Automatic Space Bound Analysis for Functional Programs with Garbage Collection

Yue Niu and Jan Hoffmann

Carnegie Mellon University, Pittsburgh, PA, United States
{yuen,jhoffmann}@cs.cmu.edu

Abstract

This article introduces a novel system for deriving upper bounds on the heap-space requirements of functional programs with garbage collection. The space cost model is based on a perfect garbage collector that immediately deallocates memory cells when they become unreachable. Heap-space bounds are derived using type-based automatic amortized resource analysis (AARA), a template-based technique that efficiently reduces bound inference to linear programming. The first technical contribution of the work is a new operational cost semantics that models a perfect garbage collector. The second technical contribution is an extension of AARA to take into account automatic deallocation. A key observation is that deallocation of a perfect collector can be modeled with destructive pattern matching if data structures are used in a linear way. However, the analysis uses destructive pattern matching to accurately model deallocation even if data is shared. The soundness of the extended AARA with respect to the new cost semantics is proven in two parts via an intermediate linear cost semantics. The analysis and the cost semantics have been implemented as an extension to Resource Aware ML (RaML). An experimental evaluation shows that the system is able to derive tight symbolic heap-space bounds for common algorithms. Often the bounds are asymptotic improvements over bounds that RaML derives without taking into account garbage collection.

1 Introduction

The memory footprint of a program is an important performance metric that determines if a program can be safely executed on a given system. Ideally, developers should describe or approximate the memory footprint of programs as functions of the inputs. However, such memory bounds are often difficult to derive and to prove sound. To assist programmers with deriving memory bounds, the programming language community has developed automatic and semi-automatic analysis techniques [24, 12, 2]. These systems are often special cases of more general resource bound analyses that are based on abstract interpretation [18, 7, 37], recurrence solving [16, 1, 14, 28], type systems [27, 22, 29, 43, 42, 15], program logics [5, 10, 9, 35], proof assistants [33, 11], and term rewriting [6, 34, 17].

This article introduces a novel type system for automatically deriving upper bounds on the heap-space requirements of functional programs with garbage collection (GC). Due to the challenges of modeling and predicting garbage collection, most existing techniques for automating and guiding the derivation of bounds on the heap memory requirements assume manual memory management or simply ignore deallocation in the analysis [24, 26, 36, 13, 12, 2]. As a result, the derived bounds are not accurate when the underlying system employs garbage collection. The only exceptions we are aware of are the works by Albert et al. [3, 4], Braberman et al. [8], and Unnikrishnan et al. [40, 39]. They analyze the heap-space usage of programs with GC in two steps. First, they make the deallocation of GC explicit; for example with a static analysis for estimating object lifetimes [4] or with a program translation [39]. Second, they extract and solve recurrence relations to derive a bound. The difference of our work is that our technique is based on a type system, which is proved sound with respect to a formal cost semantics. Advantages of a type-based approach include natural compositionality and the use of type derivations as certificates for resource bounds.

We model the (highwater mark) memory usage based on a perfect garbage collector that immediately deallocates memory cells when they become unreachable. The bounds that are derived with respect to this cost model are not only a good theoretical measure of the heap-space consumption of the program but also have practical relevance. Consider a function $f : A \to B$ and assume we derived a bound $b_f : [\![A]\!] \to \mathbb{N}$. In an execution of $f(a)$, we can then keep track of the memory usage and start the garbage collector whenever the bound $b_f(a)$ is reached. It is then guaranteed that the evaluation will succeed using $b_f(a)$ heap-memory cells.[1] To improve performance, we could trigger GC more often (to compactify the heap) or allow memory use of more than $b_f(a)$ cells (to amortize the cost of garbage collection).

The first technical contribution of the work is a new operational cost semantics that models a perfect garbage collector. The cost semantics is a big-step (or natural) semantics that keeps track of the reachable memory cells in the style of Spoonhower et al. [38] and Minamide [30]. Operationally, this cost is the high-water mark on the heap usage, or the maximum number of cells used in the mutable store during evaluation. If we traverse the evaluation tree in preorder and view each node as a "step" of the computation, then a cell is used in the current node if it is reachable from the reminder of the computation. Our formalization of reachability is identical to the concept that garbage collectors implement to decide if a cell can be freed during evaluation. For simplicity, we assume that evaluation of the cons node allocates one fresh heap cell and that all other operations do not allocate heap cells. However, the semantics can be instantiated with more realistic cost metrics. A difference to existing formulations of cost semantics with GC [31, 38, 30] is that we update the highwater mark when reachability changes at inner nodes of the derivation of the evaluation judgement instead of at leaves. Moreover, we use a *freelist*, which represents *named* cells available for evaluation. This alternative formulation is equivalent to the existing semantics and mainly motivated by the soundness proof of our type system for bound analysis. However, the cost semantics is a natural approach and different enough from its predecessors [38, 30] to be of interest in its own right.

