A Syntax-Guided Framework for Modular Analysis

JOONHYUP LEE

1 ABSTRACT SYNTAX

In this section we define the abstract syntax for a simple language that captures the essence of modules and linking. The language is basically an extension of untyped lambda calculus with modules and the linking construct.

$$
\begin{array}{lrcll}
\text{Identifiers} & x, M & \in & \text{Var} & \\
\text{Expression} & e & \rightarrow & x & \text{value identifier} \\
 & & | & \lambda x.e & \text{function} \\
 & & | & e\ e & \text{application} \\
 & & | & e \infty e & \text{linked expression} \\
 & & | & \varepsilon & \text{empty module} \\
 & & | & M & \text{module identifier} \\
 & & | & \texttt{let}\ x\ e\ e & \text{binding expression} \\
 & & | & \texttt{let}\ M\ e\ e & \text{binding module}
\end{array}
$$

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

1.1 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 the nonterminals for the modules and expressions are not separated. Why is this so?

The rationale for the exclusion of recursive modules/first-class modules/functors is because we want to enforce static scoping. That is, we need to be able to statically determine where variables were bound when using them. To enforce static scoping when function applications might return modules, we need to employ signatures to project the dynamically computed modules onto a statically known context. Concretely, we need to define signatures S where $\lambda M \texttt{:>} S.e$ statically resolves the context when M is used in the body e, and $(e_1\ e_2)\texttt{:>} S$ enforces that a dynamic computation is resolved into one static form. 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 $(m_1 \infty m_2) \infty e$. $m_1 \infty m_2$ links a module with a module, and $(m_1 \infty m_2) \infty e$ 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.

Author's address: Joonhyup Lee.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Association for Computing Machinery.
XXXX-XXXX/2023/8-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 CONCRETE SEMANTICS

In this section, we present the dynamics of the simple language presented in the previous section.

2.1 Structural Operational Semantics

First, we give the operational semantics for the dynamic execution of the module language. The one-step transition relation $\rightsquigarrow$ will relate a configuration(expression and state) either to (1) another configuration of which its results are used for the evaluation of the first configuration, or to (2) the final result.

Prior to defining this relation, the semantic domains must be set up. As is common in defining dynamics for call-by-value lambda calculus, one must define *environments* to record what values were bound to variables. For the ease of program analysis, this environment is divided again into (1) a context C that binds variables in scope to the *time* those variables were bound, and (2) a memory m which records which values were bound at what time.

The representation of the context C is a stack that records variables *in the order* they were bound. In the spirit of de Bruijn, to access the value of a variable x, one has to read off the closest binding time from C and consult the memory to determine what value was bound at that time. In contrast, to access the exported context from a variable M, one has to look up the exported context from C, not from the memory.

This separation between where we store modules and where we store closures emphasizes the fact that *where* the variables are bound is guided by syntax. The only thing that is dynamic is *when* the variables are bound, which is represented by the time component.

Now, we start by defining what we mean by *time* and *context*, which is the essence of our model.

2.1.1 Time and Context. We first define sets that are parametrized by our choice of the time domain, namely the *value*, *memory*, and *context* domains. Also, we present the notational conventions used in this paper to represent members of each domain.

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

Fig. 2. Definition of the semantic domains.

Note that $\mathrm{State}(\mathbb{T}) \subseteq \mathrm{Result}(\mathbb{T})$. This is because the results from modules are the states that they export. Later on, when we define predicates on results, it is to be understood that their definition applies to states as well.

Also, note that we have defined a domain $\mathrm{Tick}(\mathbb{T})$ which is comprised of functions that receive a state, a variable, and a value and returns a timestamp. As can be inferred from the name of the domain, a $\mathsf{tick} \in \mathrm{Tick}(\mathbb{T})$ is the policy that the designer of the analysis chooses to represent the concrete flow of the program. The time $\mathsf{tick}(s, x, v)$ is the time that is incremented when the value

v is bound to a variable x under state s. Naturally, our definition of the one-step transition relation is parametrized by the choice of tick.

Why does the tick function for the concrete time take in s, x, v? This is a suggestion to the analysis designer. For program analysis, the designer must think of an *abstract* tick operator that *simulates* its concrete counterpart. If the concrete tick function is not able to take into account the environment that the time is incremented, the abstraction of the tick function will not be able to hold much information about the execution of the program.

Now for the auxiliary operators that is used when defining the evaluation relation. We define the function that extracts the address for an value id x, and the function that looks up the dynamic context bound to a module id M.

$$\mathsf{addr}(C,x) \triangleq \begin{cases} \bot & C = [\,] \\ t & C = (x,t) :: C' \\ \mathsf{addr}(C',x) & C = (x',t) :: C' \land x' \neq x \\ \mathsf{addr}(C'',x) & C = (M,C') :: C'' \end{cases} \qquad \mathsf{ctx}(C,M) \triangleq \begin{cases} \bot & C = [\,] \\ C' & C = (M,C') :: C'' \\ \mathsf{ctx}(C'',M) & C = (M',C') :: C'' \land M' \neq M \\ \mathsf{ctx}(C',M) & C = (x,t) :: C' \end{cases}$$

Fig. 3. Definitions for the addr and ctx operators.

2.1.2 The Relation. Now we define the one-step transition relation. The relation $\rightsquigarrow_{\mathsf{tick}}$ relates $(e, C, m, t) \in \text{Expr} \times \text{State}(\mathbb{T})$ with either $(V, m, t) \in \text{Result}(\mathbb{T})$ or the next expression and state, when tick is used to increment the time. Note that we constrain whether an expression returns v or C by the definition of $\rightsquigarrow_{\mathsf{tick}}$. The complete definition for the relation is given in Fig. 4. Also, the equivalence of the relation with a reference interpreter is formalized in Coq.

