

# 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), 11 pages. <https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

## 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
			$\lambda x.e$	function
			$e e$	application
			$e!e$	linked expression
			$\varepsilon$	empty module
			$M$	identifier, module
			$\text{let } x e e$	let-binding, expression
			$\text{let } M e e$	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  $\lambda M :> S.e$  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.

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,

---

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](mailto:permissions@acm.org).

© 2023 Association for Computing Machinery.

XXXX-XXXX/2023/7-ART \$15.00

<https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

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 STRUCTURAL OPERATIONAL SEMANTICS

In this section 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.

This relation is nonstandard in that the *environment* that is often used to define closures in the call-by-value dynamics is not a finite map from variables to values. Rather, the surrounding *syntactic* context annotated with the *binding times* for the variables serve as the environment. To access the value of the variable  $x$  from the context  $C$ , one has to read off the closest binding time from the context and look up the value bound at that time from the memory. 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 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.

Time	$t$	$\in$	$\mathbb{T}$	
Context	$C$	$\in$	$\text{Ctx}$	
Result of expressions	$v$	$\in$	$\text{Val} \triangleq \text{Expr} \times \text{Ctx}$	
Result of expressions/modules	$V$	$\in$	$\text{Val} + \text{Ctx}$	
Memory	$m$	$\in$	$\text{Mem} \triangleq \mathbb{T} \xrightarrow{\text{fin}} \text{Val}$	
Configuration	$s$	$\in$	$\text{Config} \triangleq \text{Ctx} \times \text{Mem} \times \mathbb{T}$	
Result	$r$	$\in$	$\text{Result} \triangleq (\text{Val} + \text{Ctx}) \times \text{Mem} \times \mathbb{T}$	
Context	$C$	$\rightarrow$	$[\ ]$	hole
		$ $	$\lambda x^t.C$	function parameter binding
		$ $	$\text{let } x^t C$	let expression binding
		$ $	$\text{let } M C C$	let module binding
Result 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).  $(\mathbb{T}, \leq, \text{tick})$  is a *concrete time* when

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

Now for the auxiliary operators that is used when defining the evaluation relation. We define the plugin operator for the dynamic context, the function that extracts the address for an ExprVar, and the function that looks up the dynamic context bound to a ModVar  $M$ .

$$\begin{aligned}
C_1[C_2] &\triangleq \begin{cases} C_2 & C_1 = [] \\ \lambda x^t. C' [C_2] & C_1 = \lambda x^t. C' \\ \text{let } x^t C' [C_2] & C_1 = \text{let } x^t C' \\ \text{let } M C' C'' [C_2] & C_1 = \text{let } M C' C'' \end{cases} \\
\text{addr}(C, x) &\triangleq \begin{cases} \perp & C = [] \\ t & (C = \lambda x^t. C' \vee C = \text{let } x^t C') \wedge (x' = x \wedge \text{addr}(C', x) = \perp) \\ \text{addr}(C', x) & (C = \lambda x^t. C' \vee C = \text{let } x^t C') \wedge (x' \neq x \vee \text{addr}(C', x) \neq \perp) \\ \text{addr}(C'', x) & C = \text{let } M C' C'' \end{cases} \\
\text{ctx}(C, M) &\triangleq \begin{cases} \perp & C = [] \\ C' & C = \text{let } M' C' C'' \wedge (M' = M \wedge \text{ctx}(C'', M) = \perp) \\ \text{ctx}(C'', M) & C = \text{let } M' C' C'' \wedge (M' \neq M \vee \text{ctx}(C'', M) \neq \perp) \\ \text{ctx}(C', M) & C = \lambda x^t. C' \vee C = \text{let } x^t C' \end{cases}
\end{aligned}$$