Our second technical contribution is the type system for deriving bounds on the heap-space for programs with perfect GC. The type system is an extension of type-based automatic amortized resource analysis (AARA) [24, 27, 41, 21, 22, 32]. AARA is a template-based technique that introduces potential functions to efficiently and automatically reduce bound inference to linear programming. Existing type systems based on AARA can derive bounds on the highwater mark of the heap usage for programs with manual deallocation [27], but can only derive a bound on the number of total heap allocations for programs with GC [22]. This is usually a gross over-approximation of the actual memory requirement. Our extension is based on the observation that deallocation of a perfect collector can be modeled with destructive pattern matching (deallocating the matched cell) if data structures are used in a linear way. In the type system, we extend this observation to non-linear programs and use destructive pattern matching to accurately model deallocation even if data is shared.

The third technical contribution is to prove the soundness of the extended AARA with respect to the GC-based cost semantics. The proof is non-trivial and proceeds in two parts: First, we prove the soundness of the type system with respect to a semantics that copies data structures if they are shared. Second, we prove for all programs that our GC semantics uses less memory than this copying semantics. While the proofs are relatively standard, many details—like relating program states of the two semantics in the simulation proof—are quite involved. Briefly, we have to provide and maintain a mapping γ from the heap used in the GC semantics H_{gc} to *subsets* of the heap used in the copying semantics H_{copy} such that the image of H_{gc} under γ forms a partition on the second heap. The intuition is that given a cell $l \in H_{gc}$, there must be multiple cells $\gamma(l) \in H_{copy}$ that were allocated during sharing, and thus "morally the same" as l.

The analysis and the cost semantics have been implemented as an extension to Resource Aware ML (RaML) [21, 22]. RaML is an implementation of AARA for a subset of OCaml that can derive multivariate polynomial bounds. However, we restrict the technical development in this paper to a simple first-order language with tuples and lists. The proofs and ideas carry over to the more complex case of RaML.[2] An experimental evaluation shows that the system is able to derive tight symbolic heap-space bounds for common algorithms. Our results suggest that our new analysis provides asymptotic bound improvements to several classes of commonly used functions and programming patterns. We examine the reasons for these improvements and design decisions throughout the system.

[1] We are not considering memory fragmentation, which can be avoided using a copying collector.

[2] An exception are function closures that we discuss in the Section 7.

BTypes	τ ::=		Exp	e ::=	
	nat	nat		$\texttt{var}(x)$	x
	unit	unit		$\texttt{nat}[n]$	$\overline{n}$
	bool	bool		unit	$()$
	$\texttt{prod}(\tau_1;\tau_2)$	$\tau_1 \times \tau_2$		T	T
	$\texttt{list}(\tau)$	$L(\tau)$		F	F
				$\texttt{if}(x;e_1;e_2)$	$\texttt{if}\,x\,\texttt{then}\,e_1\,\texttt{else}\,e_2$
FTypes	ρ ::=			$\texttt{ap}(f;x)$	$f(x)$
	$\texttt{arr}(\tau_1;\tau_2)$	$\tau_1 \to \tau_2$		$\texttt{tpl}(x_1;x_2)$	$\langle x_1, x_2\rangle$
				$\texttt{match}_{\texttt{P}}(x_1,x_2.e_1)$	$\texttt{match}\ p\ \{(x_1;x_2) \hookrightarrow e_1\}$
Val	v ::=			nil	$[]$
	$\texttt{val}(n)$	n		$\texttt{cons}(x_1;x_2)$	$x_1 :: x_2$
	$\texttt{val}(\texttt{T})$	T		$\texttt{match}_{\texttt{L}}\{l\}(e_1;x,xs.e_2)$	$\texttt{match}\,l\,\{\texttt{nil} \hookrightarrow e_1 \mid \texttt{cons}(x;xs) \hookrightarrow e_2\}$
	$\texttt{val}(\texttt{F})$	F		$\texttt{let}(e_1;x:\tau.e_2)$	$\texttt{let}\ x = e_1\ \texttt{in}\ e_2$
	$\texttt{val}(\texttt{Null})$	Null		$\texttt{share}(x;x_1,x_2.e)$	$\texttt{share}\ x\ \texttt{as}\ x_1,x_2\ \texttt{in}\ e$
	$\texttt{val}(l)$	l			
	$\texttt{val}(\texttt{pair}(v_1;v_2))$	$\langle v_1, v_2\rangle$			