Our definition of $\rightsquigarrow_{\mathsf{tick}}$ is parametrized by the choice of $\mathbb{T}$, and the choice of tick. Note that without putting some constraints on the behavior of tick, the semantics might not agree with what is conventionally accepted as an operational semantics for the lambda calculus, such as the CESK machine. That is, the tick function must produce *fresh* timestamps. To ensure that tick is well-behaved, we always assume that: (1) tick produces a strictly larger timestamp, and (2) all reachable addresses are bound by the current time. The first guarantee is formalized by:

Definition 2.1 (Concrete time). $(\mathbb{T}, \leq, \mathsf{tick})$ is a *concrete time* when

(1) $(\mathbb{T}, \leq)$ is a total order.
(2) $\mathsf{tick} \in \text{Tick}(\mathbb{T})$ satisfies: $\forall t \in \mathbb{T} : t < \mathsf{tick}((\_,\_,t),\_,\_)$

and the second guarantee is formalized by:

Definition 2.2 (Time-boundedness). All $(V, m, t) \in \text{Result}(\mathbb{T})$ satisfies $V < t$ and $m < t$, when:

$$C < t \triangleq \begin{cases} \mathsf{True} & C = [\,] \\ t' < t \land C' < t & C = (x,t') :: C' \\ C' < t \land C'' < t & C = (M,C') :: C'' \end{cases} \qquad V < t \triangleq \begin{cases} C < t & V = \langle \_, C \rangle \\ C < t & V = C \end{cases}$$
$$m < t \triangleq \forall t' \in \mathsf{dom}(m) : t' < t \land m(t') < t$$

The above two definitions ensure that tick allocates fresh timestamps. That is, it does not matter what tick you choose to instantiate $\rightsquigarrow$ as long as it satisfies our simple requirement. Still, how do we formalize this notion, that the choice of tick is "irrelevent" in formulating the semantics? We first define what it means for two results(and states) to be isomorphic, and prove that for isomorphic initial states, all reachable configurations are isomorphic, *no matter what* tick is used.

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

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

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

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

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

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

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

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

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

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

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

Fig. 4. The concrete one-step transition relation. The subscript tick is omitted for brevity.

Definition 2.3 (Isomorphic results). Let $r = (V, m, t) \in \text{Result}(\mathbb{T})$ and $r' = (V', m', t') \in \text{Result}(\mathbb{T}')$. We say r is *isomorphic* to r' and write $r \cong r'$ when there exists $f : \mathbb{T} \to \mathbb{T}'$ and $g : \mathbb{T}' \to \mathbb{T}$ such that $f(V) = V' \wedge f \circ m = m' \circ f \wedge f(t) = t'$ and $g(V') = V \wedge g \circ m' = m \circ g \wedge g(t') = t$. $f(V)$ is defined by mapping f over all timestamps in the context part of V.

f and g in the above definition serves to translate timestamps from one time domain to another, and since the results have the same structure, the translations must be the same. Naturally, this definition can be extended between $\ell \in \mathrm{L} \triangleq \text{Expr} \times \text{State}(\mathbb{T})$, which is the **l**eft-hand-side of $\rightsquigarrow$, and $\rho \in \mathrm{R} \triangleq \mathrm{L} \cup \mathrm{Result}(\mathbb{T})$, which is the **r**ight-hand-side. That is, we say that $\ell \cong \ell'$ when the expression parts are equal and the state parts are isomorphic, and $\rho \cong \rho'$ follows directly, since $\rho \in \mathrm{L}$ or $\rho \in \mathrm{Result}(\mathbb{T})$. Now we state what it means for the tick function to be irrelevent.

Theorem 2.1 (Irrelevence of tick)**.** Let $s \in \mathrm{State}(\mathbb{T})$ and $s' \in \mathrm{State}(\mathbb{T}')$. If $s \cong s'$, then:

$$\forall \mathsf{tick}, \mathsf{tick}', e, \rho : (e, s) \rightsquigarrow_{\mathsf{tick}} \rho \Rightarrow \exists \rho' : (e, s') \rightsquigarrow_{\mathsf{tick}'} \rho' \land \rho \cong \rho'$$

The above theorem suggests (1) $\rightsquigarrow_{\mathsf{tick}}$ is well-defined, and (2) no matter what $\mathbb{T}$ and tick we approximate in our analysis, the results will be valid in the sense that it models all concrete executions starting from an isomorphic initial state. For example, in the common case when one would like to analyze an expression evaluated starting from $([], \varnothing, 0)$, any instantiation of $\mathbb{T}$ or tick will be meaningful. Moreover, later on, when we describe concrete linking, we link two states to obtain a state *isomorphic* to what is exported from an external module. Theorem 2.1 provides the reason why such a process is acceptable in describing the semantics of a linked expression.

2.2 Collecting Semantics

For program analysis, we need to define a collecting semantics that captures the strongest property we want to model. In the case of modular analysis, we need to collect *all* intermediate nodes in the proof tree when trying to prove what the initial configuration evaluates to. Consider the case when $e_1 \infty e_2$ is evaluated from state s. Since e_2 has free variables that are exported by e_1, separately analyzing e_2 will result in an incomplete proof tree. What it means to separately analyze, then link two expressions e_1 and e_2 is to (1) compute what e_1 will export to e_2, (2) partially compute the proof tree for e_2, and (3) inject the exported context into the partial proof to complete the execution.

What should be the *type* of the collecting semantics? The analysis must keep track of all configurations that were reached along with the results computed from the intermediate configurations, thus it must be a set that collects all those elements. Concretely, the collecting semantics $[\![e]\!]S$ of an expression e evaluated under initial conditions in $S \subseteq \mathrm{State}(\mathbb{T}) \times \mathrm{Tick}(\mathbb{T})$ must be a subset of $(\mathrm{L} \times \mathrm{Tick}(\mathbb{T}) \times \mathrm{R}) \cup (\mathrm{R} \times \mathrm{Tick}(\mathbb{T}))$, when L and R are the left and right sides of $\rightsquigarrow_{\mathsf{tick}}$ as defined in the previous subsection, and $\mathrm{Tick}(\mathbb{T})$ specifies what tick is used.

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, $D = \wp((\mathrm{L} \times \mathrm{Tick}(\mathbb{T}) \times \mathrm{R}) \cup (\mathrm{R} \times \mathrm{Tick}(\mathbb{T})))$ as defined previously. Defining the transfer function is straightforward from the definition of the transition relation.

