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 big-step operational semantics for the dynamic execution of the module language. The big-step evaluation relation relates the initial *configuration*(expression and state) with the *result*(value and state) it returns.

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, namely the *value*, *memory*, and *context* domains. Also, we present the notational conventions used in this paper to represent members of each domain.

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

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) $\mathsf{tick} : \mathrm{Ctx}(\mathbb{T}) \to \mathrm{Mem}(\mathbb{T}) \to \mathbb{T} \to \mathrm{ExprVar} \to \mathrm{Val}(\mathbb{T}) \to \mathbb{T}$ 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 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.

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

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

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

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

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

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.

2.2 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* 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 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 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. This is actually a function from the *reached configurations* during the execution of the program to the set of results the configuration returns.

This "set of reached configurations" is intuitively clear. Given a big-step interpreter that takes in (e, s) and returns r, the set of reached configurations is the set that the interpreter looks at during the computation. To formulate this intuition into a formally correct definition, we need to define a relation defining what configurations are looked at when evaluating the initial configuration.

2.2.1 Formalizing reachability. The reachability relation that formalizes the concept of what (e', s') the interpreter "sees" when evaluating (e, s) directly follows the definition of the big-step interpreter. For example, in the case for application, first the function part must be evaluated(AppL), and if the function part is returned, the argument part must be evaluated(AppR), and finally the function body must be evaluated (AppBody). The complete definition for the reachability relation is given in 5.

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

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

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

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

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

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

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

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.

2.2.2 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, elements of the domain D is a cache that remembers what configurations (e, s) the interpreter saw and what results r the interpreter computed, which is common in defining collecting semantics for higher-order languages[Darais et al. 2017; Hudak and Young 1991]. Mathematically, the cache a is an element of $(\text{Expr} \times \text{State}(\mathbb{T})) \to \wp(\text{Result}(\mathbb{T}))$, when the ordering is defined pointwise, with undefined outputs as the bottom and the rest ordered by the set inclusion order.

Intuitively, such a cache can be interpreted as holding the *intermediate* state of the interpreter. That is, the cache records *all* intermediate inputs to the interpreter and *all* results that the interpreter computed. Then the transfer function that takes in a cache and outputs a cache that corresponds to the cache after *one step* of the interpreter becomes clear. For each reached configurations in the domain of the cache, the transfer function simulates what the actual interpreter will do, assuming that what it knows is constrained by the given cache. For example, if the function and argument part in an application is already known but the body is yet not computed, the transfer function adds the body of the application to the next cache.

Definition 2.2 (Transfer function)**.** Given a partial function $a : (\text{Expr} \times \text{State}(\mathbb{T})) \to \wp(\text{Result}(\mathbb{T}))$,

- Define $\Downarrow_a$ and $\rightsquigarrow_a$ by replacing all premises $(e, s) \Downarrow r$ by $r \in a(e, s)$ in $\Downarrow$ and $\rightsquigarrow$.
- Define the step function that does what the interpreter will do with input (e, s) under a.

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

We define the transfer function Step by:

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

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

Definition 2.3 (Concrete semantics)**.**

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

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 $[\![e_1]\!](C_1, m_1, 0_1)$ and $[\![e_2]\!]([\,], \varnothing, 0_2)$, when the context and memory part of 0_2 are empty and therefore is ready for injection from the final memory and context of $[\![e_1]\!](\_, \_, 0_1)$. 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 $(f_1, 0_2)$, when $f_1 \in \mathbb{T}_1$ is the final timestamp of $[\![e_1]\!](\_, \_, 0_1)$.

