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
(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 an extension of untyped lambda calculus with modules and the linking construct.

$$
\begin{array}{rrcll}
\text{Identifiers} & x, d & \in & \mathsf{Var} & \\
\text{Expression} & e & \to & x \mid \lambda x.e \mid e\ e & \text{untyped } \lambda\text{-calculus} \\
 & & \mid & e \rtimes e & \text{linked expression} \\
 & & \mid & \varepsilon & \text{empty module} \\
 & & \mid & d & \text{module identifier} \\
 & & \mid & \mathtt{let}\ x\ e\ e & \text{expression binding} \\
 & & \mid & \mathtt{let}\ d\ e\ e & \text{module binding}
\end{array}
$$

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

The language is expressive enough to encode simple imports and exports that use module names as interfaces.

$$
\begin{aligned}
e_1 &\triangleq \mathtt{let\ x\ =\ 1\ in}\ \varepsilon \\
e_2 &\triangleq \mathtt{let\ fact\ =\ fix}\ \lambda\mathtt{fact}.\lambda\mathtt{n.if0\ n\ 1\ (*\ n\ (fact\ (-\ n\ 1)))\ in}\ \varepsilon \\
e &\triangleq (\mathtt{let\ M\ =}\ e_1\ \mathtt{in}\ \varepsilon) \rtimes (\mathtt{let\ F\ =}\ e_2\ \mathtt{in}\ \varepsilon) \rtimes (\mathtt{+\ (F} \rtimes \mathtt{fact\ 100)\ (M} \rtimes \mathtt{x))}
\end{aligned}
$$

Above is how the example in the introduction is translated into the simple module language in Figure 1, assuming that the definitions for arithmetic and the fixpoint combinator `fix` are given.

2.1 Operational Semantics

$$
\begin{array}{rrcll}
\text{Environment/Context} & \sigma & \in & \mathsf{Ctx} & \\
\text{Value of expressions} & v & \in & \mathsf{Val} \triangleq \mathsf{Var} \times \mathsf{Expr} \times \mathsf{Ctx} & \\
\text{Value of expressions/modules} & V & \in & \mathsf{Val} + \mathsf{Ctx} & \\
\text{Configuration (left)} & c & \in & \mathsf{Config} \triangleq \mathsf{Expr} \times \mathsf{Ctx} & \\
\text{Configuration (right)} & r & \in & \mathsf{Right} \triangleq \mathsf{Config} + \mathsf{Val} + \mathsf{Ctx} & \\
\text{Context} & \sigma & \to & [] & \text{empty stack} \\
 & & \mid & (x, v) :: \sigma & \text{expression binding} \\
 & & \mid & (d, \sigma) :: \sigma & \text{module binding} \\
\text{Value of expressions} & v & \to & \langle \lambda x.e, \sigma \rangle & \text{closure}
\end{array}
$$

Fig. 2. Definition of the semantic domains.

We present the operational semantics $\hookrightarrow$ 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 c of Config with an element r of Right. Note that $\sigma(x)$ pops the highest value that is associated with x from the stack σ and $\sigma(d)$ pops the highest context

$$\boxed{(e,\sigma) \hookrightarrow V \text{ or } (e',\sigma')}$$

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

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

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

$$[\textsc{Empty}]\ \frac{}{(\varepsilon,\sigma)\hookrightarrow\sigma} \qquad [\textsc{ModID}]\ \frac{\sigma'=\sigma(d)}{(d,\sigma)\hookrightarrow\sigma'}$$

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

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

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

associated with d from σ. The relation $\hookrightarrow$ is unorthodox in that unlike normal big-step operaional semantics, it 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.

2.2 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 Trace to Trace, when:

$$\Sigma \triangleq \mathsf{Right}{+} \hookrightarrow \qquad \mathsf{Trace} \triangleq \mathcal{P}(\Sigma)$$

The semantics of an expression e starting from initial states in $C \subseteq \mathsf{Ctx}$ is the collection of $c \hookrightarrow r$ and r derivable from initial configurations (e,σ) with $\sigma \in C$. Defining the transfer function is straightforward from the definition of the transition relation.

Definition 2.1 (Transfer function). Given $A \subseteq \Sigma$, define

$$\mathsf{Step}(A) \triangleq \left\{c \hookrightarrow r, r \middle| \frac{A'}{c \hookrightarrow r} \text{ and } A' \subseteq A \text{ and } c \in A\right\}$$

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 $C \subseteq \mathsf{Ctx}$, define:

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

Note that the above definition can be defined even when the σ in (e, σ) does not close e. Then the collecting semantics will store the proof tree only up to the point the first free variable is evaluated.

