A Syntax-Guided Framework for Modular Analysis

JOONHYUP LEE

ACM Reference Format:
Joonhyup Lee. 2023. A Syntax-Guided Framework for Modular Analysis. 1, 1 (July 2023), 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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
		$\mid$	$\texttt{let}\ x\ e\ e$	let-binding, expression
		$\mid$	$\texttt{let}\ M\ e\ e$	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 $\lambda M\texttt{:>}\ S.e$ 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.

Author's address: Joonhyup Lee.

XXXX-XXXX/2023/7-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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.

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 big-step operational semantics for the dynamic execution of the module language. The big-step evaluation relation relates the initial configuration(context, memory and time) and expression with the resulting value and state.

Note that the representation of the *environment* that is often used to define closures in the call-by-value dynamics is not simply a finite map from variables to addresses. Rather, the environment is a stack that records variables *in the order* they were bound. In the spirit of de Bruijn, to access the value of the variable x from the environment(or the *binding context*) C, one has to read off the closest binding time. Then, the value bound at that time from the memory is read. Likewise, to access the exported context from the 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 the evaluated values from expressions 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, mainly the *value*, *memory*, and *context* domains. Also, we present the notational conventions used in this paper to represent members of each domain.

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

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) $\mathsf{tick} \in \mathbb{T} \rightarrow \mathbb{T}$ satisfies: $\forall t \in \mathbb{T} : t < \mathsf{tick}\ t$.

Now for the auxiliary operators that is used when defining the evaluation relation. We define the 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 Evaluation Relation. Now we are in a position to define the big-step evaluation relation. The relation $\Downarrow$ relates $(e, C, m, t) \in \text{Expr} \times \text{State}(\mathbb{T})$ with $(V, m, t) \in \text{Result}(\mathbb{T})$. Note that we constrain whether the evaluation relation returns $v \in \text{Val}(\mathbb{T})$(when the expression being evaluated is not a module) or $C \in \text{Ctx}(\mathbb{T})$ by the definition of the relation.

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

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

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

$$[\textsc{Empty}]\ \frac{}{(\varepsilon, C, m, t) \Downarrow (C, m, t)} \qquad [\textsc{ModVar}]\ \frac{C' = \mathsf{ctx}(C, M)}{(M, C, m, t) \Downarrow (C', m, t)}$$

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

Fig. 4. The concrete big-step evaluation relation.

Note that we do not constrain whether v or C is returned by e_2 in the linking case. That is, linking may return either values or modules.

The equivalence of the evaluation relation with a reference interpreter is formalized in Coq.

2.1.3 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* pairs of $(e, s) \Downarrow r$ that appear 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 configuration 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 of e_2.

What should be the *type* of the collecting semantics? Obviously, given the type of the evaluation relation, $\wp((\text{Expr} \times \text{State}(\mathbb{T})) \times \text{Result}(\mathbb{T}))$ seems to be the natural choice. However, by requiring that all collected pairs have a result fails to collect the configurations that are reached but does not return. Such a situation will occur frequently when separately analyzing an expression that depends on an external module to resolve its free variables. Therefore, we extend the relation to relate an element of $\text{Expr} \times \text{State}(\mathbb{T})$ to a *set* of results. An (e, s) that is not related to any set means that the configuration is not reached, and an (e, s) that is related to an empty set means that the configuration does not return.

Definition 2.2 (Collecting Semantics)**.** The collecting semantics of an expression e under initial configuration s is a set $[\![e]\!](s) \subseteq (\text{Expr} \times \text{State}(\mathbb{T})) \times \wp(\text{Result}(\mathbb{T}))$ that collects all pairs of reachable configurations with the set of results they return.

2.2 Fixpoint Semantics

The collecting semantics in the previous section was defined in a declarative style and does not provide *how* to actually calculate the semantics. To formalize the notion of reachability and to utilize this in proofs, we define the single-step reachability relation $\rightsquigarrow$ in 5.

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

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

$$[\text{AppBody}]\ \frac{(e_1, C, m, t) \Downarrow (\langle \lambda x.e_\lambda, C_\lambda \rangle, m_\lambda, t_\lambda) \quad (e_2, C, m_\lambda, t_\lambda) \Downarrow (v, m_a, t_a)}{(e_1\ e_2, C, m, t) \rightsquigarrow (e_\lambda, (x, t_a) :: C_\lambda, m_a[t_a \mapsto v], \text{tick}\ t_a)}$$

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

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

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

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

Fig. 5. The concrete single-step reachability relation.

The well-definedness of the reachability relation with respect to a reference interpreter is formalized in Coq.