(2) Increments $(f_1, 0_2)$ up to (f_1, f_2), when $f_2 \in \mathbb{T}_2$ 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 f_1. Also, in the case that the first timestamp is greater or equal to f_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 $(e_1, C_1, m_1, (0_1, 0_2)) \Downarrow (C_2, m_2, (f_1, 0_2))$ is computed, and only then is 0_2 incremented under C_2, m_2. For tick_+ to produce the same timestamps produced by tick_2 under the empty initial conditions, C_2 has to be dug out from the bottom of the stack and m_2 must be filtered out. Filtering out m_2 is easy to do; just ignore addresses with a timestamp less than f_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 $C_2\langle C\rangle$ that satisfies: if $(e_2, [\,], \varnothing, 0_2) \rightsquigarrow^* (e, C, m, t)$, then $(e_2, C_2, \varnothing, 0_2) \rightsquigarrow^* (e, C_2\langle C\rangle, C_2\langle m\rangle, t)$ with tick_+. The notation $C_2\langle m\rangle$ means that the context C_2 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. 6. 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, the context that is bound *automatically* includes the exported context. This is a consequence of the design choice to exclude signatures. In a more sensible 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. 7. 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. The filter operation is defined in 8. Note that the function $\pi_1 : A \times B \longrightarrow A$ projects a tuple to its first component, and $\pi_2 : A \times B \longrightarrow B$ projects to the second component.

$$\mathsf{filter}_1(C) \triangleq \begin{cases} [\,] & C = [\,] \\ (x, \pi_1(t)) :: \mathsf{filter}_1(C') & C = (x,t) :: C' \wedge \pi_1(t) < f_1 \\ \mathsf{filter}_1(C') & C = (x,t) :: C' \wedge \pi_1(t) \geq f_1 \\ (M, \mathsf{filter}_1(C')) :: \mathsf{filter}_1(C'') & C = (M, C') :: C'' \end{cases}$$

$$\mathsf{filter}_2(C) \triangleq \begin{cases} [\,] & C = [\,] \\ (x, \pi_2(t)) :: \mathsf{filter}_2(C') & C = (x,t) :: C' \wedge \pi_1(t) \geq f_1 \\ \mathsf{filter}_2(C') & C = (x,t) :: C' \wedge \pi_1(t) < f_1 \\ (M, \mathsf{filter}_2(C')) :: \mathsf{filter}_2(C'') & C = (M, C') :: C'' \end{cases}$$

Fig. 8. Definitions for the filter operation.

The definition of the tick_+ can finally be given:

Definition 3.1 (Concrete linking of time domains)**.**

- Let $(\mathbb{T}_1, \leq_1, \mathsf{tick}_1)$ and $(\mathbb{T}_2, \leq_2, \mathsf{tick}_2)$ be two concrete times.
- Let $s_1 = (C_1, m_1, f_1)$ be a state in $\mathbb{T}_1$.
- Define $\leq_+$ as the lexicographic order on $\mathbb{T}_1 \times \mathbb{T}_2$.
- Define the $\mathsf{tick}_+(s_1)$ function as:

$$\mathsf{tick}_+(s_1)(C, m, t, x, v) \triangleq \begin{cases} (\mathsf{tick}_1\ \mathsf{filter}_1(C, m, t, x, v), \pi_2(t)) & (\pi_1(t) < f_1) \\ (\pi_1(t), \mathsf{tick}_2\ \mathsf{filter}_2((C_1, 0_2)\overline{\langle C, m, t, x, v\rangle})) & (\pi_1(t) \geq f_1) \end{cases}$$

when $(C_1, 0_2)$ is C_1 with all timestamps lifted to $\mathbb{T}_1 \times \mathbb{T}_2$ by fixing the second time to 0_2.

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

We can confirm that the linked time domain indeed preserves the timestamps produced by tick_2 without knowledge about the exported state s_1. 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)**.**

Let $s = (C_1, m_1, f_1) \in \mathrm{State}(\mathbb{T}_1)$ be an exported state, and let $r = (V_2, m_2, t_2) \in \mathrm{Result}(\mathbb{T}_2)$.

Define $s \triangleright m_2$ and $s \triangleright r$ as:

$$s \triangleright m_2 \triangleq \lambda t. \begin{cases} m_1(\pi_1(t)) & \pi_1(t) < f_1 \\ C_1\langle m_2\rangle(\pi_2(t)) & \pi_1(t) \geq f_1 \end{cases} \qquad s \triangleright r \triangleq (C_1\langle V_2\rangle, s \triangleright m_2, t_2)$$

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 (f_1, t).