Figure 1: Simple Types, Values, and Expressions

2 Setting the Stage

In the technical part of the paper, we focus our attention to a first-order, strictly evaluated functional language. One can think of this language as a simple subset of OCaml or SML. The only recursive data type in the language is the list type. However, our work extends to the expected algebraic data types definable in RaML. Being first order, the language does not allow arbitrary local functional definitions. Instead, all functions are defined at the top level and are mutually recursive by default. The types of these functions form a signature for the program, and the semantics and typing judgments will be indexed by this signature. Thus, the function types of the language can be expressed as arrows between zero-order (base) types. Types are formally defined in Figure 1. Like in all grammars, we provide the abstract (left) and concrete (right) syntax for every type former [19]. A *signature* $\Sigma : \mathsf{Var} \to \mathsf{FTypes}$ is a map from variables to first-order types. A *program* P is a Σ indexed map from Var to pairs $(y_f, e_f)_{f\in\Sigma}$, where $\Sigma(y_f) = \tau \to \tau'$, and $\Sigma; y_f : \tau \vdash e_f : \tau'$ (the type system is discussed in Section 4). We write $P : \Sigma$ to mean P is a program with signature Σ.

To simplify the presentation, the expressions of our language (see Figure 1) are in *let normal form* (also *A normal form*). The one nonstandard construct is $\texttt{share}\ x\ \texttt{as}\ x_1, x_2\ \texttt{in}\ e$, which we will explain in more detail in the following sections. We introduce two distinct notions of *linearity*, one on the syntactic level, and one on the semantic level. Syntactic linearity is linearity in expression variables, while semantic linearity is linearity in locations (defined below). We say that a semantics is linear if it respects semantic linearity.

In line with previous works on space cost semantics [38, 30], we employ a heap, which persistently binds locations to values (normalized terms). As usual, we derive the cost of a (terminating) program from the number of heap locations used during execution, which in our case is the *maximum difference* between the sizes of the initial and final freelist. We let Loc be an infinite set of names for addressing the heap. For the rest of the paper, we use the following: $\mathsf{Stack} \triangleq \{V \mid V : \mathsf{Var} \to \mathsf{Val}\}$ and $\mathsf{Heap} \triangleq \{H \mid H : \mathsf{Loc} \to \mathsf{Val}\}$ for the set of stacks and heaps respectively.

Reachability Before we define the rules for the cost semantics, we relate the heap locations to values with the 3-place reachability relation $reach(H, v, L)$ on $\mathsf{Heap} \times \mathsf{Val} \times \wp(\mathsf{Loc})$, where $\wp$ is the powermultiset. This is read as "under heap H, the value v reaches the multiset of locations L". Write $L = reach_H(v)$ to indicate this is a functional relation justified by the (valid) mode $(+, +, -)$. We say that the reachable set of v is L.

$$\frac{A = reach_H(v_1) \qquad B = reach_H(v_2)}{A \uplus B = reach_H(\langle v_1, v_2\rangle)} \qquad \frac{A = reach_H(H(l))}{\{l\} \uplus A = reach_H(l)} \qquad \frac{v \in \mathbb{N} \cup \{\texttt{T}, \texttt{F}, \texttt{Null}\}}{\emptyset = reach_H(v)}$$

In the rules, $\uplus$ is multiset union. L is a multiset because we need to keep track of the *number* of ways a location might be reached in order to prove soundness. However, the cost semantics can be read by truncating any multiset to a set. Furthermore, we will sometimes mix multiset and set operations as the situation calls for. For example, we will write $l \in S$ for a multiset S if $S(l) \geq 1$. Complete definitions and notations can be found in the appendix.

The notion of reachability naturally lifts to expressions and contexts:

$$reach_H(V) = \biguplus_{x \in dom(V)} reach_H(V(x)) \qquad\qquad locs_{V,H}(e) = reach_H(V \restriction_{FV(e)})$$

Where $FV : \mathsf{Exp} \to \mathcal{P}(\mathsf{Var})$ denotes the set of free-variables of expressions as usual.