Definition 2.4 (Transfer function)**.** Given $A \subseteq (\mathrm{L} \times \mathrm{Tick}(\mathbb{T}) \times \mathrm{R}) \cup (\mathrm{R} \times \mathrm{Tick}(\mathbb{T}))$, define

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

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

Definition 2.5 (Concrete semantics)**.**

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

We extend $\cong$ to a relation between two caches $A \subseteq (\mathrm{L} \times \mathrm{Tick}(\mathbb{T}) \times \mathrm{R}) \cup (\mathrm{R} \times \mathrm{Tick}(\mathbb{T}))$ and $A' \subseteq (\mathrm{L} \times \mathrm{Tick}(\mathbb{T}') \times \mathrm{R}) \cup (\mathrm{R} \times \mathrm{Tick}(\mathbb{T}'))$ to mean that: (1) $\forall (\rho, \_) \in A : \exists (\rho', \_) \in A' : \rho \cong \rho'$ and vice versa, and (2) $\forall \ell \rightsquigarrow \rho \in A : \exists \ell' \rightsquigarrow \rho' \in A' : \ell \cong \ell' \land \rho \cong \rho'$ and vice versa. It is clear from Theorem 2.1 that if $S \cong S'$, then $[\![e]\!]S \cong [\![e]\!]S'$.

3 CONCRETE LINKING

Before we go into definitions, we would like to make our objectives clear. Assume we want to analyze $e_1 \propto e_2$ under initial condition S. Normally, the final results of $e_1 \propto e_2$ are calculated by first calculating the results for e_1 under S, which is exported to e_2, then calculating the results for e_2 under the exported states. That is, if we write $|[\![e]\!]S|$ for the final results of e under S, $|[\![e_1 \propto e_2]\!]S| = |[\![e_2]\!]|[\![e_1]\!]S||$.

Instead, what we want to do is to calculate a part of $[\![e_2]\!]|[\![e_1]\!]S|$ *in advance*, then fill in the blanks later to obtain $|[\![e_1 \propto e_2]\!]S|$. Since we do not know what $|[\![e_1]\!]S|$ will be, we must *assume* an initial state S_2 for e_2, which must be isomorphic to a *fragment* of what e_1 will export to e_2. For example, if we assume that when e_1 returns, the identifier `id` is bound to $\langle \lambda x.x, [] \rangle$, S_2 will be something like $\{((\mathtt{id}, 0) :: [], \{0 \mapsto \langle \lambda x.x, [] \rangle\}, 1)\}$.

So we first analyze $[\![e_2]\!]S_2$. Later on, after we have calculated $|[\![e_1]\!]S|$, we check if our assumption is *guaranteed*. That is, we check if $|[\![e_1]\!]S| \cong S_1 \rhd S_2$, when $S_1 \rhd S_2$ means that $|[\![e_1]\!]S|$ can be separated as the injection($\rhd$) of some S_1 into our assumed S_2. Then, we link($\propto$) the missing part S_1 with the separately analyzed $[\![e_2]\!]S_2$ to obtain the final result for $e_1 \propto e_2$. Thus our main theorem will be:

Theorem 3.1 (Concrete Linking). For $S \subseteq \mathrm{State}(\mathbb{T}) \times \mathrm{Tick}(\mathbb{T})$ and $S_i \subseteq \mathrm{State}(\mathbb{T}_i) \times \mathrm{Tick}(\mathbb{T}_i)$,

$$|[\![e_1 \propto e_2]\!]S| \cong |S_1 \propto [\![e_2]\!]S_2|$$

where $|[\![e_1]\!]S| \cong S_1 \rhd S_2$.

Now, to formalize the above theorem, we need to define (1) what $|[\![e]\!]S|$ is, (2) what $S_1 \rhd S_2$ is, and (3) what $S_1 \propto A_2$ is. The first definition is straightforward: $|[\![e]\!]S| \triangleq \{r | (e, \_) \rightsquigarrow r \in [\![e]\!]S\}$, and we understand $|S_1 \propto [\![e]\!]S_2|$ to be $\{r | (e, \_) \rightsquigarrow r \in S_1 \propto [\![e]\!]S_2\}$. The definitions for the injection and (semantic) linking operators need more consideration.

3.1 Injection and Deletion

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

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

As the first step in defining $\rhd$, we first define the injection operator for contexts, when $C_2\langle C_1 \rangle$ "fills in the blank" in C_2 with C_1. The deletion operator $C_2\langle C_1 \rangle^{-1}$, which "digs out" C_1 from C_2, is also defined. Why it is defined might not so be obvious here, but it is necessary to guarantee the second property we expect of $\rhd$.

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

Fig. 5. Definition of the injection operator $C_2\langle C_1 \rangle$ and the deletion operator $C_2\langle C_1 \rangle^{-1}$.

Note that if we inject $C_1 \in \mathrm{Ctx}(\mathbb{T}_1)$ into $C_2 \in \mathrm{Ctx}(\mathbb{T}_2)$, we obtain $C_2\langle C_1\rangle \in \mathrm{Ctx}(\mathbb{T}_1+\mathbb{T}_2)$. Why the linked time is the separate sum of the two time domains is because we want to separate what came from outside and what was assumed. Then naturally, we expect tick_+ to increment timestamps in $\mathbb{T}_1+\mathbb{T}_2$ by using tick_1 for timestamps in $\mathbb{T}_1$, and by using tick_2 for timestamps in $\mathbb{T}_2$. However, the context and memory will contain timestamps both in $\mathbb{T}_1$ and $\mathbb{T}_2$. Therefore, we need to define the filter operations $C.1$ and $C.2$ which selects only timestamps from the time domain of interest.

$$C.i \triangleq \begin{cases} [] & C = [] \\ (x,t) :: C'.i & C = (x,t) :: C' \wedge t \in \mathbb{T}_i \\ C'.i & C = (x,t) :: C' \wedge t \notin \mathbb{T}_i \\ (M, C'.i) :: C''.i & C = (M, C') :: C'' \end{cases} \qquad V.i \triangleq \begin{cases} C.i & V = C \\ \langle \lambda x.e, C.i\rangle & V = \langle \lambda x.e, C\rangle \end{cases}$$

$$m.i \triangleq \bigcup_{t \in \mathsf{dom}(m)\cap\mathbb{T}_i} \{t \mapsto m(t).i\}$$

Fig. 6. Definition for the filter operations ($i = 1, 2$).

