be guaranteed, when $\mathsf{supp}(C, m)$ is the set

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

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

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

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

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

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

Fig. 5. Excerpt of the concrete instrumented semantics, corresponding to the big-step evaluation rules.

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 that $C \leq t$ and $m \leq t$, when $C \leq t$ means that all timestamps in $C$ are bound by $t$, and $m \leq t$ means that all timestamps allocated in the memory are bound by $t$.

Then the criteria that $\mathsf{tick}(t)$ must be fresh is formalized by demanding that:

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

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 $\mathbb{T}$. Now, to allow only such valid transitions, we define:

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

as the set of *valid* states that enable tick to generate fresh timestamps. It is almost trivial that the set $\mathsf{Left} \times \mathsf{Right}$, when $\mathsf{Left} \triangleq \mathsf{Expr} \times \mathsf{State}$ and $\mathsf{Right} \triangleq \mathsf{Left} \uplus \mathsf{Result}$, is *closed* under the inductive definition of $\rightsquigarrow$. That is,

**Lemma 3.1** (Valid States Transition to Valid States)**.** *For all $\ell \in \mathsf{Left}$ and $\rho$, if $\ell \rightsquigarrow \rho$ according to the inductive rules, $\rho \in \mathsf{Right}$.*

Sketch. Induction on $\rightsquigarrow$. □

### 3.3 Collecting Semantics

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

**Definition 3.1** (Transfer function)**.** Given $A \in D$, define

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

and

**Definition 3.2** (Collecting semantics)**.** Given $e \in \mathsf{Expr}$ and $S \subseteq \mathsf{State}$, define:

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

## 4 ABSTRACT SEMANTICS

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

Fig. 6. Definition of the semantic domains in the abstract case.

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

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. An excerpt for the abstract version of the operational semantics $^\circ\rightsquigarrow$ is in Figure 7. $^\circ\rightsquigarrow \subseteq {}^\circ\mathsf{Left} \times {}^\circ\mathsf{Right}$, when $^\circ\mathsf{Left} \triangleq \mathsf{Expr} \times {}^\circ\mathsf{State}$ and $^\circ\mathsf{Right} \triangleq {}^\circ\mathsf{Left} \uplus {}^\circ\mathsf{Result}$.

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:

$$\begin{aligned}
{}^\circ\alpha([]) &\triangleq [] \\
{}^\circ\alpha((x, t_x) :: C) &\triangleq (x, {}^\circ\alpha(t_x)) :: {}^\circ\alpha(C) \\
{}^\circ\alpha((M, C_M) :: C) &\triangleq (M, {}^\circ\alpha(C_M)) :: {}^\circ\alpha(C) \\
{}^\circ\alpha(\langle \lambda x.e, C \rangle) &\triangleq \langle \lambda x.e, {}^\circ\alpha(C) \rangle \\
{}^\circ\alpha(m) &\triangleq \lambda\, {}^\circ t.\{{}^\circ\alpha(m(t)) | \, {}^\circ\alpha(t) = {}^\circ t \text{ and } t \in \mathsf{dom}(m)\} \\
{}^\circ\alpha(e, C, m, t) &\triangleq (e, {}^\circ\alpha(C), {}^\circ\alpha(m), {}^\circ\alpha(t)) \\
{}^\circ\alpha(V, m, t) &\triangleq ({}^\circ\alpha(V), {}^\circ\alpha(m), {}^\circ\alpha(t))
\end{aligned}$$

We have:

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

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

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

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

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

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

Fig. 7. Excerpt of the abstract operational semantics, corresponding to the big-step evaluation rules.