Using the transitive and reflexive closure $\rightsquigarrow^*$ of $\rightsquigarrow$, we can formalize the notion of collecting semantics.

Lemma 2.1 (Collecting semantics using $\rightsquigarrow^*$)**.**

$$[\![e]\!](s) = \bigcup_{(e,s) \rightsquigarrow^* (e',s')} \{((e', s'), \{r | (e', s') \Downarrow r\})\}$$

Now we define a transfer function that takes a cache and returns a cache that takes one step further. Using this transfer function, we will express the collecting semantics as a fixpoint, which directly shows how to compute the semantics.

Definition 2.3 (Transfer function). Given a subset a of $(\text{Expr} \times \text{State}(\mathbb{T})) \times \wp(\text{Result}(\mathbb{T}))$,

- Define $\Downarrow_a$ and $\rightsquigarrow_a$ by replacing all premises $(e, s) \Downarrow r$ by $\exists R : ((e, s), R) \in a \land r \in R$ in $\Downarrow$ and $\rightsquigarrow$.
- Define the step function that collects all results derivable in one step from (e, s) using a.

$$\mathsf{step}(a)(e, s) \triangleq \{((e, s), \{r | (e, s) \Downarrow_a r\})\} \cup \bigcup_{(e,s) \rightsquigarrow_a (e',s')} \{((e', s'), \varnothing)\}$$

We define the transfer function Step by:

$$\mathsf{Step}(a) \triangleq \bigcup_{(e,s) \in \mathsf{dom}(a)} \mathsf{step}(a)(e, s)$$

when the set $\mathsf{dom}(a) \triangleq \{(e, s) | \exists R \subseteq \text{Result}(\mathbb{T}) : ((e, s), R) \in a\}$.

We can finally formulate the collecting semantics in fixpoint form.

Lemma 2.2 (Concrete semantics as a fixpoint).

$$[\![e]\!](s) = \mathsf{lfp}(\lambda a.\mathsf{Step}(a) \cup \{((e, s), \varnothing)\}$$

3 CONCRETE LINKING

To justify separate analysis, we decompose the collecting semantics of the linking expression into a composition of the semantics of the left and right expressions.

Definition 3.1 (Auxiliary operators for concrete linking).

$$\begin{aligned}
\mathsf{Exp}\ e_1\ s &\triangleq \bigcup_{((e_1,s),R) \in [\![e_1]\!](s)} R && \text{(Exported under } s\text{)} \\
\mathsf{L}\ E\ e_2 &\triangleq \bigcup_{s' \in E} [\![e_2]\!](s') && \text{(Reached under } E\text{)} \\
\mathsf{F}\ E\ e_2 &\triangleq \bigcup_{s' \in E} \bigcup_{((e_2,s'),R) \in [\![e_2]\!](s')} R && \text{(Final results under } E\text{)}
\end{aligned}$$

The intuition is, when linking e_1 and e_2 under initial configuration s, first e_1 is computed, then exports(Exp) its results to e_2, which e_2 is linked(L) with. The final result for the total expression $e_1!e_2$ will be the final result(F) of e_2 under the exported context.

Definition 3.2 (Concrete linking operator).

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

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

Theorem 3.1 (Concrete linking).

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

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 $\mathbb{T}$.

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

Fig. 6. Definition of the semantic domains.

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

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

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

$$
[\textsc{Linking}]\ \frac{\begin{array}{c}(e_1, C^\#, m^\#, t^\#) \Downarrow^\# (C'^\#, m'^\#, t'^\#) \\ (e_2, C'^\#, m'^\#, t'^\#) \Downarrow^\# (V^\#, m''^\#, t''^\#)\end{array}}{(e_1 ! e_2, C^\#, m^\#, t^\#) \Downarrow^\# (V^\#, m''^\#, t''^\#)}
$$

$$
[\textsc{Empty}]\ \frac{}{(\varepsilon, C^\#, m^\#, t^\#) \Downarrow^\# (C^\#, m^\#, t^\#)}
\qquad
[\textsc{ModVar}]\ \frac{C'^\# = \mathsf{ctx}(C^\#, M)}{(M, C^\#, m^\#, t^\#) \Downarrow^\# (C'^\#, m^\#, t^\#)}
$$

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

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

Fig. 7. The abstract big-step evaluation relation.

4.1 Big-Step Evaluation

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

Definition 4.1 (Weak update)**.** Given $m^\# \in \mathrm{Mem}^\#(\mathbb{T}^\#)$, $t^\# \in \mathbb{T}^\#$, $v^\# \in \mathrm{Val}(\mathbb{T}^\#)$, we 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. Since we want to utilize the information when the binding is performed, the $\mathsf{tick}^\#$ function takes in more information than simply the previous abstract time.