Now we give the definition for $\triangleright$.

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

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

Note that $r_2\langle s_1\rangle \in \mathrm{Result}(\mathbb{T}_1+\mathbb{T}_2)$ is time-bounded if we define the order relation on $\mathbb{T}_1+\mathbb{T}_2$ as $t_1 < t_2$ for all $t_1 \in \mathbb{T}_1$ and $t_2 \in \mathbb{T}_2$. Also, we define $V_2\langle C_1\rangle^{-1}$ and $m_2\langle C_1\rangle^{-1}$ analogously to injection. Now we only have to define tick_+ which preserves the separate transitions even after injection.

Definition 3.2 (Injection). Let $(s_1, \mathsf{tick}_1) \in \mathrm{State}(\mathbb{T}_1) \times \mathrm{Tick}(\mathbb{T}_1)$ and $(r_2, \mathsf{tick}_2) \in \mathrm{Result}(\mathbb{T}_2) \times \mathrm{Tick}(\mathbb{T}_2)$. We define $(s_1, \mathsf{tick}_1) \triangleright (r_2, \mathsf{tick}_2) \triangleq (r_2\langle s_1\rangle, \mathsf{tick}_+) \in \mathrm{Result}(\mathbb{T}_1+\mathbb{T}_2) \times \mathrm{Tick}(\mathbb{T}_1+\mathbb{T}_2)$, when tick_+ is given by:

$$\mathsf{tick}_+((C,m,t),x,v) \triangleq \begin{cases} \mathsf{tick}_1((C.1, m.1, t), x, v.1) & t \in \mathbb{T}_1 \\ \mathsf{tick}_2((C\langle C_1\rangle^{-1}.2, m\langle C_1\rangle^{-1}.2, t), x, v\langle C_1\rangle^{-1}.2) & t \in \mathbb{T}_2 \end{cases}$$

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

3.2 Semantic Linking

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

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

Fig. 7. Extension of $\triangleright$ to define injection into a cache.

Now we need to define the semantic linking operator $\propto$. More specifically, we must define $S_1 \propto A_2$, when $S_1 \subseteq \mathrm{State}(\mathbb{T}_1) \times \mathrm{Tick}(\mathbb{T}_1)$ and $A_2 \subseteq (\mathrm{L}_2 \times \mathrm{Tick}(\mathbb{T}_2) \times \mathrm{R}_2) \cup (\mathrm{R}_2 \times \mathrm{Tick}(\mathbb{T}_2))$. Remember that A_2 is the separately computed semantics, and S_1 is what was missing. Thus, we must first inject all $(s_1, \mathsf{tick}_1) \in S_1$ into $(\rho_2, \mathsf{tick}_2), \ell_2 \rightsquigarrow_{\mathsf{tick}_2} \rho_2 \in A_2$. The definition for elementwise injection into a cache is given in Fig. 7. Next, since we have gained new information about the external environment, we must collect more that can be gleaned from S_1. Thus, the definition of semantic linking is as follows:

Definition 3.3 (Semantic Linking)**.** Let $S_1 \subseteq \text{State}(\mathbb{T}_1) \times \text{Tick}(\mathbb{T}_1)$ and $A_2 \subseteq (\text{L}_2 \times \text{Tick}(\mathbb{T}_2) \times \text{R}_2) \cup (\text{R}_2 \times \text{Tick}(\mathbb{T}_2))$. Then:

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

Since we defined $\rhd$ and thus ∞ well, we have the following property:

Lemma 3.1 (Advance)**.** Let $S_1 \subseteq \text{State}(\mathbb{T}_1) \times \text{Tick}(\mathbb{T}_1)$ and $S_2 \subseteq \text{State}(\mathbb{T}_2) \times \text{Tick}(\mathbb{T}_2)$. Then:

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

This means that we can compute part of $[\![e]\!]S$ in *advance*, when S is separable into $S_1 \rhd S_2$, by $[\![e]\!]S_2$, then link S_1 later to obtain the full semantics. Thus our main theorem follows directly: since $|[\![e_1]\!]S| \cong S_1 \rhd S_2$(separability),

$$|[\![e_1 \infty e_2]\!]S| = |[\![e_2]\!]|[\![e_1]\!]S|| \cong |[\![e_2]\!](S_1 \rhd S_2)| = |S_1 \infty [\![e_2]\!]S_2|$$

when the first equality is from the definition of $|[\![e]\!]S|$, $\cong$ is due to the separability assumption and irrelevence of tick, and the final equality is due to the advance lemma.

3.3 A Simple Case

The most obvious case in separability is when e_2 does not depend on what e_1 exports. In this case, $S_2 = \mathsf{empty} \triangleq \{([], \varnothing, 0)\}$. Since any S is trivially separable as $S \cong S \rhd \mathsf{empty}$, we have that $|[\![e_1]\!]S| \cong |[\![e_1]\!]S| \rhd \mathsf{empty}$. Thus, we have:

Corollary 3.1 (A Simple Case)**.**

$$|[\![e_1 \infty e_2]\!]S| \cong ||[\![e_1]\!]S| \infty [\![e_2]\!]\mathsf{empty}|$$

4 ABSTRACT SEMANTICS

The abstract semantics is almost exactly the same as the concrete semantics, except for the fact that the memory domain is now a finite map from the abstract time domain to a *set* of values. Note we do not need to define the $C^\#, v^\#, V^\#$ components, as they are *exactly* their concrete counterparts. They are simply C, v, V, parametrized by a different $\mathbb{T}$.