**Lemma 4.1** (Operational Soundness). *For all* $\ell \in \mathsf{Left}$ *and* $\rho \in \mathsf{Right}$, *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 ${}^\circ\Sigma \triangleq {}^\circ\mathsf{Right} \uplus {}^\circ\!\rightsquigarrow$, we can establish a Galois connection between $D$ and $D^\#$. The abstraction and concretization functions are given by:

**Definition 4.1** (Abstraction and Concretization). Define $\alpha : D \to D^\#$ and $\gamma : D^\# \to D$ by:

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

Then it is straightforward to see that:

**Lemma 4.2** (Galois Connection). $\mathcal{P}(\Sigma) = D \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} D^\# = \mathcal{P}({}^\circ\Sigma)$. *That is:*

$$\forall A \in D, A^\# \in D^\# : \alpha(A) \subseteq A^\# \Leftrightarrow A \subseteq \gamma(A^\#)$$

Sketch. Straightforward from the definitions of $\alpha$ and $\gamma$. □

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

**Definition 4.2** (Abstract transfer function). Given $A^\# \in D^\#$, define

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

**Definition 4.3** (Abstract semantics). Given $e \in \mathsf{Expr}$ and $S^\# \subseteq {}^\circ\mathsf{State}$, define:

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

Then we can prove that:

**Theorem 4.1** (Soundness). *For all* $e \in \mathsf{Expr}$, $[\![e]\!] \circ \gamma \subseteq \gamma \circ [\![e]\!]^\#$.

Proof. From operational soundness, we have that $\alpha \circ \mathsf{Step} \subseteq \mathsf{Step}^\# \circ \alpha$. Then by the fixpoint transfer theorem and Galois connection, we have our desired result. □

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? The answer to this question is "yes".

**Theorem 4.2** (Finiteness). *For all* $e \in \mathsf{Expr}$ *and* $S^\# \subseteq {}^\circ\mathsf{Ctx}$, *if* $S^\#$ *is finite,* $[\![e]\!]^\#S^\#$ *is finite.*

Sketch. Given ${}^\circ s \in S^\#$, we want to prove that there is some finite set $X$ satisfying:

$$\forall\, {}^\circ\rho \in {}^\circ\mathsf{Right} : (e, {}^\circ s)\ {}^\circ\!\rightsquigarrow^*\ {}^\circ\rho \Rightarrow {}^\circ\rho \in X$$

Note that ${}^\circ\rho$ is of the form $(\langle \lambda x.e', {}^\circ C\rangle, {}^\circ m, {}^\circ t)$ or $({}^\circ C, {}^\circ m, {}^\circ t)$ or $(e', {}^\circ C, {}^\circ m, {}^\circ t)$. Since there is a finite number of abstract timestamps, we only have to show that there is a finite number of *shapes* of ${}^\circ\rho$ that is stripped of the timestamps. This is proven in Coq (in `Abstract.v`). □

## 5 LINKING IN THE INSTRUMENTED SEMANTICS

Now we need to define an injection operation that fills in the blanks of a $r = (V, m, t) \in \mathsf{Result}$ 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 $\mathsf{Result}$ 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 ${}^\circ\mathsf{tick}$ also be linked. Concretely, we demand that the linked $\mathsf{tick}_+$ preserves the condition that ${}^\circ\alpha_+ \circ \mathsf{tick}_+ = {}^\circ\mathsf{tick}_+ \circ {}^\circ\alpha_+$. Also, we need to link tick well so that for all valid transitions $\ell_2 \rightsquigarrow \rho_2$ under $\mathsf{tick}_2$, $\ell_2\langle s_1\rangle \rightsquigarrow \rho_2\langle s_1\rangle$ is also a valid transition under $\mathsf{tick}_+$. 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.

### 5.1 $\mathsf{tick}_+$, ${}^\circ\alpha_+$, ${}^\circ\mathsf{tick}$

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

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

it is easy to check that all requirements are satisfied.