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

## 2.2 The Evaluation Relation

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

$$\begin{aligned}
&\boxed{(e, C, m, t) \Downarrow (V, m', t')} \\
&\text{[EXPRVAR]} \frac{t_x = \text{addr}(C, x) \quad v = m(t_x)}{(x, C, m, t) \Downarrow (v, m, t)} \quad \text{[FN]} \frac{}{(\lambda x. e, C, m, t) \Downarrow ((\lambda x. e, C), m, t)} \\
&\text{[APP]} \frac{\begin{array}{c} (e_1, C, m, t) \Downarrow ((\lambda x. e_\lambda, C_\lambda), m_\lambda, t_\lambda) \\ (e_2, C, m_\lambda, t_\lambda) \Downarrow (v, m_a, t_a) \\ (e_\lambda, C_\lambda [\lambda x^{t_a}. []], m_a [t_a \mapsto v], \text{tick } t_a) \Downarrow (v', m', t') \end{array}}{(e_1 e_2, C, m, t) \Downarrow (v', m', t')} \quad \text{[LINKING]} \frac{\begin{array}{c} (e_1, C, m, t) \Downarrow (C', m', t') \\ (e_2, C', m', t') \Downarrow (V, m'', t'') \end{array}}{(e_1!e_2, C, m, t) \Downarrow (V, m'', t'')} \\
&\text{[EMPTY]} \frac{}{(\epsilon, C, m, t) \Downarrow (C, m, t)} \quad \text{[MODVAR]} \frac{C' = \text{ctx}(C, M)}{(M, C, m, t) \Downarrow (C', m, t)} \\
&\text{[LETE]} \frac{\begin{array}{c} (e_1, C, m, t) \Downarrow (v, m', t') \\ (e_2, C[\text{let } x^{t'} []], m' [t' \mapsto v], \text{tick } t') \Downarrow (C', m'', t'') \end{array}}{(\text{let } x e_1 e_2, C, m, t) \Downarrow (C', m'', t'')} \quad \text{[LETM]} \frac{\begin{array}{c} (e_1, C, m, t) \Downarrow (C', m', t') \\ (e_2, C[\text{let } M C' []], m', t') \Downarrow (C'', m'', t'') \end{array}}{(\text{let } M e_1 e_2, C, m, t) \Downarrow (C'', m'', t'')}
\end{aligned}$$

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.

## 2.3 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,  $\wp((\text{Expr} \times \text{Config}) \times \text{Result})$  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 interpret the relation as a partial function that maps an element of  $\text{Expr} \times \text{Config}$  to a *set* of results. An  $(e, s)$  that is not in the domain of the collecting semantics means that it is not *reached* by the initial configuration, and a  $(e, s)$  that is inside the domain but is mapped to  $\emptyset$  means that the configuration is reached but does not return.

**Definition 2.2** (Collecting Semantics). The collecting semantics of an expression  $e$  under initial configuration  $s$  is a partial function  $\llbracket e \rrbracket(s) \in (\text{Expr} \times \text{Config}) \rightarrow \wp(\text{Result})_{\perp}$  that collects all pairs of reachable configurations with the results they return.

### 3 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.

$$\begin{array}{c}
 \boxed{(e, C, m, t) \rightsquigarrow (e', C', m', t')} \\
 \text{[APPL]} \frac{}{(e_1 \ e_2, C, m, t) \rightsquigarrow (e_1, C, m, t)} \quad \text{[APPR]} \frac{(e_1, C, m, t) \Downarrow ((\lambda x. e_{\lambda}, C_{\lambda}), m_{\lambda}, t_{\lambda})}{(e_1 \ e_2, C, m, t) \rightsquigarrow (e_2, C, m_{\lambda}, t_{\lambda})} \\
 \text{[APPBODY]} \frac{(e_1, C, m, t) \Downarrow ((\lambda x. e_{\lambda}, C_{\lambda}), m_{\lambda}, t_{\lambda}) \quad (e_2, C, m_{\lambda}, t_{\lambda}) \Downarrow (v, m_a, t_a)}{(e_1 \ e_2, C, m, t) \rightsquigarrow (e_{\lambda}, C_{\lambda}[\lambda x^{t_a}. []], m_a[t_a \mapsto v], \text{tick } t_a)} \\
 \text{[LINKL]} \frac{}{(e_1!e_2, C, m, t) \rightsquigarrow (e_1, C, m, t)} \quad \text{[LINKR]} \frac{(e_1, C, m, t) \Downarrow (C', m', t')}{(e_1!e_2, C, m, t) \rightsquigarrow (e_2, C', m', t')} \\
 \text{[LETEL]} \frac{}{(\text{let } x \ e_1 \ e_2, C, m, t) \rightsquigarrow (e_1, C, m, t)} \\
 \text{[LETERR]} \frac{(e_1, C, m, t) \Downarrow (v, m', t')}{(\text{let } x \ e_1 \ e_2, C, m, t) \rightsquigarrow (e_2, C[\text{let } x^{t'} []], m'[t' \mapsto v], \text{tick } t')} \\
 \text{[LETML]} \frac{}{(\text{let } M \ e_1 \ e_2, C, m, t) \rightsquigarrow (e_1, C, m, t)} \quad \text{[LETMR]} \frac{(e_1, C, m, t) \Downarrow (C', m', t')}{(\text{let } M \ e_1 \ e_2, C, m, t) \rightsquigarrow (e_2, C[\text{let } M \ C' []], m', t')}
 \end{array}$$

