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.

Expression Identifier	x	$\in$	ExprVar	
Module Identifier	M	$\in$	ModVar	
Expression	e	$\in$	Expr	
Expression	e	$\rightarrow$	x	identifier, expression
		$\mid$	$\lambda x.e$	function
		$\mid$	$e\ e$	application
		$\mid$	$e!e$	linked expression
		$\mid$	ε	empty module
		$\mid$	M	identifier, module
		$\mid$	`let` $x\ e\ e$	let-binding, expression
		$\mid$	`let` $M\ e\ e$	let-binding, module

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 λM`:>` $S.e$ statically resolves the context when M is used in the body e, and $(e_1\ e_2)$`:>` 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 $(m_1!m_2)!e$. $m_1!m_2$ links a module with a module, and $(m_1!m_2)!e$ 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.

XXXX-XXXX/2023/7-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 reachability 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 returned result from the configuration.

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.

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

Fig. 2. Definition of the semantic domains.

Above, there are no constraints placed upon the set $\mathbb{T}$. Now we give the conditions that the concrete time domain must satisfy.

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

(1) $(\mathbb{T}, \leq)$ is a total order.
(2) $\mathsf{tick} : \mathrm{Ctx}(\mathbb{T}) \rightarrow \mathrm{Mem}(\mathbb{T}) \rightarrow \mathbb{T} \rightarrow \mathrm{ExprVar} \rightarrow \mathrm{Val}(\mathbb{T}) \rightarrow \mathbb{T}$ satisfies:

$$\forall t \in \mathbb{T} : t < \mathsf{tick}\ \_\ \_\ t\ \_\ \_$$

The time $\mathsf{tick}\ C\ m\ t\ x\ v$ is the time that is incremented when the value v is bound to a variable x at time t under context C and memory m.

Why must the tick function for the concrete time take in C, m, t, x, v? This is for the purpose of program analysis. For program analysis, the analysis 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 ExprVar, and the function that looks up the dynamic context bound to a ModVar M.

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

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

2.1.2 The Relation. Now we are in a position to define the one-step reachability relation. The relation $\rightsquigarrow$ relates $(e, C, m, t) \in \mathrm{Expr} \times \mathrm{State}(\mathbb{T})$ with either $(V, m, t) \in \mathrm{Result}(\mathbb{T})$ or (e', C', m', t'). Note that we constrain whether an expression returns v or C by the definition of $\rightsquigarrow$.

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.

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 $e_1 ! e_2$ is evaluated under 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 state s must be a *subset* of $(L \times R) \cup R$, when $L = \mathrm{Expr} \times \mathrm{State}(\mathbb{T})$ is the **l**eft-hand-side of $\rightsquigarrow$, and $R = L \cup \mathrm{Result}(\mathbb{T})$ is the **r**ight-hand-side. Later on, when we link a context in $\mathbb{T}_1$ with the semantics in $\mathbb{T}_2$, we shall write L_i for $\mathrm{Expr} \times \mathrm{State}(\mathbb{T}_i)$, and R_i for $L_i \cup \mathrm{Result}(\mathbb{T}_i)$. Also, we shall write ℓ for an element of L, and ρ for an element of R.

2.2.1 Formulating semantics in terms of a fixpoint. 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, $D = \wp((L \times R) \cup R)$ as defined previously. Defining the transfer function is straightforward from the definition of the reachability relation.

Definition 2.2 (Transfer function)**.** Given $A \subseteq (L \times R) \cup R$, define

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 4. The concrete one-step reachability relation.

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 2.3 (Concrete semantics)**.**

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

For convenience, we shall write $[\![e]\!](t)$ instead of $[\![e]\!]([\,], \varnothing, t)$.