### 5.2 Concrete Linking

Now we define injection between $s_1 = (C_1, m_1, t_1) \in \mathsf{State}_1$ and $r_2 = (V_2, m_2, t_2) \in \mathsf{Result}_2$:

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

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

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, $r_2\langle s_1\rangle \in \mathsf{Result}_+$ with respect to the ordering $\leq_+$. 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,

**Lemma 5.1** (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* $\mathsf{tick}_2$, *then* $\ell_2\langle s_1\rangle \rightsquigarrow \rho_2\langle s_1\rangle$ *under* $\mathsf{tick}_+$.

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

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

**Definition 5.1** (Injection). For $S_1 \subseteq \mathsf{State}_1$ and $A_2 \in D_2$, define:

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

**Definition 5.2** (Semantic Linking). For $S_1 \subseteq \mathsf{State}_1$ and $A_2 \in D_2$, define:

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

**Theorem 5.1** (Advance). *For all* $e \in \mathsf{Expr}$ *and* $S_1 \subseteq \mathsf{State}_1$, $S_2 \subseteq \mathsf{State}_2$,

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

### 5.3 Abstract Linking

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 **??** 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:

**Lemma 5.2** (Injection Preserves Abstract Evaluation). *For all* ${}^\circ s_1 \in {}^\circ\mathsf{State}_1$, ${}^\circ \ell_2 \in {}^\circ\mathsf{Left}_2$ *and* ${}^\circ \rho_2 \in {}^\circ\mathsf{Right}_2$, *if* ${}^\circ \ell_2 \, {}^\circ\!\rightsquigarrow {}^\circ \rho_2$ *under* ${}^\circ\mathsf{tick}_2$, *then* ${}^\circ \ell_2\langle {}^\circ s_1\rangle \, {}^\circ\!\rightsquigarrow {}^\circ \rho_2\langle {}^\circ s_1\rangle$ *under* ${}^\circ\mathsf{tick}_+$.

Sketch. Induction on ${}^\circ\!\rightsquigarrow$ under ${}^\circ\mathsf{tick}_2$. □

and thus we can define:

**Definition 5.3** (Abstract Injection). For $S_1^\# \subseteq {}^\circ\mathsf{State}_1$ and $A_2^\# \in D_2^\#$, define:

$$S_1^\# \triangleright^\# A_2^\# \triangleq \{{}^\circ \rho_2\langle {}^\circ s_1\rangle | {}^\circ s_1 \in S_1^\#, {}^\circ \rho_2 \in A_2^\#\} \cup \{{}^\circ \ell_2\langle {}^\circ s_1\rangle \, {}^\circ\!\rightsquigarrow {}^\circ \rho_2\langle {}^\circ s_1\rangle | {}^\circ s_1 \in S_1^\#, {}^\circ \ell_2 \, {}^\circ\!\rightsquigarrow {}^\circ \rho_2 \in A_2^\#\}$$

**Definition 5.4** (Abstract Linking). For $S_1^\# \subseteq {}^\circ\mathsf{State}_1$ and $A_2^\# \in D_2^\#$, define:

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

so that the *best possible result* is achieved:

**Theorem 5.2** (Abstract Advance). *For all* $e \in \mathsf{Expr}$ *and* $S_1^\# \subseteq {}^\circ\mathsf{State}_1$, $S_2^\# \subseteq {}^\circ\mathsf{State}_2$,

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

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 theorem directly leads to overapproximation by:

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

## 6 EQUIVALENCE BETWEEN INITIAL STATES

### 6.1 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 section, we want to find a $s''$ 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 $s \in \gamma(\{{}^\circ s\}) \Rightarrow$ must have an equivalent $s' \in \gamma'(\{{}^\circ s'\})$.
(2) If $s$ and $s'$ are equivalent, $(e, s)$ and $(e, s')$ must step 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 $\gamma'(\{{}^\circ s'\})$.

### 6.2 Definitions

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

We first define what it means for two states $s \in \mathsf{State}$ and $s' \in \mathsf{State}'$ to be equivalent. Recall 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 are more recent than the contexts and memories, tick will continue producing fresh addresses. Thus, the notion of equivalence is defined by how $C$ and $m$ components "look the same".