2.3 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 σ_1 and (2) some external context σ_2, gives the meaning of e under the *linked* context of σ_1 and σ_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 $r_2 \in \mathsf{Right}$ with a $\sigma_1 \in \mathsf{Ctx}$:

$$r_2\langle\sigma_1\rangle \triangleq \begin{cases} \sigma_1 & r_2 = [] \\ (x, v\langle\sigma_1\rangle) :: \sigma\langle\sigma_1\rangle & r_2 = (x, v) :: \sigma \\ (d, \sigma\langle\sigma_1\rangle) :: \sigma'\langle\sigma_1\rangle & r_2 = (d, \sigma) :: \sigma' \\ \langle\lambda x.e, \sigma\langle\sigma_1\rangle\rangle & r_2 = \langle\lambda x.e, \sigma\rangle \\ (e, \sigma\langle\sigma_1\rangle) & r_2 = (e, \sigma) \end{cases}$$

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

Lemma 2.3 (Fill in the Blanks). *For all $\sigma_1, \sigma_2 \in \mathsf{Ctx}$, for each expression variable x,*

$$\sigma_2(x) = v \Rightarrow \sigma_2\langle\sigma_1\rangle(x) = v\langle\sigma_1\rangle \text{ and } \sigma_2(x) = \bot \Rightarrow \sigma_2\langle\sigma_1\rangle(x) = \sigma_1(x)$$

and for each module variable d,

$$\sigma_2(d) = \sigma \Rightarrow \sigma_2\langle\sigma_1\rangle(d) = \sigma\langle\sigma_1\rangle \text{ and } \sigma_2(d) = \bot \Rightarrow \sigma_2\langle\sigma_1\rangle(d) = \sigma_1(d)$$

Sketch. Induction on σ_2. □

Moreover, filling in the blanks preserves the evaluation relation $\hookrightarrow$.

Lemma 2.4 (Injection Preserves Evaluation). *For all $c \in \mathsf{Config}$, $r \in \mathsf{Right}$, $c \hookrightarrow r \Rightarrow c\langle\sigma\rangle \hookrightarrow r\langle\sigma\rangle$.*

Sketch. Induction on $\hookrightarrow$. □

Thus, we can define $\triangleright$ that injects a *set* of contexts C into an subset A of Σ and a semantic linking operation ∞ that does the rest of the computation:

Definition 2.5 (Injection). For $C \subseteq \mathsf{Ctx}$ and $A \subseteq \Sigma$, define:

$$C \triangleright A \triangleq \{r\langle\sigma\rangle | \sigma \in C, r \in A\} \cup \{c\langle\sigma\rangle \hookrightarrow r\langle\sigma\rangle | \sigma \in C, c \hookrightarrow r \in A\}$$

Definition 2.6 (Semantic Linking). For $C \subseteq \mathsf{Ctx}$ and $A \subseteq \Sigma$, define:

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

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

Theorem 2.7 (Advance). *For all $e \in \mathsf{Expr}$ and $C_1, C_2 \subseteq \mathsf{Ctx}$,*

$$[\![e]\!](C_1 \triangleright C_2) = C_1 \infty [\![e]\!]C_2$$

Proof. Let A be $\{(e, \sigma) | \sigma \in C_1 \triangleright C_2\}$, and let B be $C_1 \triangleright [\![e]\!]C_2$. Note that $A \subseteq B$ by the definition of $[\![e]\!]C_2$. Also, let X_A be $\mathsf{lfp}(\lambda X.\mathsf{Step}(X) \cup A) = [\![e]\!](C_1 \triangleright C_2)$ and let X_B be $\mathsf{lfp}(\lambda X.\mathsf{Step}(X) \cup B) = C_1 \infty [\![e]\!]C_2$. 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:

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

Trace 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 $X_A = X_B$. □

2.4 Skeleton of a Static Analysis

We require a CPO $\mathsf{Trace}^\#$ that is Galois connected with Trace:

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

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 $\propto^\#$ as:

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

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

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

Then we can soundly approximate fractional specifications by:

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

3 INSTRUMENTED SEMANTICS

All that is left is to present an abstraction for the semantics in the previous section. We need to abstract $C \subseteq \mathsf{Ctx}$ to finitely compute an overapproximation. However, devising such an abstraction is not immediately obvious.

The problem is that closures bound in $\sigma \in \mathsf{Ctx}$ again contain contexts. 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

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 Outcome.

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

$$
\begin{array}{rccll}
\text{Time} & t & \in & \mathbb{T} & \\
\text{Environment/Context} & \sigma & \in & \mathsf{Ctx} & \\
\text{Value of expressions} & v & \in & \mathsf{Val} \triangleq \mathsf{Var} \times \mathsf{Expr} \times \mathsf{Ctx} & \\
\text{Value of expressions/modules} & V & \in & \mathsf{Val} + \mathsf{Ctx} & \\
\text{Memory} & m & \in & \mathsf{Mem} \triangleq \mathbb{T} \xrightarrow{\text{fin}} \mathsf{Val} & \\
\text{State} & s & \in & \mathsf{State} \triangleq \mathsf{Ctx} \times \mathsf{Mem} \times \mathbb{T} & \\
\text{Outcome} & o & \in & \mathsf{Outcome} \triangleq (\mathsf{Val} + \mathsf{Ctx}) \times \mathsf{Mem} \times \mathbb{T} & \\
\text{Configuration (left)} & c & \in & \mathsf{Config} \triangleq \mathsf{Expr} \times \mathsf{State} & \\
\text{Configuration (right)} & r & \in & \mathsf{Right} \triangleq \mathsf{Config} + \mathsf{Outcome} & \\
\text{Context} & \sigma & \to & [] & \text{empty stack} \\
 & & \mid & (x, t) :: \sigma & \text{expression binding} \\
 & & \mid & (d, \sigma) :: \sigma & \text{module binding} \\
\text{Value of expressions} & v & \to & \langle \lambda x.e, \sigma \rangle & \text{closure}
\end{array}
$$