3 CONCRETE LINKING

To prepare for modular analysis, we need to be able to paste together the semantics of e_1 and e_2 to obtain the semantics of $e_1!e_2$. Why do we need to define linking on the level of the concrete semantics? This is because for separate analysis, e_1 and e_2 must be analyzed in separate abstract time domains $\mathbb{T}_1^\#$ and $\mathbb{T}_2^\#$, which are sound approximations of *concrete* $\mathbb{T}_1$ and $\mathbb{T}_2$ domains. To elaborate, since the abstract tick must be a sound approximation of the concrete tick, for the analysis to be sound the *initial time* must approximate the concrete initial time. If a single abstract time and concrete time domain is used, it means that the analysis of e_2 must start from an abstract time that approximates the time that e_1 exports to e_2. This is against the idea of separate analysis; we want to analyze the behavior of e_2 without running e_1 beforehand. Therefore, we need to define a way to link the concrete time domains that allows the analysis of e_1 and e_2 to approximate the concrete execution without looking at the final time of e_1.

Now assume that we have computed $[\![e_1]\!](0_1)$ and $[\![e_2]\!](0_2)$, when 0_i are the initial times in $\mathbb{T}_i$. What we want to define is a tick_+ function on $\mathbb{T}_1 \times \mathbb{T}_2$ that:

(1) Increments $(0_1, 0_2)$ up to $(t_1, 0_2)$, when $t_1 \in \mathbb{T}_1$ is the final timestamp of $[\![e_1]\!](0_1)$.
(2) Increments $(t_1, 0_2)$ up to (t_1, t_2), when $t_2 \in \mathbb{T}_2$ is the largest timestamp that can be computed *without* knowing about what e_1 exports to e_2.

To satisfy the above constraints, the tick_+ function has to use the tick_1 function to increment the first timestamp when the first timestamp is less than t_1. Also, in the case that the first timestamp is greater or equal to t_1, the second timestamp is incremented by tick_2 under the context and memory that is *removed* of the exported context.

What does it mean that the exported context should be *removed*? The second timestamp is assumed to be incremented separately, using tick_2, under the empty context. However, under the linked time $\mathbb{T}_1 \times \mathbb{T}_2$, first $(e_1, [], \emptyset, (0_1, 0_2)) \rightsquigarrow (C_1, m_1, (t_1, 0_2))$ is computed, and only then is 0_2 incremented under C_1, m_1. For tick_+ to produce the same timestamps produced by tick_2 under the empty initial conditions, C_1 has to be dug out from the bottom of the stack and m_1 must be filtered out. Filtering out m_2 is easy to do; just ignore addresses with a timestamp different from t_1 in the first time component. Digging out C_2 from the stack needs more consideration.

To determine what parts of the stack should be deleted, we need to be able to describe what C will look like if it started from C_2, not from $[]$. That is, we need to determine the injection operator $C_1\langle C\rangle$ that satisfies: if $(e_2, 0_2) \rightsquigarrow^* (e, C, m, t)$, then $(e_2, C_1, \emptyset, 0_2) \rightsquigarrow^* (e, C_1\langle C\rangle, C_1\langle m\rangle, t)$ with tick_+, when $\rightsquigarrow^*$ is the reflexive and transitive closure of $\rightsquigarrow$. The notation $C_1\langle m\rangle$ means that the context C_1 is injected into all closures in m.

$$C_1[C_2] \triangleq \begin{cases} [] & C_2 = [] \\ (x,t) :: C_1[C'] & C_2 = (x,t) :: C' \\ (M, C_1\langle C'\rangle) :: C_1[C''] & C_2 = (M, C') :: C'' \end{cases} \qquad C_1\langle C_2\rangle \triangleq C_1[C_2] +\!\!+ C_1$$

Fig. 5. Definition of the injection operator $C_1\langle C_2\rangle$.

The definition for the injection operator in our simple language is more complicated than expected. This is because when modules are bound to module identifiers under some imported context, the context that is bound *automatically* includes that imported context. This is a consequence of the design choice to exclude signatures. In a language where signatures designate what variable bindings should be exported, the definition of the injection operator would be simply the list

append operator $+\!\!+$. However, in our language when the injection has to *map* over all module bindings, the injection operator is defined in a mutually recursive manner; one that maps the injection over all bindings($C_1[C_2]$) and one that actually appends the stacks together($C_1\langle C_2\rangle$).