$$
\begin{array}{rccl}
\text{Abstract Time} & t^\# & \in & \mathbb{T}^\# \\
\text{Environment/Context} & C^\# & \in & \mathrm{Ctx}(\mathbb{T}^\#) \\
\text{Value of expressions} & v^\# & \in & \mathrm{Val}(\mathbb{T}^\#) \\
\text{Value of expressions/modules} & V^\# & \in & \mathrm{Val}(\mathbb{T}^\#) + \mathrm{Ctx}(\mathbb{T}^\#) \\
\text{Abstract Memory} & m^\# & \in & \mathrm{Mem}^\#(\mathbb{T}^\#) \triangleq \mathbb{T}^\# \xrightarrow{\text{fin}} \wp(\mathrm{Val}(\mathbb{T}^\#)) \\
\text{Abstract State} & s^\# & \in & \mathrm{State}^\#(\mathbb{T}^\#) \triangleq \mathrm{Ctx}(\mathbb{T}^\#) \times \mathrm{Mem}\mathbb{T}^\# \times \mathbb{T}^\# \\
\text{Abstract Result} & r^\# & \in & \mathrm{Result}^\#(\mathbb{T}^\#) \triangleq (\mathrm{Val}(\mathbb{T}^\#) + \mathrm{Ctx}(\mathbb{T}^\#)) \times \mathrm{Mem}^\#(\mathbb{T}^\#) \times \mathbb{T}^\#
\end{array}
$$

Fig. 8. Definition of the semantic domains.

First the abstract evaluation relation $\rightsquigarrow^\#$ is defined. Note that the update for the memory is now a weak update. That is,

Definition 4.1 (Weak update). Given $m^\# \in \mathrm{Mem}^\#(\mathbb{T}^\#)$, $t^\# \in \mathbb{T}^\#$, $v^\# \in \mathrm{Val}(\mathbb{T}^\#)$, define $m^\#[t^\# \mapsto^\# v^\#]$ as:

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

Also, for the abstract time, we do not enforce the existence of an ordering on the timestamps, but we do need a policy for performing the tick operation. The abstract $\mathrm{tick}^\#$ must simulate the tick function, so it must have the same type as tick.

Definition 4.2 (Abstract time). $(\mathbb{T}^\#, \mathrm{tick}^\#)$ is an *abstract time* when $\mathrm{tick}^\# : \mathrm{Ctx}(\mathbb{T}^\#) \to \mathrm{Mem}^\#(\mathbb{T}^\#) \to \mathbb{T}^\# \to \mathrm{Var} \to \mathrm{Val}(\mathbb{T}^\#) \to \mathbb{T}^\#$ is the policy for advancing the timestamp.

The abstract one-step reachability relation is defined in Fig. 9. From this relation, we can define the abstract semantics in the same way as the concrete version.

Definition 4.3 (Transfer function). Given $A^\# \subseteq (L^\# \times R^\#) \cup R^\#$, define

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

Definition 4.4 (Abstract semantics).

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

5 WHOLE-PROGRAM ANALYSIS

This section clarifies what we mean by that the abstract semantics is a *sound approximation* of the concrete semantics. Since the only values in our language are closures that pair code with a context, we can make a Galois connection between $\wp((L \times R) \cup R)$ and $\wp((L^\# \times R^\#) \cup R^\#)$ given a function $\alpha : \mathbb{T} \to \mathbb{T}^\#$.

Definition 5.1 (Extensions of abstraction). Given a function $\alpha : \mathbb{T} \to \mathbb{T}^\#$,

- Extend α to a function on $\mathrm{Ctx}(\mathbb{T}) \to \mathrm{Ctx}(\mathbb{T}^\#)$ by mapping α over all timestamps.
- Extend α to a function on $\mathrm{Val}(\mathbb{T}) \to \mathrm{Val}(\mathbb{T}^\#)$ by mapping α over all timestamps.

$$\boxed{(e,C^\#,m^\#,t^\#)\rightsquigarrow^\#(V^\#,m'^\#,t'^\#)/(e',C'^\#,m'^\#,t'^\#)}$$

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

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

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

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

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

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

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

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

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

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

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

Fig. 9. The abstract one-step reachability relation.

- Extend α to a function on $\text{Mem}(\mathbb{T}) \to \text{Mem}^\#(\mathbb{T}^\#)$ by defining
$$\alpha(m) \triangleq \bigcup_{t \in \mathsf{dom}(m)} [\alpha(t) \mapsto \{\alpha(m(t))\}]$$
- Extend α to a function on $\text{Result}(\mathbb{T}) \to \text{Result}^\#(\mathbb{T}^\#)$ by $\alpha(V, m, t) \triangleq (\alpha(V), \alpha(m), \alpha(t))$
- Extend α to a function on $L \to L^\#$ by $\alpha(e, s) \triangleq (e, \alpha(s))$.

Then it is obvious that:

Lemma 5.1 (Galois connection)**.** Given a function $\alpha : \mathbb{T} \to \mathbb{T}^\#$,
- Extend α by $\alpha(A) \triangleq \{\alpha(\rho) | \rho \in A\} \cup \{\alpha(\ell) \rightsquigarrow^\# \alpha(\rho) | \ell \rightsquigarrow \rho \in A\}$.
- Define γ by $\gamma(A^\#) \triangleq \{\rho | \alpha(\rho) \in A^\#\} \cup \{\ell \rightsquigarrow \rho | \alpha(\ell) \rightsquigarrow^\# \alpha(\rho) \in A^\#\}$.

Then $\forall A \subseteq (L \times R) \cup R, A^\# \subseteq (L^\# \times R^\#) \cup R^\# : \alpha(A) \subseteq A^\# \Leftrightarrow A \subseteq \gamma(A^\#)$.

The ordering between elements of $\wp((L \times R) \cup R)$ and $\wp((L^\# \times R^\#) \cup R^\#)$ is the subset order, because currently the only thing we are abstracting is the *time* component, which describes the control flow of the program. That is, the abstract semantics can be viewed as a control flow graph of the program, with the notion of "program points" described by $\rho^\# \in R^\#$ and the edges described by $\rightsquigarrow^\#$. Then all we need to show is that the abstract semantics overapproximates the concrete semantics, i.e, that $[\![e]\!](s) \subseteq \gamma([\![e]\!]^\#(\alpha(s)))$. However, it is not the case that this holds for arbitrary α. It must be that, as emphasized constantly in the previous sections, that $\mathsf{tick}^\#$ is a sound approximation of tick with respect to α.