Fig. 4. Definition of the instrumented semantic domains.

$$\boxed{(e, \sigma, m, t) \hookrightarrow (V, m', t') \text{ or } (e', \sigma', m', t')}$$

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

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

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

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

$$
[\textsc{LetM}]\ \frac{\begin{array}{c}(e_1, \sigma, m, t) \hookrightarrow (\sigma', m', t') \\ (e_2, (d, \sigma') :: \sigma, m', t') \hookrightarrow (\sigma'', m'', t'')\end{array}}{(\mathtt{let}\ d\ e_1\ e_2, \sigma, m, t) \hookrightarrow (\sigma'', m'', t'')}
$$

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

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 $\hookrightarrow$ as it is. There are no restrictions on tick and the states (σ, m, t), thus a write to the address $\mathsf{tick}(t)$ may overwrite an existing value that may be used for future computations. Thus, $\mathsf{tick}(t) \notin \mathsf{supp}(\sigma, m)$ must be guaranteed, when $\mathsf{supp}(\sigma, m)$ is the set of timestamps reachable from (σ, m). To enforce this invariant upon all *valid* concrete executions defined by the relation $\hookrightarrow$, we enforce that there be a *total order* on $\mathbb{T}$. Then our criteria can be guaranteed by first enforcing that $\sigma \leq t$ and $m \leq t$, where $\sigma \leq t$ means that all timestamps in σ 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:

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

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:

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

as the set of valid states that enable tick to generate fresh timestamps. It is almost trivial that the set $\mathsf{Config} \times \mathsf{Right}$ is closed under the inductive definition of $\hookrightarrow$. That is,

Lemma 3.1 (Valid States Transition to Valid States). *For all $c \in \mathsf{Config}$, if $c \hookrightarrow r$ according to the inductive rules, $r \in \mathsf{Right}$.*

Sketch. Induction on $\hookrightarrow$. □

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 write:

$$\Sigma \triangleq \mathsf{Right}+ \hookrightarrow \qquad \mathsf{Trace} \triangleq \mathcal{P}(\Sigma)$$

Definition 3.2 (Transfer function). Given $A \subseteq \Sigma$, define

$$\mathsf{Step}(A) \triangleq \left\{c \hookrightarrow r, r \middle| \frac{A'}{c \hookrightarrow r} \text{ and } A' \subseteq A \text{ and } c \in A\right\}$$

and

Definition 3.3 (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

$$\begin{array}{rcll}
\text{Abstract Time} & \dot{t} & \in & \dot{\mathbb{T}} \\
\text{Environment/Context} & \dot{\sigma} & \in & \mathsf{C\dot{t}x} \\
\text{Value of expressions} & \dot{v} & \in & \mathsf{\dot{V}al} \subseteq \mathsf{Var} \times \mathsf{Expr} \times \mathsf{C\dot{t}x} \\
\text{Value of expressions/modules} & \dot{V} & \in & \mathsf{\dot{V}al} + \mathsf{C\dot{t}x} \\
\text{Abstract Memory} & \dot{m} & \in & \mathsf{M\dot{e}m} \triangleq \dot{\mathbb{T}} \xrightarrow{\text{fin}} \mathcal{P}(\mathsf{\dot{V}al}) \\
\text{Abstract State} & \dot{s} & \in & \mathsf{St\dot{a}te} \triangleq \mathsf{C\dot{t}x} \times \mathsf{M\dot{e}m} \times \dot{\mathbb{T}} \\
\text{Abstract outcome} & \dot{o} & \in & \mathsf{Outc\dot{o}me} \triangleq (\mathsf{\dot{V}al} + \mathsf{C\dot{t}x}) \times \mathsf{M\dot{e}m} \times \dot{\mathbb{T}} \\
\text{Abstract configuration (left)} & \dot{c} & \in & \mathsf{Con\dot{f}ig} \triangleq \mathsf{Expr} \times \mathsf{St\dot{a}te} \\
\text{Abstract configuration (right)} & \dot{r} & \in & \mathsf{R\dot{i}ght} \triangleq \mathsf{Con\dot{f}ig} + \mathsf{Outc\dot{o}me} \\
\text{Context} & \dot{\sigma} & \rightarrow & [] \qquad\qquad\qquad\qquad \text{empty stack} \\
 & & | & (x, \dot{t}) :: \dot{\sigma} \qquad\qquad\quad \text{expression binding} \\
 & & | & (d, \dot{\sigma}) :: \dot{\sigma} \qquad\qquad\quad \text{module binding} \\
\text{Value of expressions} & \dot{v} & \rightarrow & \langle \lambda x.e, \dot{\sigma} \rangle \qquad\qquad\quad \text{closure}
\end{array}$$