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 3.1** (Collecting semantics using  $\rightsquigarrow^*$ ).

$$\llbracket e \rrbracket(s) = \bigcup_{(e,s) \rightsquigarrow^* (e',s')} [(e',s') \mapsto \{r \mid (e',s') \Downarrow r\}]$$

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 3.1** (Transfer function). Given an element  $a$  of  $(\text{Expr} \times \text{Config}) \rightarrow \wp(\text{Result})_{\perp}$ ,

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

$$\text{step}(a)(e,s) \triangleq [(e,s) \mapsto \{r \mid (e,s) \Downarrow_a r\}] \cup \bigcup_{(e,s) \rightsquigarrow_a (e',s')} [(e',s') \mapsto \emptyset]$$

We define the transfer function  $F$  by:

$$F(a) \triangleq \bigcup_{(e,s) \in \text{dom}(a)} \text{step}(a)(e,s)$$

We can finally formulate the collecting semantics in fixpoint form.

**Lemma 3.2** (Concrete semantics as a fixpoint).

$$\llbracket e \rrbracket(s) = \text{lfp}(\lambda a. F(a) \cup [(e,s) \mapsto \emptyset])$$

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 3.2** (Auxiliary operators for concrete linking).

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

**Definition 3.3** (Concrete linking operator).

$$\text{Link } e_1 e_2 s \triangleq \llbracket e_1 \rrbracket(s) \cup \text{L } (\text{Exp } e_1 s) e_2 \cup [(e_1!e_2, s) \mapsto \text{F } (\text{Exp } e_1 s) e_2]$$

**Theorem 3.1** (Concrete linking).

$$\llbracket e_1!e_2 \rrbracket(s) = \text{Link } e_1 e_2 s$$

That is, the semantics of the linked expression is the union of the semantics of the exporting expression and the semantics of the consuming expression under the exported configuration.

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

Fig. 6. Definition of the semantic domains.

$$\begin{array}{c}
\boxed{(e, C^\#, m^\#, t^\#) \Downarrow^\# (V^\#, m'^\#, t'^\#)} \\
\text{[EXPRVAR]} \frac{t_x^\# = \text{addr}(C^\#, x) \quad v^\# \in m^\#(t_x^\#)}{(x, C^\#, m^\#, t^\#) \Downarrow^\# (v^\#, m^\#, t^\#)} \quad \text{[FN]} \frac{}{(\lambda x.e, C^\#, m^\#, t^\#) \Downarrow^\# ((\lambda x.e, C^\#), m^\#, t^\#)} \\
\text{[APP]} \frac{\begin{array}{c} (e_1, C^\#, m^\#, t^\#) \Downarrow^\# ((\lambda x.e_\lambda, C_\lambda^\#), m_\lambda^\#, t_\lambda^\#) \\ (e_2, C^\#, m_\lambda^\#, t_\lambda^\#) \Downarrow^\# (v^\#, m_a^\#, t_a^\#) \\ (e_\lambda, C_\lambda^\# [\lambda x.t_a^\# . []], m_a^\# [t_a^\# \mapsto^\# v^\#], \text{tick}^\# C^\# m_a^\# t_a^\# x v^\#) \Downarrow^\# (v^\#, m'^\#, t'^\#) \end{array}}{(e_1 e_2, C^\#, m^\#, t^\#) \Downarrow^\# (v'^\#, m'^\#, t'^\#)} \\
\text{[LINKING]} \frac{\begin{array}{c} (e_1, C^\#, m^\#, t^\#) \Downarrow^\# (C'^\#, m'^\#, t'^\#) \\ (e_2, C'^\#, m'^\#, t'^\#) \Downarrow^\# (V^\#, m''^\#, t''^\#) \end{array}}{(e_1 ! e_2, C^\#, m^\#, t^\#) \Downarrow^\# (V^\#, m''^\#, t''^\#)} \\
\text{[EMPTY]} \frac{}{(\epsilon, C^\#, m^\#, t^\#) \Downarrow^\# (C^\#, m^\#, t^\#)} \quad \text{[MODVAR]} \frac{C'^\# = \text{ctx}(C^\#, M)}{(M, C^\#, m^\#, t^\#) \Downarrow^\# (C'^\#, m^\#, t^\#)} \\
\text{[LETE]} \frac{\begin{array}{c} (e_1, C^\#, m^\#, t^\#) \Downarrow^\# (v^\#, m'^\#, t'^\#) \\ (e_2, C^\# [\text{let } x^{t'^\#} []], m'^\# [t'^\# \mapsto^\# v^\#], \text{tick}^\# C^\# m'^\# t'^\# x v^\#) \Downarrow^\# (C'^\#, m''^\#, t''^\#) \end{array}}{(\text{let } x e_1 e_2, C^\#, m^\#, t^\#) \Downarrow^\# (C'^\#, m''^\#, t''^\#)} \\
\text{[LETM]} \frac{\begin{array}{c} (e_1, C^\#, m^\#, t^\#) \Downarrow^\# (C'^\#, m'^\#, t'^\#) \\ (e_2, C^\# [\text{let } M C'^\# []], m'^\#, t'^\#) \Downarrow^\# (C''^\#, m''^\#, t''^\#) \end{array}}{(\text{let } M e_1 e_2, C^\#, m^\#, t^\#) \Downarrow^\# (C''^\#, m''^\#, t''^\#)}
\end{array}$$