We extend the $\triangleright$ operator to inject s in a cache $a \in (\mathrm{Expr} \times \mathrm{State}(\mathbb{T}_2)) \longrightarrow \wp(\mathrm{Result}(\mathbb{T}_2))$:

$$s \triangleright a \triangleq \bigcup_{(e,s') \in \mathsf{dom}(a)} [(e, s \triangleright s') \mapsto \{s \triangleright r | r \in a(e, s')\}]$$

Then we can prove that the tick_+ function is indeed *well-defined*.

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

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

Now, as promised in the start of this section, we present how to link the semantics to obtain the semantics under linked time.

Definition 3.3 (Auxiliary operators for concrete linking)**.**

$$\mathsf{Exp}\ e_1\ s \triangleq [\![e_1]\!](s)(e_1, s) \qquad \text{(Exported under } s\text{)}$$
$$\mathsf{L}\ E\ e_2 \triangleq \bigcup_{s' \in E} [\![e_2]\!](s' \triangleright 0_2) \qquad \text{(Reached under } E\text{)}$$
$$\mathsf{F}\ E\ e_2 \triangleq \bigcup_{s' \in E} [\![e_2]\!](s' \triangleright 0_2)(e_2, s' \triangleright 0_2) \qquad \text{(Final results under } E\text{)}$$

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 3.4 (Concrete linking operator)**.**

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

when the timestamps t in $[\![e_1]\!](s)$ are lifted to $(t, 0_2)$.

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 $\mathbb{T}$.

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

Fig. 9. Definition of the semantic domains.

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. 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 $\mathsf{tick}^\# : \mathsf{Ctx}(\mathbb{T}^\#) \to \mathsf{Mem}^\#(\mathbb{T}^\#) \to \mathbb{T}^\# \to \mathsf{ExprVar} \to \mathsf{Val}(\mathbb{T}^\#) \to \mathbb{T}^\#$ is the policy for advancing the timestamp.

The abstract big-step evaluation relation is defined in 10, and the single-step reachability relation is defined in 11.

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

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

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

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

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

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

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

Fig. 10. The abstract big-step evaluation relation.

From the relations, we can define the abstract semantics in the same way as the concrete version:

Definition 4.3 (Transfer function). Given a cache $a^\#$ in $(\mathsf{Expr} \times \mathsf{State}^\#(\mathbb{T}^\#)) \to \wp(\mathsf{Result}^\#(\mathbb{T}^\#))$,

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

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

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

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

Definition 4.4 (Abstract semantics).

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

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

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

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

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

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

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

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

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

Fig. 11. The abstract single-step reachability relation.

5 NON-MODULAR 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 $\wp(\text{Result}(\mathbb{T}))$ and $\wp(\text{Result}^\#(\mathbb{T}^\#))$ 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 $\text{Ctx}(\mathbb{T}) \to \text{Ctx}(\mathbb{T}^\#)$ by mapping α over all timestamps.
- Extend α to a function on $\text{Val}(\mathbb{T}) \to \text{Val}(\mathbb{T}^\#)$ by mapping α over all timestamps.
- Extend α to a function on $\text{Mem}(\mathbb{T}) \to \text{Mem}^\#(\mathbb{T}^\#)$ by defining
$$\alpha(m) \triangleq \bigcup_{t \in \mathsf{dom}(m)} [\alpha(t) \mapsto \{\alpha(m(t))\}]$$
- Extend α to a function on $\text{Result}(\mathbb{T}) \to \text{Result}^\#(\mathbb{T}^\#)$ by $\alpha(V, m, t) \triangleq (\alpha(V), \alpha(m), \alpha(t))$

Then it is obvious that:

Lemma 5.1 (Galois connection)**.** Given a function $\alpha : \mathbb{T} \to \mathbb{T}^\#$,

- Extend $\alpha : \wp(\text{Result}(\mathbb{T})) \to \wp(\text{Result}^\#(\mathbb{T}^\#))$ by $\alpha(R) \triangleq \{\alpha(r) | r \in R\}$.
- Extend $\gamma : \wp(\text{Result}^\#(\mathbb{T}^\#)) \to \wp(\text{Result}(\mathbb{T}))$ by $\gamma(R^\#) \triangleq \{r | \alpha(r) \in R^\#\}$.