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 $\dot{\mathbb{T}}$ that is connected to the concrete time domain via an auxiliary function $\dot{\alpha} : \mathbb{T} \rightarrow \dot{\mathbb{T}}$. Since the policy to update the timestamp must also be compatible with respect to $\dot{\alpha}$, we require the $\mathsf{ti\dot{c}k} : \dot{\mathbb{T}} \rightarrow \dot{\mathbb{T}}$ function to satisfy $\dot{\alpha} \circ \mathsf{tick} = \mathsf{ti\dot{c}k} \circ \dot{\alpha}$.

$$\boxed{(e, \dot{\sigma}, \dot{m}, \dot{t}) \mathrel{\dot{\hookrightarrow}} (\dot{V}, \dot{m}', \dot{t}') \text{ or } (e', \dot{\sigma}', \dot{m}', \dot{t}')}$$

$$\text{[ExprID]}\ \frac{\dot{t}_x = \dot{\sigma}(x) \qquad \dot{v} \in \dot{m}(\dot{t}_x)}{(x, \dot{\sigma}, \dot{m}, \dot{t}) \mathrel{\dot{\hookrightarrow}} (\dot{v}, \dot{m}, \dot{t})}$$

$$\text{[App]}\ \frac{\begin{array}{c}(e_1, \dot{\sigma}, \dot{m}, \dot{t}) \mathrel{\dot{\hookrightarrow}} (\langle \lambda x.e_\lambda, \dot{\sigma}_\lambda \rangle, \dot{m}_\lambda, \dot{t}_\lambda) \\ (e_2, \dot{\sigma}, \dot{m}_\lambda, \dot{t}_\lambda) \mathrel{\dot{\hookrightarrow}} (\dot{v}, \dot{m}_a, \dot{t}_a) \\ (e_\lambda, (x, \dot{\mathsf{tick}}(\dot{t}_a)) :: \dot{\sigma}_\lambda, \dot{m}_a[\dot{\mathsf{tick}}(\dot{t}_a) \mapsto \dot{v}], \dot{\mathsf{tick}}(\dot{t}_a)) \mathrel{\dot{\hookrightarrow}} (\dot{v}', \dot{m}', \dot{t}')\end{array}}{(e_1\ e_2, \dot{\sigma}, \dot{m}, \dot{t}) \mathrel{\dot{\hookrightarrow}} (\dot{v}', \dot{m}', \dot{t}')}$$

$$\text{[LetE]}\ \frac{\begin{array}{c}(e_1, \dot{\sigma}, \dot{m}, \dot{t}) \mathrel{\dot{\hookrightarrow}} (\dot{v}, \dot{m}', \dot{t}') \\ (e_2, (x, \dot{\mathsf{tick}}(\dot{t}')) :: \dot{\sigma}, \dot{m}'[\dot{\mathsf{tick}}(\dot{t}') \mapsto \dot{v}], \dot{\mathsf{tick}}(\dot{t}')) \mathrel{\dot{\hookrightarrow}} (\dot{\sigma}', \dot{m}'', \dot{t}'')\end{array}}{(\mathtt{let}\ x\ e_1\ e_2, \dot{\sigma}, \dot{m}, \dot{t}) \mathrel{\dot{\hookrightarrow}} (\dot{\sigma}', \dot{m}'', \dot{t}'')}$$

Fig. 7. Excerpt of the abstract operational semantics, corresponding to the big-step evaluation rules that differ from the concrete version.

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 now a weak update:

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

and a 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 $\dot{\hookrightarrow} \subseteq \mathsf{Config} \times \dot{\mathsf{Right}}$ is in Figure 7.

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

$$\begin{aligned} \dot{\alpha}([]) &\triangleq [] \\ \dot{\alpha}((x, t_x) :: \sigma) &\triangleq (x, \dot{\alpha}(t_x)) :: \dot{\alpha}(\sigma) \\ \dot{\alpha}((d, \sigma_d) :: \sigma) &\triangleq (d, \dot{\alpha}(\sigma_d)) :: \dot{\alpha}(\sigma) \\ \dot{\alpha}(\langle \lambda x.e, \sigma \rangle) &\triangleq \langle \lambda x.e, \dot{\alpha}(\sigma) \rangle \\ \dot{\alpha}(m) &\triangleq \lambda \dot{t}.\{\dot{\alpha}(m(t)) | \dot{\alpha}(t) = \dot{t}\} \\ \dot{\alpha}(e, \sigma, m, t) &\triangleq (e, \dot{\alpha}(\sigma), \dot{\alpha}(m), \dot{\alpha}(t)) \\ \dot{\alpha}(V, m, t) &\triangleq (\dot{\alpha}(V), \dot{\alpha}(m), \dot{\alpha}(t)) \end{aligned}$$