Note that information in $C$ and $m$ is 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 as representing the way that $(C, m)$ is *viewed*. 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 $\mathbb{T}'$.

$$\begin{array}{rcll}
p & \to & \epsilon & \text{empty path} \qquad \varphi(\epsilon) \triangleq \epsilon \\
& | & \xrightarrow{x} t\ p & \text{address access} \qquad \varphi(\xrightarrow{x} t\ p) \triangleq \xrightarrow{x} \varphi(t)\ \varphi(p) \\
& | & \xrightarrow{M} p & \text{module access} \qquad \varphi(\xrightarrow{M} p) \triangleq \xrightarrow{M} \varphi(p) \\
& | & \xrightarrow{\lambda x.e} p & \text{value access} \qquad \varphi(\xrightarrow{\lambda x.e} p) \triangleq \xrightarrow{\lambda x.e} \varphi(p)
\end{array}$$

From now on, we shall write $\mathsf{Path}$ for the set of access paths that use timestamps in $\mathbb{T}$, and $\mathsf{Path}'$ for the set of access paths that use timestamps in $\mathbb{T}'$. Then given an access path, we can define a predicate $\mathsf{valid} \in (\mathsf{Ctx} \uplus \mathbb{T}) \times \mathsf{Mem} \times \mathsf{Path} \to \mathsf{Prop}$. $\mathsf{valid}(r, m, p)$ is true iff starting from $r$, all

accesses edges in $p$ are valid. Likewise, we can define a predicate $^\circ\mathsf{valid} \in (^\circ\mathsf{Ctx} \uplus {}^\circ\mathbb{T}) \times {}^\circ\mathsf{Mem} \times {}^\circ\mathsf{Path} \to \mathsf{Prop}$. $^\circ\mathsf{valid}(^\circ r, {}^\circ m, {}^\circ p)$ is true iff starting from $^\circ r$, all access edges in $^\circ p$ are valid.

$$\begin{array}{ll}
\mathsf{valid}(_, m, \epsilon) \triangleq \mathsf{True} & {}^\circ\mathsf{valid}(_, {}^\circ m, \epsilon) \triangleq \mathsf{True} \\
\mathsf{valid}(C, m, \xrightarrow{x} t\ p) \triangleq t = C(x) \wedge \mathsf{valid}(t, m, p) & {}^\circ\mathsf{valid}({}^\circ C, {}^\circ m, \xrightarrow{x} {}^\circ t\ {}^\circ p) \triangleq {}^\circ t = {}^\circ C(x) \wedge {}^\circ\mathsf{valid}({}^\circ t, {}^\circ m, {}^\circ p) \\
\mathsf{valid}(C, m, \xrightarrow{M} p) \triangleq \mathsf{valid}(C(M), m, p) & {}^\circ\mathsf{valid}({}^\circ C, {}^\circ m, \xrightarrow{M} {}^\circ p) \triangleq {}^\circ\mathsf{valid}({}^\circ C(M), {}^\circ m, {}^\circ p) \\
\mathsf{valid}(t, m, \xrightarrow{\lambda x.e} p) \triangleq \langle \lambda x.e, {}^\exists C\rangle = m(t) \wedge \mathsf{valid}(C, m, p) & {}^\circ\mathsf{valid}({}^\circ t, {}^\circ m, \xrightarrow{\lambda x.e} {}^\circ p) \triangleq \langle \lambda x.e, {}^\exists {}^\circ C\rangle \in {}^\circ m({}^\circ t) \wedge {}^\circ\mathsf{valid}({}^\circ C, {}^\circ m, {}^\circ p) \\
\mathsf{valid}(\mathsf{otherwise}) \triangleq \mathsf{False} & {}^\circ\mathsf{valid}(\mathsf{otherwise}) \triangleq \mathsf{False}
\end{array}$$

Fig. 8. Definitions for the valid and $^\circ$valid predicates.