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  $m^\# \in \text{Mem}^\#$ ,  $t^\# \in \mathbb{T}^\#$ ,  $v^\# \in \text{Val}$ , 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  $\text{tick}^\#$  function takes in more information than simply the previous abstract time.

$$\begin{array}{c}
\boxed{(e, C^\#, m^\#, t^\#) \rightsquigarrow^\# (e', C'^\#, m'^\#, t'^\#)} \\
\text{[APPL]} \frac{}{(e_1 \ e_2, C^\#, m^\#, t^\#) \rightsquigarrow^\# (e_1, C^\#, m^\#, t^\#)} \quad \text{[APPR]} \frac{(e_1, C^\#, m^\#, t^\#) \Downarrow^\# (\langle \lambda x. e_\lambda, C_\lambda^\# \rangle, m_\lambda^\#, t_\lambda^\#)}{(e_1 \ e_2, C^\#, m^\#, t^\#) \rightsquigarrow^\# (e_2, C^\#, m_\lambda^\#, t_\lambda^\#)} \\
\text{[APPBODY]} \frac{(e_1, C^\#, m^\#, t^\#) \Downarrow^\# (\langle \lambda x. e_\lambda, C_\lambda^\# \rangle, m_\lambda^\#, t_\lambda^\#) \quad (e_2, C^\#, m_\lambda^\#, t_\lambda^\#) \Downarrow^\# (v^\#, m_a^\#, t_a^\#)}{(e_1 \ e_2, C^\#, m^\#, t^\#) \rightsquigarrow^\# (e_\lambda, C_\lambda^\# [\lambda x^{t_a^\#}. []], m_a^\# [t_a^\# \mapsto^\# v^\#], \text{tick}^\# C^\# m_a^\# t_a^\# x v^\#)} \\
\text{[LINKL]} \frac{}{(e_1 ! e_2, C^\#, m^\#, t^\#) \rightsquigarrow^\# (e_1, C^\#, m^\#, t^\#)} \quad \text{[LINKR]} \frac{(e_1, C^\#, m^\#, t^\#) \Downarrow^\# (C'^\#, m'^\#, t'^\#)}{(e_1 ! e_2, C^\#, m^\#, t^\#) \rightsquigarrow^\# (e_2, C'^\#, m'^\#, t'^\#)} \\
\text{[LETEL]} \frac{}{(\text{let } x \ e_1 \ e_2, C^\#, m^\#, t^\#) \rightsquigarrow^\# (e_1, C^\#, m^\#, t^\#)} \\
\text{[LETMR]} \frac{(e_1, C^\#, m^\#, t^\#) \Downarrow^\# (v^\#, m'^\#, t'^\#)}{(\text{let } x \ e_1 \ e_2, C^\#, m^\#, t^\#) \rightsquigarrow^\# (e_2, C^\# [\text{let } x^{t'^\#} []], m'^\# [t'^\# \mapsto^\# v^\#], \text{tick}^\# C^\# m'^\# t'^\# x v^\#)} \\
\text{[LETML]} \frac{}{(\text{let } M \ e_1 \ e_2, C^\#, m^\#, t^\#) \rightsquigarrow^\# (e_1, C^\#, m^\#, t^\#)} \\
\text{[LETMR]} \frac{(e_1, C^\#, m^\#, t^\#) \Downarrow^\# (C'^\#, m'^\#, t'^\#)}{(\text{let } M \ e_1 \ e_2, C^\#, m^\#, t^\#) \rightsquigarrow^\# (e_2, C^\# [\text{let } M \ C'^\# []], m'^\#, t'^\#)}
\end{array}$$