Towards the Garbage Collection Cost Semantics Now we are ready to give a first attempt to modeling the cost semantics for a tracing garbage collector. Before we present our new semantics, we explain an existing cost semantics we experimented with [30]. Judgements have the form $V, H, R \vdash e \Downarrow^s v, H'$, which can be read as follows. Under stack $V \in \mathsf{Stack}$, heap $H \in \mathsf{Heap}$, and continuation set $R \subseteq \mathsf{Loc}$, e evaluates to v and H' using s heap locations. The idea is that R keeps track of the set of locations necessary to complete the evaluation *after* e is evaluated (hence the name continuation). For example, we have the let rule:

$$\frac{V, H, R \uplus locs_{V,H}(x.e_2) \vdash e_1 \Downarrow^{s_1} v_1, H_1 \qquad V[x \mapsto v_1], H, R \vdash e_2 \Downarrow^{s_2} v_2, H_2}{V, H, R \vdash \mathtt{let}(e_1; x : \tau.e_2) \Downarrow^{\max s_1, s_2} v_2, H_2}$$

Notice that to evaluate e_1, we have to extend the continuation R with locations in e_2, which will be used *after* e_1 is evaluated. The total space used is the max of the component, indicating that locations used for e_1 can be reused for e_2. This is clear when we look at the variable rule:

$$\frac{V(x) = v}{V, H, R \vdash x \Downarrow^{|dom(R \uplus reach_H(v))|} v, H}$$

It states that evaluating a variable x requires the locations reachable from x as well as the continuation set R. While this way of counting heap locations does model a tracing garbage collector, it is not compatible with the existing type systems for amortized analysis. In these systems, such as RaML, the type rules count the heap locations as data is created, i.e. at each data constructor. Thus looking up a variable incurs no cost, since it was accounted for during creation. On the other hand, the cost of indexing a variable in the semantics includes the cost of the entire continuation set, which is potentially unbounded. This mismatch between the dynamics and statics of language prevents us from proving the soundness of the analysis. We give a new cost semantics that is 1) compatible with the type system and 2) also a more concrete model of a garbage collector since costs are realized with explicit locations.

3 Garbage Collection Cost Semantics

In this section, we present our novel cost semantics by combining *freelist semantics* from [25] with the cost semantics for modeling perfect GC [30] that we discussed in the previous section. The resulting semantics, called $\mathcal{E}_{\mathsf{gc}}$, is well suited for proving the soundness of the novel type-based bound analysis.

The garbage collection cost semantics $\mathcal{E}_{\mathsf{gc}}$ is defined by a collection of judgement of the form

$$\mathcal{C} \vdash_{P:\Sigma} e \Downarrow v, H', F'$$

Where $\mathcal{C} \in \mathsf{Stack} \times \mathsf{Heap} \times \wp(\mathsf{Loc}) \times \mathcal{P}(\mathsf{Loc})$ is a *configuration* usually written with variables V, H, R, F. Because the signature Σ for the mapping of function names to first-order functions does not change during evaluation, we drop the subscript $P : \Sigma$ from $\vdash_{P:\Sigma}$ when the context of evaluation is clear. Given a configuration $\mathcal{C} = (V, H, R, F)$, the evaluation judgment states that under stack V, heap H, continuation (multi)set

R, freelist F, and program P with signature Σ, the expression e evaluates to value v, and engenders a new heap H' and freelist F'. In comparison with the attempt from the previous section, the key ingredient we added is the freelist, which serves as the set of available locations. Similar to the predicate *reach*, We call R a (multi)set since the fact that it's a multiset is only useful during the soundness proof. For evaluation, it is convenient to just view R as a set. Define a *computation* as a pair $(\mathcal{C}, e)$ of a configuration $\mathcal{C}$ and an expression e. Next, we give some coherence conditions to a configuration. For a configuration (V, H, R, F), denote the garbage w.r.t. a set of locations L as $\mathit{collect}(R, L, H, F) = \{l \in H \mid l \notin F \cup R \cup L\}$.