Now, the definition for the deletion operator that *digs out* the exported context can be naturally defined as the inverse operations of injection.

$$C_2 \overline{+\!\!+} C_1 \triangleq \begin{cases} C_2' \overline{+\!\!+} C_1' & (C_1, C_2) = (C_1' +\!\!+ [(x,t)], C_2' +\!\!+ [(x,t)]) \\ C_2' \overline{+\!\!+} C_1' & (C_1, C_2) = (C_1' +\!\!+ [(M,C)], C_2' +\!\!+ [(M,C)]) \\ C_2 & \text{otherwise} \end{cases}$$

$$C_1\overline{[C_2]} \triangleq \begin{cases} [\,] & C_2 = [\,] \\ (x,t) :: C_1\overline{[C']} & C_2 = (x,t) :: C' \\ (M, C_1\overline{\langle C'\rangle}) :: C_1\overline{[C'']} & C_2 = (M,C') :: C'' \end{cases} \qquad C_1\overline{\langle C_2\rangle} \triangleq C_1\overline{[C_2 \overline{+\!\!+} C_1]}$$

Fig. 6. Definition of the deletion operators.

The deletion operators satisfy $(C_2 +\!\!+ C_1)\overline{+\!\!+}C_1 = C_2$, $C_1\overline{[C_1[C_2]]} = C_2$, and $C_1\overline{\langle C_1\langle C_2\rangle\rangle} = C_2$.

Now only the filter operation has to be defined for the total definition of the tick_+ function. Note that the linked time domain only uses timestamps of the form $(t, 0_2)$ and (t_1, t). Therefore, we are actually considering a linked time on a *subset* of $\mathbb{T}_1 \times \mathbb{T}_2$. For notational convenience, we write this subset as $\underline{\mathbb{T}_1} \uplus \underline{\mathbb{T}_2}$, when $\underline{\mathbb{T}_1} \triangleq (\mathbb{T}_1 - \{t_1\}) \times \{0_2\}$ and $\underline{\mathbb{T}_2} \triangleq \{t_1\} \times \mathbb{T}_2$. The filter operation in $\underline{\mathbb{T}_1} \uplus \underline{\mathbb{T}_2}$ must determine whether a timestamp is a member of $\underline{\mathbb{T}_1}$ or $\underline{\mathbb{T}_2}$, and project the timestamps onto the corresponding domains.

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

Fig. 7. Definition for the filter operations ($i = 1, 2$).

From now on, we assume that all $(C, m, t) \in \mathrm{State}(\mathbb{T})$ satisfy the property that timestamps in C and m are strictly less than t. Now we can give the definition of the linked tick function:

Definition 3.1 (Concrete linking of time domains)**.** Given $(\mathbb{T}_1, \leq_1, \mathsf{tick}_1)$ and $(\mathbb{T}_2, \leq_2, \mathsf{tick}_2)$,

- Define $\leq_+$ as the lexicographic order on $\mathbb{T}_1 \times \mathbb{T}_2$.
- Given $s_1 = (C_1, m_1, t_2) \in \mathrm{State}(\mathbb{T}_1)$, define the $\mathsf{tick}_+(s_1)$ function as:

$$\mathsf{tick}_+(s_1)(C, m, t, x, v) \triangleq \begin{cases} (\mathsf{tick}_1\ \mathsf{filter}_1(C, m, t, x, v), 0_2) & t \in \underline{\mathbb{T}_1} \\ (t_1, \mathsf{tick}_2\ \mathsf{filter}_2(C_1\overline{\langle C, m, t, x, v\rangle})) & t \in \underline{\mathbb{T}_2} \end{cases}$$

  when all timestamps in C_1 are lifted to $\underline{\mathbb{T}_1} \uplus \underline{\mathbb{T}_2}$.

Then we call the concrete time $(\underline{\mathbb{T}_1} \uplus \underline{\mathbb{T}_2}, \leq_+, \mathsf{tick}_+(s_1))$ the linked time when s_1 is exported.

We must prove that $\mathsf{tick}_+(s_1)$ indeed satisfies the properties described in the start of this section. To formulate this intuition, we need to extend the injection operator to an operator $\triangleright$ that injects a configuration from $\mathbb{T}_1$ to a result in $\mathbb{T}_2$.