We have:

Lemma 4.1 (Operational Soundness). *For all $c \in \mathsf{Config}$ and $r \in \mathsf{Right}$, if $c \hookrightarrow r$ then $\dot{\alpha}(c) \mathrel{\dot{\hookrightarrow}} \dot{\alpha}(r)$.*

Sketch. Induction on $\hookrightarrow$. □

Then if we define:

$$\dot{\Sigma} \triangleq \dot{\mathsf{Right}} + \dot{\hookrightarrow} \qquad \mathsf{Trace}^{\#} \triangleq \mathcal{P}(\dot{\Sigma})$$

we can establish a Galois connection between Trace and $\mathsf{Trace}^{\#}$. The abstraction and concretization functions are given by:

Definition 4.2 (Abstraction and Concretization). Define α : Trace $\to$ Trace$^\#$ and γ : Trace$^\#$ $\to$ Trace by:

$$\alpha(A) \triangleq \{\dot{\alpha}(c) \mathrel{\dot{\hookrightarrow}} \dot{\alpha}(r) | c \hookrightarrow r \in A\} \cup \{\dot{\alpha}(r) | r \in A\}$$
$$\gamma(A^\#) \triangleq \{c \hookrightarrow r | \dot{\alpha}(c) \mathrel{\dot{\hookrightarrow}} \dot{\alpha}(r) \in A^\#\} \cup \{r | \dot{\alpha}(r) \in A^\#\}$$

Then it is straightforward to see that:

Lemma 4.3 (Galois Connection). Trace $= \mathcal{P}(\Sigma) \xrightleftharpoons[\alpha]{\gamma}$ Trace$^\# = \mathcal{P}(\dot{\Sigma})$. *That is:*

$$\forall A \in \mathsf{Trace}, A^\# \in \mathsf{Trace}^\# : \alpha(A) \subseteq A^\# \Leftrightarrow A \subseteq \gamma(A^\#)$$

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

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

Definition 4.4 (Abstract transfer function). Given $A^\# \subseteq \dot{\Sigma}$, define:

$$\mathsf{Step}^\#(A^\#) \triangleq \left\{\dot{c} \mathrel{\dot{\hookrightarrow}} \dot{r}, \dot{r} \middle| \frac{A'^\#}{\dot{c} \mathrel{\dot{\hookrightarrow}} \dot{r}} \text{ and } A'^\# \subseteq A^\# \text{ and } \dot{c} \in A^\#\right\}$$

Definition 4.5 (Abstract semantics). Given $e \in$ Expr and $S^\# \subseteq \dot{\mathsf{State}}$, define:

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

Then we can prove that:

Theorem 4.6 (Soundness). *For all* $e \in$ Expr, $[\![e]\!] \circ \gamma \subseteq \gamma \circ [\![e]\!]^\#$.

Proof. By Lemma 4.1, 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 4.7 (Finiteness). *For all* $e \in$ Expr *and* $S^\# \subseteq \dot{\mathsf{State}}$, *if* $S^\#$ *is finite,* $[\![e]\!]^\# S^\#$ *is finite.*

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

$$\forall \dot{r} \in \dot{\mathsf{Right}} : (e, \dot{s}) \mathrel{\dot{\hookrightarrow}}^* \dot{r} \Rightarrow \dot{r} \in X$$

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

5 LINKING IN THE INSTRUMENTED SEMANTICS

Now we need to define an injection operation that fills in the blanks of a $o \overset{\text{let}}{=} (V, m, t) \in$ Outcome with a $s \overset{\text{let}}{=} (\sigma', m', t') \in$ State. Recall the definition for injection in the semantics without memory. $V\langle\sigma\rangle$ enables access to values that were previously not available in V by filling in the bottom of the stack with σ. 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 (σ, m) does not overlap with reachable addresses in (σ', m'). Then again, this requirement may be lifted if we allow linking of semantics that use *different* time domains as addresses. Note that we can only *read* values from σ in $V\langle\sigma\rangle$; we should 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 $\mathbb{T}_1$ and r_2 uses $\mathbb{T}_2$. Then $r_2\langle s_1\rangle$ must live in a version of Outcome that uses $\mathbb{T}_1+\mathbb{T}_2$. 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, $\dot{\mathbb{T}}$, $\dot{\alpha}$, and $\dot{\mathsf{tick}}$ also be linked. Concretely, we demand that the linked tick_+ preserves the condition that $\dot{\alpha}_+\circ\mathsf{tick}_+=\dot{\mathsf{tick}}_+\circ\dot{\alpha}_+$. Also, we need to link tick well so that for all valid transitions $c_2\hookrightarrow r_2$ under tick_2, $c_2\langle s_1\rangle\hookrightarrow r_2\langle s_1\rangle$ is also a valid transition under tick_+. 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 tick_+, $\dot{\alpha}_+$, $\dot{\mathsf{tick}}$

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

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

it is easy to check that all requirements are satisfied.