Definition 1. A configuration (V, H, R, F) is well-formed if

1. $\mathit{dom}(H) \subseteq \mathit{reach}_H(V) \cup R \cup F$
2. $\mathit{reach}_H(V) \cup R \subseteq \mathit{dom}(H) \setminus F$
3. $\mathit{collect}(R, \mathit{reach}_H(V), H, F) = \emptyset$

Furthermore, this condition is invariant under evaluation:

Lemma 1. *Given a well-formed context (V, H, R, F) and $V, H, R, F \vdash e \Downarrow v, H', F'$, we have $\mathit{dom}(H') \subseteq \mathit{reach}_{H'}(v) \cup R \cup F'$, $\mathit{reach}_{H'}(v) \cup R \subseteq \mathit{dom}(H') \setminus F'$, and $\mathit{collect}(R, \mathit{reach}_{H'}(v), H', F') = \emptyset$.*

The well-formed conditions ensure the stack and continuation sets are within the active region of the heap $H \setminus F$, and that the active region of the heap does not contain garbage – all garbage locations are already in the freelist. From now on, all configurations are implicitly assumed to be coherent in the sense defined above.

The semantics $\mathcal{E}_{\texttt{gc}}$ is designed to model the heap usage of a program running with a tracing counting garbage collector: whenever a heap cell becomes unreachable from the root set, it becomes collected and added to the freelist as available for reallocation. As before, the continuation set R represents the set of locations required to compute the continuation *excluding* the current expression. We define the *root set* as the union of the locations in the continuation set R and the locations in the current expression e.

The inference rules for the semantics are given in Figure 2. For example, the rule F:CondT states that, to evaluate a conditional, look in the stack for the value of the branching boolean. In the case it is true, we proceed to evaluate the first branch. Furthermore, we collect cells in the heap that are not reachable from the root set $(R \cup \mathit{locs}_{V',H}(e_1))$ or already in the current free-list F, and add them (g) to the available cells for evaluating e_1.

Another example is the rule F:Let for let expressions: to evaluate the expressions $\texttt{let}(e_1; x{:}\tau.e_2)$, we evaluate the first expression with the corresponding restricted stack V_1 and an expanded continuation set R'. The extra locations come from the free variables of e_2 (not including the bound variable x), which we cannot collect during the evaluation of e_1. Next, we restrict the extended stack to only free variables of e_2, and evaluate e_2 with this stack and the original continuation set R. The other rules are similar.

Note that in contrast to the semantics in the previous section, evaluating a variable does not incur any cost. This ensures that we will be able prove the soundness of the type system. Also, since we don't allow local function definitions, we do not create closures during evaluation. Also note that we restrict the domain of the stack to the appropriate variables during evaluation. This is only to facilitate the proof of the linearity of the copying semantics introduced later, and not necessary for the implementation.

For example, we can implement the `append` and `appTwice` function, which has variable sharing. First, we analyze the heap usage of `append` under $\mathcal{E}_{\texttt{gc}}$. We case on the first component of the input. In case it's nil, we just return `l2`, and there are no allocations or deallocations. In case it's cons of `x` and `xs`, we need to allocate one heap location for the cons cell binding `x` and the recursive result, for which we can use the just matched-on cell. Again, the net overhead is zero. Thus, the total space overhead of `append` is zero.

For `appTwice`, we first share the list `l` as `l1` and `l2`. In the first `let`, the locations in `l2` are added to the continuation set, which prevents the first call to `append` from destructing `l1`. Thus size of `l1` new locations are allocated from the freelist to construct `l1'`. The second call has no net increase in heap allocations since `l2` can be destructed along the way. The return value is a pair which is stack-allocated and doesn't require a heap allocation. Thus, the total space overhead for `appTwice` is size of the input list `l`.

$$\frac{\begin{array}{c}V_1 = V\restriction_{FV(e_1)} \qquad R' = R \cup locs_{V,H}(\mathtt{lam}(x:\tau.e_2)) \qquad V_1, H, R', F \vdash e_1 \Downarrow v_1, H_1, F_1 \\ V_2 = (V[x \mapsto v_1])\restriction_{FV(e_2)} \qquad g = \{l \in H_1 \mid l \notin F_1 \cup R \cup locs_{V_2,H_1}(e_2)\} \qquad V_2, H_1, R, F_1 \cup g \vdash e_2 \Downarrow v_2, H_2, F_2\end{array}}{V, H, R, F \vdash \mathtt{let}(e_1; x:\tau.e_2) \Downarrow v_2, H_2, F_2} \text{(F:Let)}$$