Definition 3.2 (Injection of a configuration : $\triangleright$)**.**
Given $s = (C_1, m_1, t_1) \in \text{State}(\mathbb{T}_1)$ and $r = (V_2, m_2, t_2) \in \text{Result}(\mathbb{T}_2)$, define $s \triangleright m_2$ and $s \triangleright r$ as:

$$s \triangleright m_2 \triangleq \lambda t. \begin{cases} m_1(t.1) & t \in \underline{\mathbb{T}_1} \\ C_1\langle m_2(t.2)\rangle & t \in \underline{\mathbb{T}_2} \end{cases} \qquad s \triangleright r \triangleq (C_1\langle V_2\rangle, s \triangleright m_2, t_2)$$

assuming that all timestamps $t \in \mathbb{T}_1$ is lifted to $(t, 0_2)$ and all timestamps $t \in \mathbb{T}_2$ is lifted to (t_1, t).
Furthermore, when $\ell = (e, s') \in L_2$, and $A \subseteq (L_2 \times R_2) \cup R_2$, we define:

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

We shall omit s_1 and simply call the linked tick function as tick_+ when the exported state s_1 is clear from the context. More specifically, it is to be understood that $\mathsf{tick}_+(s_1)$ is used when computing $[\![e]\!](s_1 \triangleright s_2)$. Then we can prove that the tick_+ function is indeed *well-defined*.

Lemma 3.1 (Injection preserves timestamps under linked time)**.**

$$\forall s \in \text{State}(\mathbb{T}_1), s' \in \text{State}(\mathbb{T}_2) : s \triangleright [\![e]\!](s') \subseteq [\![e]\!](s \triangleright s')$$

Now, as promised, we present how to link the semantics to obtain the semantics under linked time. To streamline the presentation, we introduce more notation that captures what is needed when pasting together separate parts of a linked expression. First, we write $\underline{e}(s) \triangleq \{r | (e, s) \rightsquigarrow r\}$ to denote the final value (e, s) evalates to. Note that if e is a module, $\underline{e}(s)$ is the state that e exports. Second, we write $[\![e]\!](S) \triangleq \bigcup_{s \in S} [\![e]\!](s)$. This is used when e_2 imports $S = \underline{e_1}(s)$ from e_1. Third, we write $\ell \rightsquigarrow P \triangleq \{\ell, \ell \rightsquigarrow \rho | \rho \in P\}$. This is used when pasting together the initial configuration $\ell = (e_1!e_2, s)$ with the separately computed configurations. Finally, we write $(e, S) \triangleq \{(e, s) | s \in S\}$.

Definition 3.3 (Concrete linking operator)**.** Given e_1, e_2, s, let $\mathsf{Exp} = \{s_1 \triangleright 0_2 | s_1 \in \underline{e_1}(s)\}$. Then:

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

when all timestamps $t \in \mathbb{T}_1$ are lifted to $(t, 0_2)$.

Then the following result follows directly from the *definition* of the collecting semantics.

Theorem 3.1 (Concrete linking)**.** Given $s \in \text{State}(\mathbb{T}_1)$ and another time domain $\mathbb{T}_2$,

$$[\![e_1!e_2]\!](s) = \mathsf{Link}\ e_1\ e_2\ s$$

under tick_+.

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 $\mathbb{T}$.

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

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 $\mathsf{tick}^\#$ must simulate the tick function, so it must have the same type as tick.

Definition 4.2 (Abstract time)**.** $(\mathbb{T}^\#, \mathsf{tick}^\#)$ is an *abstract time* when $\mathsf{tick}^\# : \mathrm{Ctx}(\mathbb{T}^\#) \to \mathrm{Mem}^\#(\mathbb{T}^\#) \to \mathbb{T}^\# \to \mathrm{ExprVar} \to \mathrm{Val}(\mathbb{T}^\#) \to \mathbb{T}^\#$ 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

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

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.
- Extend α to a function on $\mathrm{Mem}(\mathbb{T}) \to \mathrm{Mem}^\#(\mathbb{T}^\#)$ by defining

$$\alpha(m) \triangleq \bigcup_{t \in \mathsf{dom}(m)} [\alpha(t) \mapsto \{\alpha(m(t))\}]$$

- Extend α to a function on $\mathrm{Result}(\mathbb{T}) \to \mathrm{Result}^\#(\mathbb{T}^\#)$ 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 $\rightsquigarrow^\#$.