Definition 5.2 (Tick-approximating abstraction)**.** Given a concrete time $(\mathbb{T}, \leq, \mathsf{tick})$ and an abstract time $(\mathbb{T}^\#, \mathsf{tick}^\#)$, a function $\alpha : \mathbb{T} \to \mathbb{T}^\#$ is said to be *tick-approximating* if:
$$\forall C, m, x, t, v : \alpha(\mathsf{tick}\ C\ m\ x\ t\ v) = \mathsf{tick}^\#\ \alpha(C)\ \alpha(m)\ x\ \alpha(t)\ \alpha(v)$$

Now we can prove that:

Theorem 5.1 (Soundness)**.** Given a tick-approximating $\alpha : \mathbb{T} \to \mathbb{T}^\#$,
$$\forall s \in \text{State}(\mathbb{T}) : [\![e]\!](s) \subseteq \gamma([\![e]\!]^\#(\alpha(s)))$$

What's not obvious is that if the abstract time domain is finite, the analysis is guaranteed to terminate. Since bindings for modules also exist in the stack C, showing that the state space given a finite $\mathbb{T}^\#$ is finite is nontrivial. However, since the *syntax* of the program constrains how C looks like, we can prove that:

Theorem 5.2 (Finiteness of time implies finiteness of abstraction)**.** If $\mathbb{T}^\#$ is finite,
$$\forall e, s^\# : |[\![e]\!]^\#(s^\#)| < \infty$$

6 SEPARATE ANALYSIS

6.1 Addition of time domains

For separate analysis, we need to define the linking operators in a way that soundly approximates the concrete version of linking. Note that in concrete linking, the time domains were linked based on the fact that the timestamps are ordered by a total order. Remember that the filtering operation determined whether the timestamp came *before* or *after* linking by comparing the first time component with the *final* time before linking. In the abstract semantics, such an approach is impossible, since the abstract timestamps do not preserve the order of the concrete timestamps. Thus, in the abstract semantics, the linked timestamp must live in $\mathbb{T}_1^\# + \mathbb{T}_2^\#$. The intuition is that the timestamps before linking and after linking is determined by their membership in each time domain.

$$\mathsf{filter}^\#_i(C^\#) \triangleq \begin{cases} [\,] & C^\# = [\,] \\ (x, t^\#) :: \mathsf{filter}^\#_i(C'^\#) & C^\# = (x, t^\#) :: C'^\# \wedge t^\# \in \mathbb{T}^\#_i \\ \mathsf{filter}^\#_i(C'^\#) & C^\# = (x, t^\#) :: C'^\# \wedge t^\# \notin \mathbb{T}^\#_i \\ (M, \mathsf{filter}^\#_i(C'^\#)) :: \mathsf{filter}^\#_i(C''^\#) & C^\# = (M, C'^\#) :: C''^\# \end{cases}$$

Fig. 10. Definition of the abstract filter operation ($i = 1, 2$).

Then the filtering operation for the context can naturally be defined as in Fig. 10, and the definition for the added time domain can be given.

Definition 6.1 (Addition of time domains)**.** Let $(\mathbb{T}^\#_1, \mathsf{tick}^\#_1)$ and $(\mathbb{T}^\#_2, \mathsf{tick}^\#_2)$ be two abstract time domains. Given $s^\#_1 = (C^\#_1, m^\#_1, t^\#_1) \in \mathrm{State}^\#\mathbb{T}^\#_1$, define the $\mathsf{tick}^\#_+(s^\#_1)$ function as:

$$\mathsf{tick}^\#_+(s^\#_1)(C^\#, m^\#, t^\#, x, v^\#) \triangleq \begin{cases} \mathsf{tick}^\#_1\ \mathsf{filter}^\#_1(C^\#, m^\#, t^\#, x, v^\#) & t^\# \in \mathbb{T}^\#_1 \\ \mathsf{tick}^\#_2\ \mathsf{filter}^\#_2(C^\#, m^\#, t^\#, x, v^\#\langle C^\#_1\rangle^{-1}) & t^\# \in \mathbb{T}^\#_2 \end{cases}$$

Then we call the abstract time $(\mathbb{T}^\#_1 + \mathbb{T}^\#_2, \mathsf{tick}^\#_+(s^\#_1))$ the linked time when $s^\#_1$ is exported.

Now the rest flows analogously to concrete linking. First the injection operator that injects the exported state to the next time must be defined.

Definition 6.2 (Injection of a configuration : $\rhd^\#$)**.**
Given $s^\# = (C^\#_1, m^\#_1, t^\#_1) \in \mathrm{State}^\#\mathbb{T}^\#_1$ and $r^\# = (V^\#_2, m^\#_2, t^\#_2) \in \mathrm{Result}^\#\mathbb{T}^\#_2$, let $s^\#\rhd^\# m^\#_2$ and $s^\#\rhd^\# r^\#$:

$$s^\#\rhd^\# m^\#_2 \triangleq \lambda t^\#. \begin{cases} m^\#_1(t^\#) & t^\# \in \mathbb{T}^\#_1 \\ m^\#_2(t^\#)\langle C^\#_1\rangle & t^\# \in \mathbb{T}^\#_2 \end{cases} \qquad s^\#\rhd^\# r^\# \triangleq (V^\#_2\langle C^\#_1\rangle, s^\#\rhd^\# m^\#_2, t^\#_2)$$

Furthermore, when $\ell^\# = (e, s'^\#) \in L^\#_2$, and $A^\# \subseteq (L^\#_2 \times R^\#_2) \cup R^\#_2$, we define:

$$s^\#\rhd^\#\ell^\# \triangleq (e, s^\# \rhd s'^\#) \qquad s^\#\rhd^\# A^\# \triangleq \{s^\#\rhd^\#\rho^\# | \rho^\# \in A^\#\} \cup \{s^\#\rhd^\#\ell^\# \rightsquigarrow^\# s^\#\rhd^\#\rho^\# | \ell^\# \rightsquigarrow^\# \rho^\# \in A^\#\}$$

Then in the same manner as concrete linking, we have that:

Lemma 6.1 (Injection preserves timestamps under added time)**.**

$$\forall s^\# \in \mathrm{State}^\#\mathbb{T}^\#_1, s'^\# \in \mathrm{State}^\#\mathbb{T}^\#_2 : s^\#\rhd^\#[\![e]\!]^\#(s'^\#) \subseteq [\![e]\!]^\#(s^\#\rhd^\# s'^\#)$$

We must also define the addition operator that recovers the semantics of the linked expression e_2 from the exported state $s_1{}^\#$ and the *separately* analyzed semantics of e_2.