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

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

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

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

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

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

$$[\textsc{LetML}]\ \frac{}{(\texttt{let}\ M\ e_1\ e_2, C^\#, m^\#, t^\#) \rightsquigarrow^\# (e_1, C^\#, m^\#, t^\#)}$$

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

Fig. 8. The abstract single-step reachability relation.

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

In our semantics, $\mathsf{tick}^\#\ C^\#\ m^\#\ t^\#\ x\ v^\#$ is performed when $v^\#$ is bound to x under configuration $(C^\#, m^\#, t^\#)$.

The abstract big-step evaluation relation is defined in 7, and the single-step reachability relation is defined in 8.

From the relations above, we can define the abstract semantics:

Definition 4.3 (Abstract semantics). The semantics for an expression e under configuration $s^\# \in \mathrm{State}^\#(\mathbb{T}^\#)$ is a subset of $(\mathrm{Expr} \times \mathrm{State}^\#(\mathbb{T}^\#)) \times \wp(\mathrm{Result}^\#(\mathbb{T}^\#))$ defined as:

$$[\![e]\!]^\#(s^\#) \triangleq \bigcup_{(e, s^\#) \rightsquigarrow^{\#*} (e', s'^\#)} \{((e', s'^\#), \{r^\# | (e', s'^\#) \Downarrow^\# r^\#\})\}$$

As in the concrete case, the semantics need to be expressed in fixpoint form.

Definition 4.4 (Transfer function). Given a subset $a^\#$ of $(\mathrm{Expr} \times \mathrm{State}^\#(\mathbb{T}^\#)) \times \wp(\mathrm{Result}^\#(\mathbb{T}^\#))$,

- Define $\Downarrow^\#_{a^\#}$ and $\rightsquigarrow^\#_{a^\#}$ by replacing all premises $(e, s^\#) \Downarrow^\# r^\#$ by $\exists R^\# : ((e, s^\#), R^\#) \in a^\# \wedge r^\# \in R^\#$ in $\Downarrow^\#$ and $\rightsquigarrow^\#$.
- Define the $\mathsf{step}^\#$ function that collects all results derivable in one step from $(e, s^\#)$ using $a^\#$.

$$\mathsf{step}^\#(a^\#)(e, s^\#) \triangleq \{((e, s^\#), \{r^\# | (e, s^\#) \Downarrow^\#_{a^\#} r^\#\})\} \cup \bigcup_{(e, s^\#) \rightsquigarrow^\#_{a^\#} (e', s'^\#)} \{((e', s'^\#), \varnothing)\}$$

We define the transfer function $\mathsf{Step}^\#$ by:

$$\mathsf{Step}^\#(a^\#) \triangleq \bigcup_{(e,s^\#)\in\mathsf{dom}(a^\#)} \mathsf{step}^\#(a^\#)(e,s^\#)$$

Lemma 4.1 (Abstract semantics as a fixpoint).

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

5 NON-MODULAR ANALYSIS

Definition 5.1 (Tick-approximating concretization). Given a concrete time $(\mathbb{T},\leq,\mathsf{tick})$ and an abstract time $(\mathbb{T}^\#,\mathsf{tick}^\#)$, a function $\gamma:\mathbb{T}^\#\to\wp(\mathbb{T})$ is said to be *tick-approximating* if:

$$\forall t,t^\#,C^\#,m^\#,x,v^\#: t\in\gamma(t^\#)\Rightarrow \mathsf{tick}\ t\in\gamma(\mathsf{tick}^\#\ C^\#\ m^\#\ t^\#\ x\ v^\#)$$

Then we can prove that:

Lemma 5.1 (Preservation of soundness).

- Let $s\in\mathsf{State}(\mathbb{T})$ and $s^\#\in\mathsf{State}^\#(\mathbb{T}^\#)$.
- Let all timestamps in the C and m component of s be strictly less than the t component.
- Let γ be tick-approximating, and define $\alpha(t)\triangleq\{t^\#|t\in\gamma(t^\#)\}$

Then for all e,

$$s\in\gamma(s^\#)\Rightarrow[\![e]\!](s)\subseteq\gamma\left(\bigcup_{s^\#\in\alpha(s)}[\![e]\!]^\#(s^\#)\right)$$

What's remarkable is that we did not put any constraint on the tick and $\mathsf{tick}^\#$ functions. Moreover, we can guarantee that $[\![e]\!]^\#(s^\#)$ can be computed.