5.2 Concrete Linking

Now we define injection between $s_1=(\sigma_1,m_1,t_1)\in\mathsf{State}_1$ and $o_2=(V_2,m_2,t_2)\in\mathsf{Outcome}_2$:

$$V_2\langle\sigma_1\rangle\triangleq\begin{cases}\sigma_1 & V_2=[]\\ (x,t)::\sigma\langle\sigma_1\rangle & V_2=(x,t)::\sigma\\ (d,\sigma\langle\sigma_1\rangle)::\sigma'\langle\sigma_1\rangle & V_2=(d,\sigma)::\sigma'\\ \langle\lambda x.e,\sigma_2\langle\sigma_1\rangle\rangle & V_2=\langle\lambda x.e,\sigma_2\rangle\end{cases}\qquad \begin{aligned}&m_2\langle\sigma_1\rangle\triangleq\bigcup_{t\in\mathsf{dom}(m_2)}\{t\mapsto m_2(t)\langle\sigma_1\rangle\}\\ &o_2\langle s_1\rangle\triangleq(V_2\langle\sigma_1\rangle,m_1\cup m_2\langle\sigma_1\rangle,t_2)\end{aligned}$$

As is expected, injecting s_1 into o_2 involves injecting σ_1 in every context in o_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, $o_2\langle s_1\rangle\in\mathsf{Outcome}_+$ with respect to the ordering $\leq_+$. Also, if we define $(e,s_2)\langle s_1\rangle\triangleq(e,s_2\langle s_1\rangle)$, we can show that injection preserves valid transitions.

Lemma 5.1 (Injection Preserves Evaluation). *For all* $s_1\in\mathsf{State}_1$ *and* $c_2\in\mathsf{Config}_2$, *if* $c_2\hookrightarrow r_2$ *under* tick_2, *then* $c_2\langle s_1\rangle\hookrightarrow r_2\langle s_1\rangle$ *under* tick_+.

Sketch. Induction on $\hookrightarrow$ under tick_2. □

Thus we can define $\rhd$ and ∞ that satisfies the desired property.

Definition 5.2 (Injection). For $S_1\subseteq\mathsf{State}_1$ and $A_2\subseteq\Sigma_2$, define:

$$S_1\rhd A_2\triangleq\{r_2\langle s_1\rangle|s_1\in S_1,r_2\in A_2\}\cup\{c_2\langle s_1\rangle\hookrightarrow r_2\langle s_1\rangle|s_1\in S_1,c_2\hookrightarrow r_2\in A_2\}$$

Definition 5.3 (Semantic Linking). For $S_1\subseteq\mathsf{State}_1$ and $A_2\subseteq\Sigma_2$, define:

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

Theorem 5.4 (Advance). *For all* $e\in\mathsf{Expr}$ *and* $S_1\subseteq\mathsf{State}_1$, $S_2\subseteq\mathsf{State}_2$,

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

5.3 Abstract Linking

We define injection and linking in the abstract semantics in the same way as the concrete semantics. Only the definition of $\dot m_2\langle\dot\sigma_1\rangle$ has to be adapted to account for the fact that $\dot m_2(\dot t)$ is now a *set* of closures. This means that $\dot m_2\langle\dot\sigma_1\rangle$ must be defined as:

$$\dot m_2\langle\dot\sigma_1\rangle \triangleq \lambda\dot t.\{\dot v_2\langle\dot\sigma_1\rangle | \dot v_2 \in \dot m_2(\dot t)\}$$

Then we can show that:

Lemma 5.5 (Injection Preserves Abstract Evaluation). *For all $\dot s_1 \in \dot{\mathsf{State}}_1$ and $\dot c_2 \in \dot{\mathsf{Config}}_2$, if $\dot c_2 \dot\hookrightarrow \dot r_2$ under $\dot{\mathsf{tick}}_2$, then $\dot c_2\langle\dot s_1\rangle \dot\hookrightarrow \dot r_2\langle\dot s_1\rangle$ under tick_+.*

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

and thus we can define:

Definition 5.6 (Abstract Injection). For $S_1^\# \subseteq \dot{\mathsf{State}}_1$ and $A_2^\# \subseteq \dot\Sigma_2$, define:

$$S_1^\# \rhd^\# A_2^\# \triangleq \{\dot r_2\langle\dot s_1\rangle | \dot s_1 \in S_1^\#, \dot r_2 \in A_2^\#\} \cup \{\dot c_2\langle\dot s_1\rangle \dot\hookrightarrow \dot r_2\langle\dot s_1\rangle | \dot s_1 \in S_1^\#, \dot c_2 \dot\hookrightarrow \dot r_2 \in A_2^\#\}$$

Definition 5.7 (Abstract Linking). For $S_1^\# \subseteq \dot{\mathsf{State}}_1$ and $A_2^\# \subseteq \dot\Sigma_2$, define:

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

so that the *best possible result* is achieved:

Theorem 5.8 (Abstract Advance). *For all $e \in \mathsf{Expr}$ and $S_1^\# \subseteq \dot{\mathsf{State}}_1$, $S_2^\# \subseteq \dot{\mathsf{State}}_2$,*

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

Since we have that

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

due to $\alpha_+(r_2\langle s_1\rangle) = \alpha_2(r_2)\langle\alpha_1(s_1)\rangle$, the above theorem directly leads to overapproximation by:

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

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 $\dot s$ that uses timestamps in $\{0, 1\}$. We want to analyze e under another initial state $\dot s'$ 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 $\dot s''$ that satisfies $\dot s\langle\dot s''\rangle = \dot s'$, so that linking $\dot s''$ into the cached results result in the semantics that started from $\dot s'$. 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 $\dot s$ and $\dot s'$ are equivalent, all $s \in \gamma(\{\dot s\})$ must have an equivalent $s' \in \gamma'(\{\dot s'\})$.
(2) If s and s' are equivalent, (e, s) and (e, s') must step to equivalent states.

$$
\begin{aligned}
\checkmark(\_, m, \epsilon) &\triangleq \mathsf{True} \\
\checkmark(\sigma, m, \xrightarrow{x} t\ p) &\triangleq t = \sigma(x) \land \checkmark(t, m, p) \\
\checkmark(\sigma, m, \xrightarrow{d} p) &\triangleq \checkmark(\sigma(d), m, p) \\
\checkmark(t, m, \xrightarrow{\lambda x.e} p) &\triangleq \langle \lambda x.e, {}^{\exists}\sigma\rangle = m(t) \land \checkmark(\sigma, m, p) \\
\checkmark(\text{otherwise}) &\triangleq \mathsf{False}
\end{aligned}
\qquad
\begin{aligned}
\dot{\checkmark}(\_, \dot{m}, \epsilon) &\triangleq \mathsf{True} \\
\dot{\checkmark}(\dot{\sigma}, \dot{m}, \xrightarrow{x} \dot{t}\ \dot{p}) &\triangleq \dot{t} = \dot{\sigma}(x) \land \dot{\checkmark}(\dot{t}, \dot{m}, \dot{p}) \\
\dot{\checkmark}(\dot{\sigma}, \dot{m}, \xrightarrow{d} \dot{p}) &\triangleq \dot{\checkmark}(\dot{\sigma}(d), \dot{m}, \dot{p}) \\
\dot{\checkmark}(\dot{t}, \dot{m}, \xrightarrow{\lambda x.e} \dot{p}) &\triangleq \langle \lambda x.e, {}^{\exists}\dot{\sigma}\rangle \in \dot{m}(\dot{t}) \land \dot{\checkmark}(\dot{\sigma}, \dot{m}, \dot{p}) \\
\dot{\checkmark}(\text{otherwise}) &\triangleq \mathsf{False}
\end{aligned}
$$

Fig. 8. Definitions for the $\checkmark$ and $\dot{\checkmark}$ predicates.

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

6.2 Definitions

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

We first define what it means for two states $s \in \mathsf{State}$ and $s' \in \mathsf{State}'$ to be equivalent. Recall that $s = (\sigma, m, t)$ and $s' = (\sigma', m', t')$ for some contexts σ, σ', 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 σ and m components "look the same".

Note that information in σ and m is only accessed through a sequence of names x and d. Thus, one may imagine access "paths" with names on the edges and reachable timestamps on the vertices as representing the way that (σ, 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 $\mathbb{T}'$.

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

From now on, we shall write Path for the set of access paths that use timestamps in $\mathbb{T}$, and Path$'$ for the set of access paths that use timestamps in $\mathbb{T}'$. Then given an access path, we can define a predicate $\checkmark \in (\mathsf{Ctx} + \mathbb{T}) \times \mathsf{Mem} \times \mathsf{Path} \to \mathsf{Prop}$. $\checkmark(r, m, p)$ is true iff starting from r, all accesses edges in p are valid. Likewise, we can define a predicate $\dot{\checkmark} \in (\dot{\mathsf{Ctx}} + \dot{\mathbb{T}}) \times \dot{\mathsf{Mem}} \times \dot{\mathsf{Path}} \to \mathsf{Prop}$. $\dot{\checkmark}(\dot{r}, \dot{m}, \dot{p})$ is true iff starting from $\dot{r}$, all access edges in $\dot{p}$ are valid. The definitions for $\checkmark$, $\dot{\checkmark}$ are given in Figure 8.

Now we can give straightforward definitions of equivalence.