Definition 6.3 (Addition between exported configurations and separately analyzed results)**.**
Let $s^\#_1$ be a configuration in $\mathbb{T}^\#_1$, and let $A^\#_2 = [\![e]\!]^\#(s'^\#)$ be the semantics of e under $(\mathbb{T}^\#_2, \mathsf{tick}^\#_2)$. Define the "addition" between $s^\#_1$ and $A^\#_2$ as:

$$s^\#_1 \oplus A^\#_2 \triangleq \mathsf{lfp}(\lambda X^\#.\mathsf{Step}^\#(X^\#) \cup (s^\#_1\rhd^\# A^\#_2))$$

Then because of the previous lemma, it is obvious that:

Lemma 6.2 (Addition of semantics equals semantics under added time)**.**

$$s^\# \oplus [\![e]\!]^\#(s'^\#) = [\![e]\!]^\#(s^\#\rhd^\# s'^\#)$$

6.2 Separating soundness

The only thing that remains is the formulation of soundness between $[\![e_1 \infty e_2]\!](s)$ under the linked time $(\underline{\mathbb{T}_1} \uplus \underline{\mathbb{T}_2}, \leq_+, \mathsf{tick}_+(s_1))$, when s_1 is the exported context, and the abstract semantics.

The tricky part is in the time $(t_1, 0_2)$. It is represented by *both* $\alpha_1(t_1) \in \mathbb{T}_1^\#$ and $\alpha_2(0_2) \in \mathbb{T}_2^\#$, when $\alpha_1 : \mathbb{T}_1 \to \mathbb{T}_1^\#$ and $\alpha_2 : \mathbb{T}_2 \to \mathbb{T}_2^\#$ are tick-approximating. Therefore, we cannot make a tick-approximating function between $\underline{\mathbb{T}_1} \uplus \underline{\mathbb{T}_2}$ and $\mathbb{T}_1^\# + \mathbb{T}_2^\#$. Instead, we define a function $\alpha_+ : \underline{\mathbb{T}_1} \uplus \underline{\mathbb{T}_2} \to \mathbb{T}_1^\# + \mathbb{T}_2^\#$ by using α_1 and α_2 which is not tick-approximating on the whole domain but is sound for all timestamps $t = (t_1, \_)$. That is, we will define α_+ so that the following holds:

$$\forall t \in \mathbb{T}_2, C, m, x, v : \alpha_+(\mathsf{tick}_+\ C\ m\ (t_1, t)\ x\ v) = \mathsf{tick}_+^\#\ \alpha_+(C)\ \alpha_+(m)\ \alpha_+(t_1, t)\ x\ \alpha_+(v)$$

Fortunately, such an α_+ is easy to find.

Lemma 6.3 (Linked abstraction)**.** Let $s_1 = (C_1, m_1, t_1) \in \mathsf{State}\mathbb{T}_1$, and let $\alpha_1 : \mathbb{T}_1 \to \mathbb{T}_1^\#$. Also, let $\alpha_2 : \mathbb{T}_2 \to \mathbb{T}_2^\#$ be a tick-approximating abstraction. Now define $\alpha_+ : \underline{\mathbb{T}_1} \uplus \underline{\mathbb{T}_2} \to \mathbb{T}_1^\# + \mathbb{T}_2^\#$ as:

$$\alpha_+(t) \triangleq \begin{cases} \alpha_1(t.1) & t \in \underline{\mathbb{T}_1} \\ \alpha_2(t.2) & t \in \underline{\mathbb{T}_2} \end{cases}$$

Then α_+ is tick-approximating on $\underline{\mathbb{T}_2}$ between $(\underline{\mathbb{T}_1} \uplus \underline{\mathbb{T}_2}, \leq_+, \mathsf{tick}_+(s_1))$ and $(\mathbb{T}_1^\# + \mathbb{T}_2^\#, \mathsf{tick}_+^\#(\alpha_1(s_1)))$.

Since we gave up tick-approximation for the times before linking, we need to *separate* the problem of finding a sound approximation of $[\![e_1]\!](s)$ and finding a sound approximation of $[\![e_2]\!](\mathsf{Exp})$.

Finding a sound approximation of $[\![e_1]\!](s)$ is easy. From the results of the previous section, if we have a tick-approximating α_1 between $\mathbb{T}_1$ and $\mathbb{T}_1^\#$, $[\![e_1]\!]^\#(\alpha_1(s))$ is automatically a sound approximation. The problem of finding a sound approximation for $[\![e_2]\!](\mathsf{Exp})$ is also easy if we have a sound approximation $\mathsf{Exp}^\#$ of Exp that satisfies $\alpha_1(\mathsf{Exp}) \subseteq \mathsf{Exp}^\#$. Since $\alpha_1(s) \in \mathsf{Exp}^\#$ for all $s \in \mathsf{Exp}$, if we merge $s^\# \oplus [\![e_2]\!]^\#(0_2^\#)$ for all $s^\# \in \mathsf{Exp}^\#$, $\alpha_+([\![e_2]\!](\mathsf{Exp}))$ will be contained in the merged cache. That is, if we write $\mathsf{Exp}^\# \oplus A^\# \triangleq \bigcup_{s^\# \in \mathsf{Exp}^\#} s^\# \oplus A^\#$, we have:

Lemma 6.4 (Separation of soundness)**.** Given $s \in \mathsf{State}\mathbb{T}_1$ and $\mathsf{Exp}^\# \subseteq \mathsf{State}^\#\mathbb{T}_1^\#$, assume:

- There exists an $\alpha_1 : \mathbb{T}_1 \to \mathbb{T}_1^\#$ satisfying $\alpha_1(s) \in \mathsf{Exp}^\#$.
- There exists a time-approximating $\alpha_2 : \mathbb{T}_2 \to \mathbb{T}_2^\#$.