Theorem 5.1 (Finiteness of time implies finiteness of abstraction). If $\mathbb{T}^\#$ is finite,

$$\forall e,s^\#: |[\![e]\!]^\#(s^\#)|<\infty$$

6 MODULAR ANALYSIS

For separate analysis, we analyze the two components e_1 and e_2 separately to eventually get a sound overapproximation of $e_1!e_2$. This means that we have to define an abstract time on $\mathbb{T}_1^\#+\mathbb{T}_2^\#$, when the first time domain exports $s'^\#$ to the second time domain.

Before elaborating on how to add the time domains, we need to define the injection operator that inject the exported context into the separately analyzed results. The notation for injecting C_1 into C_2 is $C_1\langle C_2\rangle$, similar to the plugin operator defined above.

Also, we need to define the deletion operator satisfying $\mathsf{delete}(C,C\langle C'\rangle)=C'$ as to recover the separately analyzed context from the injected context. This operation is essential in defining the $\mathsf{tick}^\#$ function in the added time domain that conserves the timestamps created before injection.

We abuse the notation $C\langle v\rangle$ to mean that the context C is injected in the context part of the closure v, and we mean by $C\langle m^\#\rangle$ the memory where injection is mapped over all values.

Finally, before delving into the definition of the linked time domain, we need to define a filter function that filters the context and memory by membership in each time domain.

Definition 6.1 (Injection of a configuration).

- Let $s^\#=({C_1}^\#,{m_1}^\#,{t_1}^\#)$ be an exported configuration from $\mathbb{T}_1^\#$.
- Let $r^\#=({V_2}^\#,{m_2}^\#,{t_2}^\#)$ be a result in $\mathbb{T}_2^\#$.

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

$$C_1\langle C_2\rangle \triangleq \mathsf{map\_inject}(C_1, C_2) +\!\!+ C_1$$

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

$$\mathsf{delete\_map}(C_1, C_2) \triangleq \begin{cases} [\,] & C_2 = [\,] \\ (x,t) :: \mathsf{delete\_map}(C_1, C') & C_2 = (x,t) :: C' \\ (M, \mathsf{delete\_map}(C_1, \mathsf{delete\_app}(C_1, C'))) :: \mathsf{delete\_map}(C_1, C'') & C_2 = (M, C') :: C'' \end{cases}$$

$$\mathsf{delete}(C_1, C_2) \triangleq \mathsf{delete\_map}(C_1, \mathsf{delete\_app}(C_1, C_2))$$

Fig. 9. Definitions for the injection and deletion operators.

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

$$\mathsf{filter}(v, \mathbb{T}) \triangleq \quad \langle \lambda x.e, \mathsf{filter}(C, \mathbb{T})\rangle \quad (v = \langle \lambda x.e, C\rangle)$$

$$\mathsf{filter}(m^\#, \mathbb{T}) \triangleq \lambda t \in \mathbb{T}.\{\mathsf{filter}(v, \mathbb{T}) | v \in m^\#(t)\}$$

Fig. 10. Definitions for the filter operation.

Define $s^\# \rhd r^\# \triangleq ({C_1}^\#\langle {V_2}^\#\rangle, {C_1}^\#\langle {m_2}^\#\rangle \sqcup {m_1}^\#, {t_2}^\#)$ to be a result in $\mathbb{T}_1^\# + \mathbb{T}_2^\#$.
We extend the $\rhd$ operator to inject $s^\#$ in an element of $(\text{Expr} \times \text{State}^\#(\mathbb{T}^\#)) \times \wp(\text{Result}^\#(\mathbb{T}^\#))$:

$$s^\# \rhd a^\# \triangleq \bigcup_{(e, s'^\#) \in \mathsf{dom}(a^\#)} \{((e, s^\# \rhd s'^\#), \{s^\# \rhd r^\# | r^\# \in a^\#(e, s'^\#)\})\}$$

Definition 6.2 (Addition of time domains)**.**

- Let ${s_1}^\# = ({C_1}^\#, {m_1}^\#, {t_1}^\#)$ be a configuration in $\mathbb{T}_1^\#$, and let $(\mathbb{T}_2^\#, \mathsf{tick}^\#)$ be an abstract time.
- Define the $\mathsf{tick}_+{}^\#({s_1}^\#)$ function as:

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

Then we call the abstract time $(\mathbb{T}_1^\# + \mathbb{T}_2^\#, \mathsf{tick}_+^\#({s_1}^\#))$ the linked time of $(\mathbb{T}_2^\#, \mathsf{tick}^\#)$ under exported configuration ${s_1}^\#$.