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

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

In our semantics,  $\text{tick}^\# C^\# m^\# t^\# x v^\#$  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.

## 4.2 Soundness

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

**Definition 4.3** (Abstract semantics). The semantics for an expression  $e$  under configuration  $s^\# \in \text{Config}^\#$  is an element in  $(\text{Expr} \times \text{Config}^\#) \rightarrow \wp(\text{Result}^\#)_\perp$  defined as:

$$\llbracket e \rrbracket^\#(s^\#) \triangleq \bigsqcup_{(e, s^\#) \rightsquigarrow^{a^\#} (e', s'^\#)} [(e', s'^\#) \mapsto \{r^\# \mid (e', s'^\#) \Downarrow^\# r^\#\}]$$

We need to present how to compute the abstract semantics by equating the semantics with the least fixed point of some transfer function.

**Definition 4.4** (Transfer function). Given an element  $a^\#$  of  $(\text{Expr} \times \text{Config}^\#) \rightarrow \wp(\text{Result}^\#)_\perp$ ,

- Define  $\Downarrow_{a^\#}^\#$  and  $\rightsquigarrow_{a^\#}^\#$  by replacing all assumptions  $(e, s^\#) \Downarrow^\# r^\#$  to  $r^\# \in a^\#(e, s^\#)$  in  $\Downarrow^\#$  and  $\rightsquigarrow^\#$ .
- Define the  $\text{step}^\#$  function that collects all results derivable in one step from  $(e, s^\#)$  using  $a^\#$ .

$$\text{step}^\#(a^\#)(e, s^\#) \triangleq [(e, s^\#) \mapsto \{r^\# \mid (e, s^\#) \Downarrow_{a^\#}^\# r^\#\}] \sqcup \bigsqcup_{(e, s^\#) \rightsquigarrow_{a^\#}^\# (e', s'^\#)} [(e', s'^\#) \mapsto \emptyset]$$

We define the transfer function  $F^\#$  by:

$$F^\#(a^\#) \triangleq \bigsqcup_{(e, s^\#) \in \text{dom}(a^\#)} \text{step}^\#(a^\#)(e, s^\#)$$

**Lemma 4.1** (Abstract semantics as a fixpoint).

$$\llbracket e \rrbracket^\#(s^\#) = \text{lfp}(\lambda a^\#. F^\#(a^\#) \sqcup [(e, s^\#) \mapsto \emptyset])$$

The usual thing to do now is to connect the collecting semantics  $\llbracket e \rrbracket$  with the abstract semantics  $\llbracket e \rrbracket^\#$  via a Galois connection. However, we do not enforce the existence of an explicit abstraction and concretization function. Instead, we define a notion of soundness between abstract and concrete results and prove that for any tick<sup>#</sup>, the abstract semantics overapproximate the collecting semantics if it starts from a sound configuration.

**Definition 4.5** ( $\alpha$ -soundness between results).

- Let  $(V, m, t) \in \text{Result}$  and  $(V^\#, m^\#, t^\#) \in \text{Result}^\#$ . We do not assume that  $\mathbb{T}$  and  $\mathbb{T}^\#$  are concrete/abstract times.
- Let  $\alpha : \mathbb{T} \rightarrow \mathbb{T}^\#$ , and extend  $\alpha$  to a function in  $\text{Ctx} \rightarrow \text{Ctx}$  by mapping  $\alpha$  over all timestamps.
- Extend  $\alpha$  to a function in  $(\text{Ctx} + \text{Val}) \rightarrow (\text{Ctx} + \text{Val})$  accordingly.
- Extend  $\alpha$  to a function in  $\text{Mem} \rightarrow \text{Mem}^\#$  by defining

$$\alpha(m) \triangleq \bigsqcup_{t \in \text{dom}(m)} [\alpha(t) \mapsto \{\alpha(m(t))\}]$$

We say that  $(V^\#, m^\#, t^\#)$  is an  $\alpha$ -sound approximation of  $(V, m, t)$  when  $\alpha(V) = V^\#$ ,  $\alpha(m) \sqsubseteq m^\#$ , and  $\alpha(t) = t^\#$ .