Then $\alpha_+([\![e_2]\!](s \rhd 0_2)) \subseteq \mathsf{Exp}^\# \oplus [\![e_2]\!]^\#(0_2^\#)$.

6.3 Soundness of separate analysis

Now, we may define the abstract linking operator that soundly approximates the concrete linking operator, using the same notation as in concrete linking.

Definition 6.4 (Abstract linking operator)**.** Given $e_1, e_2, s^\#$, let $\mathsf{Exp}^\# = \{s_1^\# \rhd^\# 0_2^\# | s_1^\# \in \underline{e_1}^\#(s^\#)\}$. Then:

$$\mathsf{Link}^\#\ e_1\ e_2\ s^\# \triangleq [\![e_1]\!]^\#(s^\#) \cup [\![e_2]\!]^\#(\mathsf{Exp}^\#) \cup (e_1 \infty e_2, s^\#) \rightsquigarrow^\# (\{(e_1, s^\#)\} \cup (e_2, \mathsf{Exp}^\#) \cup \underline{e_2}^\#(\mathsf{Exp}^\#))$$

Note that $[\![e_2]\!]^\#(\mathsf{Exp}^\#)$ can be computed by $\underline{e_1}^\#(s^\#) \oplus [\![e_2]\!]^\#(0_2^\#)$, hence the analysis is separate. Now we want to show that the abstract linking operation is a sound approximation of concrete linking. However, as emphasized in the previous subsection, the statement of soundness cannot be achieved through just a single concretization function. Since abstract linking approximates its concrete counterpart *separately*, we need to concretize the part *before* linking and *after* linking separately.

Theorem 6.1 (Abstract linking). Let $\mathbb{T}_i (i = 1, 2)$ be two concrete times, and let $\mathbb{T}_i^\# (i = 1, 2)$ be two abstract times. Let $\alpha_i : \mathbb{T}_i \to \mathbb{T}_i^\# (i = 1, 2)$ be tick-approximating, and let $s^\# = \alpha_1(s)$ approximate the initial state. Then, $\mathsf{Link}^\#\ e_1\ e_2\ s^\#$ is a sound approximation of $\mathsf{Link}\ e_1\ e_2\ s$. That is:

$$\mathsf{Link}\, e_1\, e_2\, s \subseteq \gamma_1(\llbracket e_1 \rrbracket^\#(s^\#)) \cup \gamma_+(\llbracket e_2 \rrbracket^\#(\mathsf{Exp}^\#) \cup (e_1 \propto e_2, s^\#) \rightsquigarrow^\# (\{(e_1, s^\#)\} \cup (e_2, \mathsf{Exp}^\#) \cup \underline{e_2}^\#(\mathsf{Exp}^\#)))$$

when the Galois pairs of α_1 and α_+, γ_1 and γ_+, are defined as in section 5.

All is fine for linking two expressions. The approximation for the exporting expression comes directly from the abstract semantics, and the approximation for the importing expression comes from linking the exporting set with the separately analyzed results. However, the above theorem is not strong enough for linking more than two expressions. This is because $\mathsf{Link}^\#\ e_1\ e_2\ s^\#$ does *not* equal $\llbracket e_1 \propto e_2 \rrbracket^\#(s^\#)$, as $\mathsf{tick}_+^\#$ cannot leap between $\mathbb{T}_1^\#$ and $\mathbb{T}_2^\#$. Thus, $\mathsf{Link}^\#\ e_1 \propto e_2\ e_3\ s^\#$ does not mean that the semantics for $e_1 \propto e_2$ is computed separately. Also, $\mathsf{Link}^\#\ e_1\ e_2 \propto e_3\ s^\#$ does not help much, since computing $\llbracket e_2 \propto e_3 \rrbracket^\#(0^\#)$ will be stuck before even reaching e_3. To clarify on how to link an *arbitrary* number of modules, we state the following theorem:

Theorem 6.2 (Compositionality). Given a sequence $\{e_n\}_{n \geq 0}$ and initial condition $s \in \mathrm{State}\mathbb{T}_0$,

- Let $\mathbb{T}_n^\#$ be abstract times connected with the concrete times by tick-approximating α_n.
- Let the linked expressions l_n be $l_0 \triangleq e_0$, $l_{n+1} \triangleq l_n \propto e_{n+1}$, and let t_n be the final time of $\llbracket l_n \rrbracket(s)$.
- Define the linked abstraction functions α_+^n as:

$$\alpha_+^0 \triangleq \alpha_0 \qquad \alpha_+^{n+1}(t) \triangleq \begin{cases} \alpha_+^n(t.1) & t.1 \neq t_n \\ \alpha_{n+1}(t.2) & t.1 = t_n \end{cases}$$

- Let $s^\# = \alpha_0(s)$, and define $\mathsf{Exp}_n^\#$ and $\mathsf{Imp}_n^\#$ as:

$$\mathsf{Imp}_0^\# \triangleq \llbracket e_0 \rrbracket^\#(s^\#) \qquad \mathsf{Exp}_0^\# \triangleq \{s_0^\# \rhd^\# 0_1^\# | s_0^\# \in \underline{e_0}^\#(s^\#)\} \qquad \mathsf{Exp}_n^\# \triangleq \{s_n^\# \rhd^\# 0_{n+1}^\# | s_n^\# \in \underline{e_n}^\#(\mathsf{Exp}_{n-1}^\#)\}$$

$$\mathsf{Imp}_{n+1}^\# \triangleq \llbracket e_{n+1} \rrbracket^\#(\mathsf{Exp}_n^\#) \cup (l_{n+1}, s^\#) \rightsquigarrow^\# (\{(l_n, s^\#)\} \cup (e_{n+1}, \mathsf{Exp}_n^\#) \cup \underline{e_{n+1}}^\#(\mathsf{Exp}_n^\#))$$

Then:

$$\llbracket l_n \rrbracket(s) \subseteq \bigcup_{i=0}^{n} \gamma_+^i(\mathsf{Imp}_i^\#)$$

What the above theorem means is that there exists a concrete tick function that can be covered separately by analyzing each component based only on the approximation of the exported context. The fact that the analysis $\mathsf{Imp}_n^\#$ can be computed without actually computing the final times t_n is why this analysis can be called separate.

REFERENCES