Definition 6.1 (Equivalent Concrete States). Let $s = (\sigma, m, \_) \in \mathsf{State}$ and $s' = (\sigma', m', \_) \in \mathsf{State}'$. $s \sim s'$ (s is equivalent to s') when $\exists \varphi \in \mathbb{T} \to \mathbb{T}', \varphi' \in \mathbb{T}' \to \mathbb{T}$:

(1) $\forall p \in \mathsf{Path} : \checkmark(\sigma, m, p) \Rightarrow (\checkmark(\sigma', m', \varphi(p)) \land p = \varphi'(\varphi(p)))$
(2) $\forall p' \in \mathsf{Path}' : \checkmark(\sigma', m', p') \Rightarrow (\checkmark(\sigma, m, \varphi'(p')) \land p' = \varphi(\varphi'(p')))$

Definition 6.2 (Weakly Equivalent Abstract States). Let $\dot{s} = (\dot{\sigma}, \dot{m}, \_) \in \dot{\mathsf{State}}$ and $\dot{s'} = (\dot{\sigma'}, \dot{m'}, \_) \in \dot{\mathsf{State}}'$. $\dot{s}$ is weakly equivalent to $\dot{s'}$ when $\exists \dot{\varphi} \in \dot{\mathbb{T}} \to \dot{\mathbb{T}}', \dot{\varphi}' \in \dot{\mathbb{T}}' \to \dot{\mathbb{T}}$:

(1) $\forall \dot{p} \in \dot{\mathsf{Path}} : \dot{\checkmark}(\dot{\sigma}, \dot{m}, \dot{p}) \Rightarrow (\dot{\checkmark}(\dot{\sigma'}, \dot{m'}, \dot{\varphi}(\dot{p})) \land \dot{p} = \dot{\varphi}'(\dot{\varphi}(\dot{p})))$
(2) $\forall \dot{p'} \in \dot{\mathsf{Path}}' : \dot{\checkmark}(\dot{\sigma'}, \dot{m'}, \dot{p'}) \Rightarrow (\dot{\checkmark}(\dot{\sigma}, \dot{m}, \dot{\varphi}'(\dot{p'})) \land \dot{p'} = \dot{\varphi}(\dot{\varphi}'(\dot{p'})))$

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

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

and

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

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* $\dot\varphi, \dot\varphi'$ when $\dot\varphi, \dot\varphi'$ are the functions that translate between abstract timestamps in Definition 6.2. Second, we say that $\dot t$ is *reachable from* $\dot s$ when there is some valid access path $\dot p$ from $\dot s$ containing $\dot t$. Now we actually give the definition:

Definition 6.3 (Equivalent Abstract States). Let $\dot s = (\_, \dot m, \_) \in \dot{\mathsf{State}}$ and $\dot s' = (\_, \dot m', \_) \in \dot{\mathsf{State}}'$. $\dot s \mathrel{\dot\sim} \dot s'$ ($\dot s$ is equivalent to $\dot s'$) when $\exists \dot\varphi \in \dot{\mathbb{T}} \to \dot{\mathbb{T}}', \dot\varphi' \in \dot{\mathbb{T}}' \to \dot{\mathbb{T}}$:

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

We extend the definition of equivalence between elements of Right and Right' as the conjunction of the syntactic equality in the expression parts and the equivalence in the context and memory parts. Then we can extend the definition of equivalence between $A \subseteq \Sigma$ and $A' \subseteq \Sigma'$ by requiring all elements $c \hookrightarrow r, r$ of A to have an equivalent counterpart in A', and vice versa. Likewise, we can extend the definition for equivalent abstract states as well.

When $A \subseteq \Sigma$ and $A' \subseteq \Sigma'$ are equivalent, we override the symbol for equivalence between individual states and write $A \sim A'$. When $A^\# \subseteq \dot\Sigma$ and $A'^\# \subseteq \dot\Sigma'$ are equivalent, we write $A^\# \sim^\# A'^\#$.

6.3 Propeties of Equivalence

We first note that the relations $\sim$ and $\dot\sim$ 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 6.4 (Concretization Preserves Equivalence). *Assume that each $\dot t, \dot t'$ in $\dot{\mathbb{T}}, \dot{\mathbb{T}}'$ corresponds to an infinite set of concrete timestamps. Then for all $S^\# \subseteq \dot{\mathsf{State}}$ and $S'^\# \subseteq \dot{\mathsf{State}}'$,*

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

Sketch. We want to prove:

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

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 6.5 (Evaluation Preserves Equivalence). *For all $c \in \mathsf{Config}$, $r \in \mathsf{Right}$, $c' \in \mathsf{Config}'$,*

$$c \hookrightarrow r \text{ and } c \sim c' \Rightarrow \exists r' : c' \hookrightarrow r' \text{ and } r \sim r'$$

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 6.4. We have required that all partitions $\dot{\alpha}^{-1}(\dot{t})$ 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 $\dot{\mathbb{T}}_2$ and a function $\dot{\mathsf{tick}}_2 \in \dot{\mathbb{T}}_2 \to \dot{\mathbb{T}}_2$.
Step 2 Assume an initial condition $S_2^\#$ and compute $[\![e]\!]^\# S_2^\#$.
Step 3 Choose a finite set $\dot{\mathbb{T}}_1$ and $\dot{\mathsf{tick}}_1 \in \dot{\mathbb{T}}_1 \to \dot{\mathbb{T}}_1$.
Step 4 Find a $S_1^\#$ such that $S_1^\# \rhd^\# S_2^\#$ 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:

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

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

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

REFERENCES