Then $\forall R \subseteq \text{Result}(\mathbb{T}), R^\# \subseteq \text{Result}^\#(\mathbb{T}^\#) : \alpha(R) \subseteq R^\# \Leftrightarrow R \subseteq \gamma(R^\#)$.

The ordering between $\wp(\text{Result}(\mathbb{T}))$ and $\wp(\text{Result}^\#(\mathbb{T}^\#))$ 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 the pair $(e, s^\#)$.

This Galois connection can naturally be extended to a connection between the abstract and concrete caches, when the abstraction $\alpha(a)$ of a cache a is defined by

$$\alpha(a) \triangleq \bigcup_{(e,s)\in \mathsf{dom}(a)} [(e, \alpha(s)) \mapsto \alpha(a(e,s))]$$

and the concretization γ is defined by

$$\gamma(a^\#) \triangleq \bigcup_{(e,s^\#)\in \mathsf{dom}(a^\#)} \bigcup_{s\in\gamma(\{s^\#\})} [(e,s) \mapsto \gamma(a^\#(e,s^\#))]$$

. Then all we need to show is that the abstract semantics overapproximates the concrete semantics, i.e, that $[\![e]\!](s) \sqsubseteq \gamma([\![e]\!]^\#(\alpha(s)))$. 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:

Lemma 5.2 (Soundness)**.** Given a tick-approximating $\alpha : \mathbb{T} \to \mathbb{T}^\#$,

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

What's truly remarkable is that if the abstract time domain is finite, the analysis is guaranteed to terminate. Given the recursive nature of the stack C, it is not obvious that the state space given a finite $\mathbb{T}^\#$ is finite. However, since the *syntax* of the program constrains how C looks like, we can prove that:

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

$$\forall e, s^\# : [\![e]\!]^\#(s^\#) \text{ can be finitely computed.}$$

6 MODULAR ANALYSIS

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:

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

Fig. 12. Definition of the abstract filter operation.

Then the linked time domain can be defined as:

Definition 6.1 (Addition of time domains)**.**

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

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

Then we call the abstract time $(\mathbb{T}_1^\# + \mathbb{T}_2^\#, \mathsf{tick}_+^\#({s_1}^\#))$ the linked time 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)**.**
Let ${s}^\# = ({C_1}^\#, {m_1}^\#, {f_1}^\#) \in \mathrm{State}^\#(\mathbb{T}_1^\#)$ be an exported state, and let ${r}^\# = ({V_2}^\#, {m_2}^\#, {t_2}^\#) \in \mathrm{Result}^\#(\mathbb{T}_2^\#)$. Define ${s}^\# \triangleright^\# {m_2}^\#$ and ${s}^\# \triangleright^\# {r}^\#$ as:

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

We extend the $\triangleright^\#$ operator to inject ${s}^\#$ in a cache ${a}^\# \in (\mathrm{Expr} \times \mathrm{State}^\#(\mathbb{T}_2^\#)) \to \wp(\mathrm{Result}^\#(\mathbb{T}_2^\#))$:

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

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

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

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

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

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

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

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

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

The auxiliary operators for linking can also be defined as in the concrete case:

Definition 6.4 (Auxiliary operators for abstract linking)**.**

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

$$\mathsf{L}^\#\ {E}^\#\ e_2 \triangleq \bigcup_{{s'}^\# \in {E}^\#} ({s'}^\# \oplus [\![e_2]\!]^\#(0_2^\#)) \qquad \text{(Reached under } {E}^\#)$$

$$\mathsf{F}^\#\ {E}^\#\ e_2 \triangleq \bigcup_{{s'}^\# \in {E}^\#} ({s'}^\# \oplus [\![e_2]\!]^\#(0_2^\#))({s'}^\# \triangleright^\# {0_2}^\#) \qquad \text{(Final results under } {E}^\#)$$

Then the abstract linking operator follows:

Definition 6.5 (Abstract linking operator)**.**

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

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

The tricky part is in the time $(f_1, 0_2)$. It is represented by *both* $\alpha_1(f_1) \in \mathbb{T}_1^\#$ 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 $\mathbb{T}_1 \times \mathbb{T}_2$ and $\mathbb{T}_1^\# + \mathbb{T}_2^\#$. Instead, we define a function α_+ that projects the time $(f_1, 0_2)$ to both ${f_1}^\#$ and ${0_2}^\#$ between $\mathsf{Result}(\mathbb{T}_1 \times \mathbb{T}_2)$ and $\mathsf{Result}^\#(\mathbb{T}_1^\# + \mathbb{T}_2^\#)$.

Definition 6.6 (Linking of abstraction)**.**

- Let $s_1 = (C_1, m_1, f_1)$ be the exported state from $\mathbb{T}_1$ to $\mathbb{T}_2$.
- Define the abstractions $\alpha_+^<, \alpha_+^\leq : \mathbb{T}_1 \times \mathbb{T}_2 \to \mathbb{T}_1^\# + \mathbb{T}_2^\#$ as:

$$\alpha_+^< \triangleq \begin{cases} \alpha_1(\pi_1(t)) & t < (f_1, 0_2) \\ \alpha_2(\pi_2(t)) & \text{otherwise} \end{cases} \qquad \alpha_+^\leq \triangleq \begin{cases} \alpha_1(\pi_1(t)) & t \leq (f_1, 0_2) \\ \alpha_2(\pi_2(t)) & \text{otherwise} \end{cases}$$

We define the linked abstraction function $\alpha_+(R) \triangleq \alpha_+^<(R) \cup \alpha_+^\leq(R)$.

Then the added concretization function γ_+ that maps a subset of $\mathsf{Result}^\#(\mathbb{T}_1^\# + \mathbb{T}_2^\#)$ to a subset of $\mathsf{Result}(\mathbb{T}_1 \times \mathbb{T}_2)$ can be defined, and the function can be extended between caches.

Now we can finally state the theorem for abstract linking:

Theorem 6.1 (Abstract linking)**.**

Let $\{e_n\}_{n \geq 0}$ be a sequence of expressions and let s be a concrete configuration. Define $\{l_n\}_{n \geq 0}$ as:

$$l_0 \triangleq e_0 \qquad l_{n+1} \triangleq l_n!e_{n+1}$$

and define $L_n \triangleq [\![l_n]\!](s)$ under the linked time $\prod_{i=0}^n \mathbb{T}_i$.

Given a sequence of tick-approximating abstractions α_i, define α_+^n that maps a subset of $\mathsf{Result}(\prod_{i=0}^n \mathbb{T}_i)$ to a subset of $\mathsf{Result}^\#(\sum_{i=0}^n \mathbb{T}_i^\#)$ by projecting each exported time into the left or right abstract times.

Now, if we let

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

we have:

$$\forall n : L_n \subseteq \gamma_+^n({L_n}^\#)$$

REFERENCES

David Darais, Nicholas Labich, Phuc C. Nguyen, and David Van Horn. 2017. Abstracting definitional interpreters (functional pearl). *PACMPL* 1, ICFP (2017), 12:1–12:25. https://doi.org/10.1145/3110256

Paul Hudak and Jonathan Young. 1991. Collecting Interpretations of Expressions. *ACM Trans. Program. Lang. Syst.* 13, 2 (Apr. 1991), 269–290. https://doi.org/10.1145/103135.103139