**Definition 4.6** (Soundness between semantics :  $\lesssim$ ).

- Let  $a \in (\text{Expr} \times \text{Config}) \rightarrow \wp(\text{Result})_\perp$  and  $a^\# \in (\text{Expr} \times \text{Config}^\#) \rightarrow \wp(\text{Result}^\#)_\perp$ .

We say that  $a^\#$  is a sound approximation of  $a$  and write  $a \lesssim a^\#$  if:

$\forall e \in \text{Expr}, s \in \text{Config}, r \in \text{Result} : r \in a(e, s) \Rightarrow \exists \alpha, \alpha', s^\#, r^\# : \alpha(s) \sqsubseteq s^\# \wedge \alpha'(r) \sqsubseteq r^\# \in a^\#(e, s^\#)$   
 $a^\#$  is sound with respect to  $a$  iff for every  $(e, s), r$  pair in  $a$ , there exists an  $\alpha$ -sound pair in  $a^\#$ .

**Lemma 4.2** (Preservation of soundness, relation version).

- Let  $s \in \text{Config}$  and  $s^\# \in \text{Config}^\#$ .
- Let all timestamps in the  $C$  and  $m$  component of  $s$  be strictly less than the  $t$  component.
- Let  $s^\#$  be an  $\alpha$ -sound approximation of  $s$  for some  $\alpha$ .

Then for all  $e$ ,

- (1) If  $(e, s) \Downarrow r$ , then  $\exists \alpha', r^\#$  such that  $(e, s^\#) \Downarrow^\# r^\#$ ,  $\alpha'(s) \sqsubseteq s^\#$ , and  $\alpha'(r) \sqsubseteq r^\#$ .
- (2) If  $(e, s) \rightsquigarrow (e', s')$ , then  $\exists \alpha', s'^\#$  such that  $(e, s^\#) \rightsquigarrow^\# (e', s'^\#)$ ,  $\alpha'(s) \sqsubseteq s^\#$ , and  $\alpha'(s') \sqsubseteq s'^\#$ .

**Lemma 4.3** (Preservation of soundness).

- Let  $s \in \text{Config}$  and  $s^\# \in \text{Config}^\#$ .
- Let all timestamps in the  $C$  and  $m$  component of  $s$  be strictly less than the  $t$  component.
- Let  $s^\#$  be an  $\alpha$ -sound approximation of  $s$  for some  $\alpha$ .

Then for all  $e$ ,  $\llbracket e \rrbracket(s) \lesssim \llbracket e \rrbracket^\#(s^\#)$ .

What's remarkable is that we did not put any constraint on the tick and tick<sup>#</sup> functions. Moreover, we can guarantee that  $\llbracket e \rrbracket^\#(s^\#)$  can be computed.

**Theorem 4.1** (Finiteness of time implies finiteness of abstraction). If  $\mathbb{T}^\#$  is finite,

$$\forall e, s^\# : |\llbracket e \rrbracket^\#(s^\#)| < \infty$$



## 5 MODULAR ANALYSIS

Now for separate analysis, we need 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 and deletion operators 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.