Now we can give straightforward definitions of equivalence.

**Definition 6.1** (Equivalent Concrete States)**.** Let $s = (C, m, \_) \in \mathsf{State}$ and $s' = (C', m', \_) \in \mathsf{State}'$. We say $s$ is *equivalent* to $s'$ and write $s \sim s'$ when there exists a $\varphi : \mathbb{T} \to \mathbb{T}'$ and $\varphi' : \mathbb{T}' \to \mathbb{T}$ such that:

(1) $\forall p \in \mathsf{Path}$ : if $\mathsf{valid}(C, m, p)$ then $\mathsf{valid}(C', m', \varphi(p))$ and $p = \varphi'(\varphi(p))$.
(2) $\forall p' \in \mathsf{Path}'$ : if $\mathsf{valid}(C', m', p')$ then $\mathsf{valid}(C, m, \varphi'(p'))$ and $p' = \varphi(\varphi'(p'))$.

**Definition 6.2** (Weakly Equivalent Abstract States)**.** Let $^\circ s = (^\circ C, {}^\circ m, \_) \in {}^\circ\mathsf{State}$ and $^\circ s' = (^\circ C', {}^\circ m', \_) \in {}^\circ\mathsf{State}'$. We say $^\circ s$ is *weakly equivalent* to $^\circ s'$ when there exists a $^\circ\varphi : {}^\circ\mathbb{T} \to {}^\circ\mathbb{T}'$ and $^\circ\varphi' : {}^\circ\mathbb{T}' \to {}^\circ\mathbb{T}$ such that:

(1) $\forall\, ^\circ p \in {}^\circ\mathsf{Path}$ : if $^\circ\mathsf{valid}(^\circ C, {}^\circ m, {}^\circ p)$ then $^\circ\mathsf{valid}(^\circ C', {}^\circ m', {}^\circ\varphi(^\circ p))$ and $^\circ p = {}^\circ\varphi'({}^\circ\varphi(^\circ p))$.
(2) $\forall\, ^\circ p' \in {}^\circ\mathsf{Path}'$ : if $^\circ\mathsf{valid}(^\circ C', {}^\circ m', {}^\circ p')$ then $\mathsf{valid}(^\circ C, {}^\circ m, {}^\circ\varphi'(^\circ p'))$ and $^\circ p' = {}^\circ\varphi(^\circ\varphi'(^\circ p'))$.

The reason that the above definition is called "weak equivalence" is because it is not sufficient to guarantee equivalence after concretization. Consider

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

and

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

They are weakly equivalent, yet their concretizations are not equivalent. Thus, we need to strengthen the definition for abstract equivalence.

Before going into the definition, we introduce some terminology. First, we say that two states are *weakly equivalent by* $^\circ\varphi, {}^\circ\varphi'$ when $^\circ\varphi, {}^\circ\varphi'$ are the functions that translate between abstract timestamps in Definition 6.2. Second, we say that $^\circ t$ is *reachable from* $^\circ s$ when there is some valid access path $^\circ p$ from $^\circ s$ containing $^\circ t$. Now we actually give the definition:

**Definition 6.3** (Equivalent Abstract States)**.** Let $^\circ s = (\_, {}^\circ m, \_) \in {}^\circ\mathsf{State}$ and $^\circ s' = (\_, {}^\circ m', \_) \in {}^\circ\mathsf{State}'$. We say $^\circ s$ is *equivalent* to $^\circ s'$ and write $^\circ s\, {}^\circ\!\sim {}^\circ s'$ when there exists a $^\circ\varphi : {}^\circ\mathbb{T} \to {}^\circ\mathbb{T}'$ and $^\circ\varphi' : {}^\circ\mathbb{T}' \to {}^\circ\mathbb{T}$ such that:

(1) $^\circ s$ and $^\circ s'$ are weakly equivalent by $^\circ\varphi, {}^\circ\varphi'$.
(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 $^\circ\varphi, {}^\circ\varphi'$ under the empty memory.
(3) The same holds for each $^\circ t'$ reachable from $^\circ s'$.