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

Let $s^\#$ be a configuration in $\mathbb{T}_1^\#$, $[\![e]\!]^\#(s'^\#)$ be the semantics of e under $(\mathbb{T}_2^\#, \mathsf{tick}^\#)$, and $[\![e]\!]^\#(s^\# \rhd s'^\#)$ be the semantics of e under $(\mathbb{T}_1^\# + \mathbb{T}_2^\#, \mathsf{tick}_+^\#(s^\#))$. Then:

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

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 $(\mathbb{T}^\#_2, \mathsf{tick}^\#)$. Define the "addition" between ${s_1}^\#$ and ${a_2}^\#$ as:

$$ {s_1}^\# \oplus {a_2}^\# \triangleq \mathsf{lfp}(\lambda a^\#.\mathsf{Step}^\#(a^\#) \cup ({s_1}^\# \rhd {a_2}^\#)) $$

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

$$ {s_1}^\# \oplus [\![e_2]\!]^\#(s'^\#) = [\![e]\!]^\#(s^\# \rhd s'^\#) $$

We introduce a notation for convenience: for any e, we denote the semantics of e under an initial configuration with empty context and memory as $[\![e]\!]^\#_0$.

Definition 6.4 (Auxiliary operators for abstract linking).

$$ \begin{aligned} \mathsf{Exp}^\#\ a^\#\ e_1\ s^\# &\triangleq \bigcup_{((e_1,s^\#),R^\#)\in a^\#} R^\# && \text{(Exported under } s^\# \text{ using } a^\#\text{)} \\ \mathsf{L}^\#\ E^\#\ e_2 &\triangleq \bigcup_{s'^\#\in E^\#} (s'^\# \oplus [\![e_2]\!]^\#_0) && \text{(Reached under } E^\#\text{)} \\ \mathsf{F}^\#\ E^\#\ e_2 &\triangleq \bigcup_{s'^\#\in E^\#} \bigcup_{((e_2,C'^\#,m'^\#,0^\#),R^\#)\in(s'^\#\oplus[\![e_2]\!]^\#_0)} R^\# && \text{(Final results under } E^\#\text{)} \end{aligned} $$

Definition 6.5 (Abstract linking operator).

$$ \mathsf{Link}^\#\ a^\#\ e_1\ e_2\ s^\# \triangleq a^\# \cup \mathsf{L}^\#\ (\mathsf{Exp}^\#\ a^\#\ e_1\ s^\#)\ e_2 \cup \{((e_1!e_2, s^\#), \mathsf{F}^\#\ (\mathsf{Exp}^\#\ a^\#\ e_1\ s^\#)\ e_2)\} $$

Theorem 6.1 (Abstract linking). Let s be a concrete configuration.

Let $a^\#$ satisfy $[\![e_1]\!](s) \subseteq \gamma_1(a^\#)$, and define $\alpha_1(t) \triangleq \{t^\# | t \in \gamma_1(t^\#)\}$.

Let γ_2 be a tick-approximating concretization between $\mathbb{T}$ and $\mathbb{T}^\#_2$. Then:

$$ [\![e_1!e_2]\!](s) \subseteq (\gamma_1 \cup \gamma_2)\left(\bigcup_{s^\#\in\alpha_1(s)} \mathsf{Link}^\#\ a^\#\ e_1\ e_2\ s^\#\right) $$

Why did we introduce $a^\#$, instead of using $[\![e_1]\!]^\#(s^\#)$ directly for the overapproximation of $[\![e_1]\!](s)$?

Theorem 6.2 (Compositionality of abstract linking).

Let $\{e_i\}_{i\geq 0}$ be a sequence of expressions and let s be a concrete configuration. Define $\{l_i\}_{i\geq 0}$ as:

$$ l_0 \triangleq e_0 \qquad l_{i+1} \triangleq l_i!e_{i+1} $$

and define $L_i \triangleq [\![l_i]\!](s)$.

Let γ_i be a tick-approximating concretization between $\mathbb{T}$ and $\mathbb{T}^\#_i$, and let the union of γ_i from $i = 0$ to n be $l\gamma_n$. Now, define $\alpha_0(t) \triangleq \{t^\# | t \in \gamma_0(t^\#)\}$ and

$$ {L_0}^\# \triangleq \bigcup_{s^\#\in\alpha_0(s)} [\![e_0]\!]^\#(s^\#) \qquad {L_{i+1}}^\# \triangleq \bigcup_{s^\#\in\alpha_0(s)} \mathsf{Link}^\#\ {L_i}^\#\ l_i\ e_{i+1}\ s^\# $$

Then we have:

$$ \forall n : L_n \subseteq l\gamma_n({L_n}^\#) $$