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

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

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

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

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

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

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

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

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

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

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

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

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

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 $\alpha : \mathbb{T} \to \mathbb{T}^\#$,

$$\forall s \in \mathrm{State}(\mathbb{T}) : [\![e]\!](s) \subseteq \gamma([\![e]\!]^\#(\alpha(s)))$$

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.

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.

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

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

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:

$$\mathsf{tick}_+^\#(s_1^\#)(C^\#, m^\#, t^\#, x, v^\#) \triangleq \begin{cases} \mathsf{tick}_1^\#\ \mathsf{filter}_1^\#(C^\#, m^\#, t^\#, x, v^\#) & t^\# \in \mathbb{T}_1^\# \\ \mathsf{tick}_2^\#\ \mathsf{filter}_2^\#(C_1^\#\overline{\langle C^\#, m^\#, t^\#, x, v^\#\rangle}) & t^\# \in \mathbb{T}_2^\# \end{cases}$$

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 : $\triangleright^\#$)**.**
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 $s^\# \triangleright^\# m_2^\#$ and $s^\# \triangleright^\# r^\#$:

$$s^\# \triangleright^\# m_2^\# \triangleq \lambda t^\#. \begin{cases} m_1^\#(t^\#) & t^\# \in \mathbb{T}_1^\# \\ C_1^\#\langle m_2^\#(t^\#)\rangle & t^\# \in \mathbb{T}_2^\# \end{cases} \qquad s^\# \triangleright^\# r^\# \triangleq (C_1^\#\langle V_2^\#\rangle, s^\# \triangleright^\# m_2^\#, t_2^\#)$$

Furthermore, when $\ell^\# = (e, s'^\#) \in L_2^\#$, and $A^\# \subseteq (L_2^\# \times R_2^\#) \cup R_2^\#$, we define:

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

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

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

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

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:

$$s_1^\# \oplus A_2^\# \triangleq \mathsf{lfp}(\lambda X^\#.\mathsf{Step}^\#(X^\#) \cup (s_1^\# \triangleright^\# A_2^\#))$$

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

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

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

6.2 Separating soundness

The only thing that remains is the formulation of soundness between $[\![e_1 ! e_2]\!](s)$ 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 $s_1 = (C_1, m_1, t_1) \in \mathrm{State}(\mathbb{T}_1)$, 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 $\alpha_+([\![e_2]\!](s \triangleright 0_2)) \subseteq \mathsf{Exp}^\# \oplus [\![e_2]\!]^\#(0_2^\#)$.

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 $\mathsf{Exp}^\# = \{s_1^\# \triangleright^\# 0_2^\# | s_1^\# \in \underline{e_1}^\#(s^\#)\}$. Then:

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

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:

$$\mathsf{Link}\ e_1\ e_2\ s \subseteq \gamma_1([\![e_1]\!]^\#(s^\#)) \cup \gamma_+([\![e_2]\!]^\#(\mathsf{Exp}^\#) \cup (e_1!e_2, s^\#) \rightsquigarrow^\# (\{(e_1, s^\#)\} \cup (e_2, \mathsf{Exp}^\#) \cup \underline{e_2}^\#(\mathsf{Exp}^\#)))$$

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 $[\![e_1!e_2]\!]^\#(s^\#)$, as $\mathsf{tick}_+^\#$ cannot leap between $\mathbb{T}_1^\#$ and $\mathbb{T}_2^\#$. Thus, $\mathsf{Link}^\#\ e_1!e_2\ e_3\ s^\#$ does not mean that the semantics for $e_1!e_2$ is computed separately. Also, $\mathsf{Link}^\#\ e_1\ e_2!e_3\ s^\#$ does not help much, since computing $[\![e_2!e_3]\!]^\#(0^\#)$ 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 $s \in \mathsf{State}(\mathbb{T}_0)$,

- Let $\mathbb{T}_n^\#$ be abstract times connected with the concrete times by tick-approximating α_n.
- Let the linked expressions l_n be $l_0 \triangleq e_0$, $l_{n+1} \triangleq l_n!e_{n+1}$, and let t_n be the final time of $[\![l_n]\!](s)$.

- 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:

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

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

Then:

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

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