We extend the definition of equivalence between elements of $\Sigma$ and $\Sigma'$:

$$(\langle \lambda x.e, C\rangle, m, t) \sim (\langle \lambda x.e, C'\rangle, m', t') \triangleq (C, m, t) \sim (C', m', t')$$
$$(e, s) \sim (e, s') \triangleq s \sim s'$$
$$(\langle \lambda x.e, {}^\circ C\rangle, {}^\circ m, {}^\circ t) \, {}^\circ\!\sim (\langle \lambda x.e, {}^\circ C'\rangle, {}^\circ m', {}^\circ t') \triangleq ({}^\circ C, {}^\circ m, {}^\circ t) \, {}^\circ\!\sim ({}^\circ C', {}^\circ m', {}^\circ t')$$
$$(e, {}^\circ s) \, {}^\circ\!\sim (e, {}^\circ s') \triangleq {}^\circ s \, {}^\circ\!\sim {}^\circ s$$

Then we can extend the definition of equivalence between the elements of $D$ and $D'$:

$$A \sim A' \triangleq (\forall \ell \rightsquigarrow \rho, \rho \in A : \exists \ell' \rightsquigarrow \rho', \rho' \in A' : \ell \sim \ell' \wedge \rho \sim \rho') \wedge$$
$$(\forall \ell' \rightsquigarrow \rho', \rho' \in A' : \exists \ell \rightsquigarrow \rho, \rho \in A : \ell \sim \ell' \wedge \rho \sim \rho')$$
$$A^\# \sim^\# A'^\# \triangleq (\forall {}^\circ\ell \, {}^\circ\!\rightsquigarrow {}^\circ\rho, {}^\circ\rho \in A^\# : \exists \, {}^\circ\ell' \, {}^\circ\!\rightsquigarrow {}^\circ\rho', {}^\circ\rho' \in A'^\# : {}^\circ\ell \, {}^\circ\!\sim {}^\circ\ell' \wedge {}^\circ\rho \, {}^\circ\!\sim {}^\circ\rho') \wedge$$
$$(\forall {}^\circ\ell' \, {}^\circ\!\rightsquigarrow {}^\circ\rho', {}^\circ\rho' \in A'^\# : \exists \, {}^\circ\ell \, {}^\circ\!\rightsquigarrow {}^\circ\rho, {}^\circ\rho \in A^\# : {}^\circ\ell \, {}^\circ\!\sim {}^\circ\ell' \wedge {}^\circ\rho \, {}^\circ\!\sim {}^\circ\rho')$$

### 6.3 Propeties of Equivalence

We first note that the relations $\sim$ and ${}^\circ\!\sim$ are actually equivalence relations. That is, they are reflexive, transitive, and commutative. We must also show that equivalence is well-behaved under the step relation and concretization. That is, we must show that concretizing equivalent abstract states lead to equivalent states, and that equivalence preserves the step relation.

**Lemma 6.1** (Concretization Preserves Equivalence). *Assume that each ${}^\circ t, {}^\circ t'$ in ${}^\circ\mathbb{T}, {}^\circ\mathbb{T}'$ corresponds to an infinite set of concrete timestamps. Then for all $S^\# \subseteq {}^\circ\mathsf{State}$ and $S'^\# \subseteq {}^\circ\mathsf{State}'$,*

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

Sketch. We want to prove:

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

If this is true, $\forall s \in \gamma(S^\#) : \exists s' \in \gamma(S'^\#) : s \sim s'$. Similarly, we have $\forall s' \in \gamma(S'^\#) : \exists s \in \gamma(S^\#) : s \sim s'$, so that $\gamma(S^\#) \sim \gamma'(S'^\#)$.