$$\begin{aligned}
 \text{map\_inject}(C_1, C_2) &\triangleq \begin{cases} [] & C_2 = [] \\ \lambda x^t. \text{map\_inject}(C_1, C') & C_2 = \lambda x^t. C' \\ \text{let } x^t \text{ map\_inject}(C_1, C') & C_2 = \text{let } x^t C' \\ \text{let } M C_1 [\text{map\_inject}(C_1, C')] \text{ map\_inject}(C_1, C'') & C_2 = \text{let } M C' C'' \end{cases} \\
 C_1 \langle C_2 \rangle &\triangleq C_1 [\text{map\_inject}(C_1, C_2)] \\
 \text{delete\_prefix}(C_1, C_2) &\triangleq \begin{cases} \text{delete\_prefix}(C'_1, C'_2) & (C_1, C_2) = (\lambda x^t. C'_1, \lambda x^t. C'_2) \\ \text{delete\_prefix}(C'_1, C'_2) & (C_1, C_2) = (\text{let } x^t C'_1, \text{let } x^t C'_2) \\ \text{delete\_prefix}(C'_1, C'_2) & (C_1, C_2) = (\text{let } M C' C'_1, \text{let } M C' C'_2) \\ C_2 & \text{otherwise} \end{cases} \\
 \text{delete\_map}(C_1, C_2) &\triangleq \begin{cases} [] & C_2 = [] \\ \lambda x^t. \text{delete\_map}(C_1, C') & C_2 = \lambda x^t. C' \\ \text{let } x^t \text{ delete\_map}(C_1, C') & C_2 = \text{let } x^t C' \\ \text{let } M \text{ delete\_map}(C_1, \text{delete\_prefix}(C_1, C')) \text{ delete\_map}(C_1, C'') & C_2 = \text{let } M C' C'' \end{cases} \\
 \text{delete}(C_1, C_2) &\triangleq \text{delete\_map}(C_1, \text{delete\_prefix}(C_1, C_2))
 \end{aligned}$$

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

We extend the injection operator to map over all closures in a memory so that  $C \langle m^\# \rangle$  can be defined naturally.

The deletion operation has the expected property that  $\text{delete}(C, C \langle C' \rangle) = C'$ .

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.

$$\begin{aligned}
 \text{filter}(C, \mathbb{T}) &\triangleq \begin{cases} [] & (C = []) \\ \lambda x^t. \text{filter}(C', \mathbb{T}) & (C = \lambda x^t. C' \wedge t \in \mathbb{T}) \\ \text{let } x^t \text{ filter}(C', \mathbb{T}) & (C = \text{let } x^t C' \wedge t \in \mathbb{T}) \\ \text{let } M \text{ filter}(C', \mathbb{T}) \text{ filter}(C'', \mathbb{T}) & (C = \text{let } M C' C'') \\ \text{filter}(C', \mathbb{T}) & (C = \lambda x^t. C' \wedge t \notin \mathbb{T}) \\ \text{filter}(C', \mathbb{T}) & (C = \text{let } x^t C' \wedge t \notin \mathbb{T}) \end{cases} \\
 \text{filter}(v, \mathbb{T}) &\triangleq \langle \lambda x. e, \text{filter}(C, \mathbb{T}) \rangle \quad (v = \langle \lambda x. e, C \rangle) \\
 \text{filter}(m^\#, \mathbb{T}) &\triangleq \lambda t \in \mathbb{T}. \{ \text{filter}(v, \mathbb{T}) \mid v \in m^\#(t) \}
 \end{aligned}$$

Fig. 10. Definitions for the filter operation.

**Definition 5.1** (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  $\mathbb{T}_2^\#$ .

Define  $s^\# \triangleright r^\# \triangleq (C_1^\# \langle V_2^\# \rangle, C_1^\# \langle m_2^\# \rangle \sqcup m_1^\#, t_2^\#)$  to be a result in  $\mathbb{T}_1^\# + \mathbb{T}_2^\#$ .

We extend the  $\triangleright$  operator to inject  $s^\#$  in an element of  $(\text{Expr} \times \text{Config}^\#) \rightarrow \wp(\text{Result}^\#)_\perp$ :

$$s^\# \triangleright a^\# \triangleq \bigsqcup_{(e, s'^\#) \in \text{dom}(a^\#)} [(e, s^\# \triangleright s'^\#) \mapsto \{s^\# \triangleright r^\# \mid r^\# \in a^\#(e, s'^\#)\}]$$

**Definition 5.2** (Addition of time domains).

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

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

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

**Lemma 5.1** (Injection preserves soundness).

Let  $s = (C, m, t)$  be a configuration in a concrete time  $(\mathbb{T}, \leq, \text{tick})$  with all timestamps in  $C, m$  strictly less than  $t$ , and let  $s^\#$  be an  $\alpha$ -sound approximation of  $s$  with timestamps in  $\mathbb{T}_1^\#$ . Then there exists an  $\alpha'$  such that  $s^\# \triangleright ([], \emptyset, 0^\#)$  is an  $\alpha'$ -sound approximation of  $s$ , when  $0^\# \in \mathbb{T}_2^\#$ .