This is proven in Coq (`ConcretEquivalence.v`). □

**Lemma 6.2** (Evaluation Preserves Equivalence). *For all $\ell \in \mathsf{Left}$, $\rho \in \mathsf{Right}$, $\ell' \in \mathsf{Left}'$,*

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

*Thus, if $S \subseteq \mathsf{State}$ and $S' \subseteq \mathsf{State}'$ are equivalent, $[\![e]\!]S \sim [\![e]\!]S'$.*

Sketch. This is proven in Coq (`OperationalEquivalence.v`). □

Note that there is a caveat in Lemma 6.1. We have required that all partitions ${}^\circ\alpha^{-1}({}^\circ t)$ of $\mathbb{T}$ to be infinite. This is natural, since if an abstract address that concretizes to a finite set corresponds to an abstract address that concretizes to an infinite set, the concretization might no longer be equivalent. This constraint is not as restrictive as it seems, as widely used abstractions such as $k$-CFA already satisfy this criterion.

### 6.4 How to Utilize Equivalence

Here is a general outline that utilize abstract equivalence and abstract linking to overapproximate any initial state. The goal is to overapproximate something equivalent to $[\![e]\!]\gamma(S^\#)$, when all abstract timestamps in $S^\#$ correspond to infinitely many concrete timestamps.

**Step 1** Choose a finite set ${}^\circ\mathbb{T}_2$ and a function ${}^\circ\mathsf{tick}_2 \in {}^\circ\mathbb{T}_2 \to {}^\circ\mathbb{T}_2$.

**Step 2** Assume an initial condition $S_2^\#$ and compute $[\![e]\!]^\# S_2^\#$.

**Step 3** Choose a finite set $^\circ\mathbb{T}_1$ and $^\circ\mathsf{tick}_1 \in {}^\circ\mathbb{T}_1 \to {}^\circ\mathbb{T}_1$.

**Step 4** Find a $S_1^\#$ such that $S_1^\# \triangleright^\# S_2^\#$ is equivalent to some *superset* $\overline{S^\#}$ of $S^\#$.

**Result** Then $S_1^\# \infty^\# [\![e]\!]^\# S_2^\#$ overapproximates an equivalent superset of $[\![e]\!]\gamma(S^\#)$.

$S_1^\# \infty^\# [\![e]\!]^\# S_2^\#$ overapproximates an equivalent superset of $[\![e]\!]\gamma(S^\#)$, since if we let:

$$\mathbb{T}_+ \triangleq ({}^\circ\mathbb{T}_1 + {}^\circ\mathbb{T}_2) \times \mathbb{Z} \quad \mathsf{tick}_+({}^\circ t, n) \triangleq ({}^\circ\mathsf{tick}_+({}^\circ t), n+1) \quad {}^\circ\alpha_+({}^\circ t, n) \triangleq {}^\circ t$$

we have a concrete time $\mathbb{T}_+$ that is connected to $^\circ\mathbb{T}_1 + {}^\circ\mathbb{T}_2$ by $^\circ\alpha_+$ such that all abstract timestamps correspond to infinitely many concrete timestamps. Thus:

$$\begin{aligned}
[\![e]\!]\gamma(S^\#) &\subseteq [\![e]\!]\gamma(\overline{S^\#}) && (\because S^\# \subseteq \overline{S^\#} \text{ and } \gamma \text{ monotonic}) \\
&\sim [\![e]\!]\gamma_+(S_1^\# \triangleright^\# S_2^\#) && (\because \text{Concretization preserves equivalence}) \\
&\subseteq \gamma_+([\![e]\!]^\#(S_1^\# \triangleright^\# S_2^\#)) && (\because \text{Soundness}) \\
&= \gamma_+(S_1^\# \infty^\# [\![e]\!]^\# S_2^\#) && (\because \text{Abstract advance})
\end{aligned}$$

## REFERENCES