**Lemma 5.2** (Injection preserves timestamps under added time).

Let  $s^\#$  be a configuration in  $\mathbb{T}_1^\#$ ,  $\llbracket e \rrbracket^\#(s'^\#)$  be the semantics of  $e$  under  $(\mathbb{T}_2^\#, \text{tick}^\#)$ , and  $\llbracket e \rrbracket^\#(s^\# \triangleright s'^\#)$  be the semantics of  $e$  under  $(\mathbb{T}_1^\# + \mathbb{T}_2^\#, \text{tick}_+^\#(s^\#))$ . Then:

$$s^\# \triangleright \llbracket e \rrbracket^\#(s'^\#) \sqsubseteq \llbracket e \rrbracket^\#(s^\# \triangleright s'^\#)$$

**Definition 5.3** (Addition between exported configurations and separately analyzed results).

Let  $s_1^\#$  be a configuration in  $\mathbb{T}_1^\#$ , and let  $a_2^\# = \llbracket e \rrbracket^\#(s'^\#)$  be the semantics of  $e$  under  $(\mathbb{T}_2^\#, \text{tick}^\#)$ . Define the “addition” between  $s_1^\#$  and  $a_2^\#$  as:

$$s_1^\# \oplus a_2^\# \triangleq \text{lfp}(\lambda a^\#. F^\#(a^\#) \sqcup (s_1^\# \triangleright a_2^\#))$$

**Lemma 5.3** (Addition of semantics equals semantics under added time).

$$s_1^\# \oplus \llbracket e_2 \rrbracket^\#(s'^\#) = \llbracket e \rrbracket^\#(s^\# \triangleright s'^\#)$$

**Definition 5.4** (Auxiliary operators for abstract linking).

$$\text{Exp}^\# a^\# e_1 s^\# \triangleq a^\#(e_1, s^\#) \quad (\text{Exported under } s^\# \text{ using } a^\#)$$

$$\text{L}^\# E^\# e_2 \triangleq \bigsqcup_{s'^\# \in E^\#} (s'^\# \oplus \llbracket e_2 \rrbracket^\#([], \emptyset, 0^\#)) \quad (\text{Reached under } E^\#)$$

$$\text{F}^\# E^\# e_2 \triangleq \bigsqcup_{s'^\# \in E^\#} (s'^\# \oplus \llbracket e_2 \rrbracket^\#([], \emptyset, 0^\#))(e_2, s'^\# \triangleright ([], \emptyset, 0^\#)) \quad (\text{Final results under } E^\#)$$

**Definition 5.5** (Abstract linking operator).

$$\text{Link}^\# a^\# e_1 e_2 s^\# \triangleq a^\# \sqcup \text{L}^\# (\text{Exp}^\# a^\# e_1 s^\#) e_2 \sqcup [(e_1!e_2, s^\#) \mapsto \text{F}^\# (\text{Exp}^\# a^\# e_1 s^\#) e_2]$$

**Theorem 5.1** (Abstract linking). Let  $s$  be a concrete configuration.

Let  $s^\#$  be an  $\alpha$ -sound approximation of  $s$  for some  $\alpha$ , let  $\llbracket e_1 \rrbracket(s) \lesssim a^\#$ , and let  $\llbracket e_1 \rrbracket(s)(e_1, s) \lesssim a^\#(e_1, s^\#)$ . Then:

$$\llbracket e_1!e_2 \rrbracket(s) \lesssim \text{Link}^\# a^\# e_1 e_2 s^\#$$

and

$$\llbracket e_1!e_2 \rrbracket(s)(e_1!e_2, s) \lesssim (\text{Link}^\# a^\# e_1 e_2 s^\#)(e_1!e_2, s^\#)$$

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

**Theorem 5.2** (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:

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

and define  $L_i \triangleq \llbracket l_i \rrbracket(s)$ . Now, given an  $\alpha$ -sound approximation  $s^\#$  of  $s$ , define

$$L_0^\# \triangleq \llbracket e_0 \rrbracket^\#(s^\#) \quad L_{i+1}^\# \triangleq \text{Link}^\# L_i^\# l_i e_{i+1} s^\#$$

Then we have:

$$\forall n : L_n \lesssim L_n^\#$$