$$\frac{\begin{array}{c}V(x) = \mathtt{T} \qquad V' = V\restriction_{FV(e_1)} \\ g = \{l \in H \mid l \notin F \cup R \cup locs_{V',H}(e_1)\} \qquad V', H, R, F \cup g \vdash e_1 \Downarrow v, H', F'\end{array}}{V, H, R, F \vdash \mathtt{if}(x; e_1; e_2) \Downarrow v, H', F'} \text{(F:CondT)}$$

$$\frac{\begin{array}{c}V(x) = \mathtt{F} \qquad V' = V\restriction_{FV(e_2)} \\ g = \{l \in H \mid l \notin F \cup R \cup locs_{V',H}(e_2)\} \qquad V', H, R, F \cup g \vdash e_2 \Downarrow v, H', F'\end{array}}{V, H, R, F \vdash \mathtt{if}(x; e_1; e_2) \Downarrow v, H', F'} \text{(F:CondF)}$$

$$\frac{\begin{array}{c}V(x) = v' \qquad P(f) = (y_f, e_f) \\ V' = ([y_f \mapsto v'])\restriction_{FV(e_f)} \qquad g = \{l \in H \mid l \notin F \cup R \cup locs_{V,H}(e_f)\} \qquad V'H, R, F \cup g \vdash e_f \Downarrow v, H', F'\end{array}}{V, H, R, F \vdash \mathtt{ap}(f; x) \Downarrow v, H', F'} \text{(F:App)}$$

$$\frac{}{V, H, R, F \vdash \mathtt{nil} \Downarrow \mathtt{val}(\mathtt{Null}), H, F} \text{(F:Nil)} \qquad \frac{v = \langle V(x_1), V(x_2)\rangle \qquad l \in F \qquad H' = H\{l \mapsto v\}}{V, H, R, F \vdash \mathtt{cons}(x_1; x_2) \Downarrow l, H', F \setminus \{l\}} \text{(F:Cons)}$$

$$\frac{\begin{array}{c}V(x) = \mathtt{Null} \\ V' = V\restriction_{FV(e_1)} \qquad g = \{l \in H \mid l \notin F \cup R \cup locs_{V',H}(e_1)\} \qquad V', H, R, F \cup g \vdash e_1 \Downarrow v, H', F'\end{array}}{V, H, R, F \vdash \mathtt{match_L}\{x\}(e_1; x_h, x_t.e_2) \Downarrow v, H', F'} \text{(F:MatNil)}$$

$$\frac{V(x) = v}{V, H, R, F \vdash x \Downarrow v, H, F} \text{(F:Var)}$$

$$\frac{\begin{array}{c}V(x) = l \qquad H(l) = \langle v_h, v_t\rangle \qquad V' = (V[x_h \mapsto v_h, x_t \mapsto v_t])\restriction_{FV(e_2)} \\ g = \{l \in H \mid l \notin F \cup R \cup locs_{V',H}(e_2)\} \qquad V', H, R, F \cup g \vdash e_2 \Downarrow v, H', F'\end{array}}{V, H, R, F \vdash \mathtt{match_L}\{x\}(e_1; x_h, x_t.e_2) \Downarrow v, H', F'} \text{(F:MatCons)}$$

$$\frac{\begin{array}{c}V(x) = v' \qquad V' = (V[x_1 \mapsto v', x_2 \mapsto v'])\restriction_{FV(e)} \\ g = \{l \in H \mid l \notin F \cup R \cup locs_{V',H}(e)\} \qquad V', H, R, F \cup g \vdash e \Downarrow v, H', F'\end{array}}{V, H, R, F \vdash \mathtt{share}(x; x_1, x_2.e) \Downarrow v, H', F'} \text{(F:Share)}$$

Figure 2: Cost Semantics for Perfect Garbage Collection

From this, we see that the minimum size for the initial freelist to successively evaluate a call to `appTwice` is exactly the length of the input. In general, we define the cost of a closed program to be the minimum size of the initial freelist that guarantees successful evaluation, which is equivalent to the cost annotation in the previous cost semantics introduced in Section 2.

4 Automatic Amortized Heap-Space Analysis with GC

Automatic Amortized Resource Analysis (AARA) The idea of AARA [24, 27, 21, 22] is to automate the potential method of amortized analysis using a type system. Types introduce potential functions that map data structures of the given type to non-negative numbers. The type rules ensure that there is always sufficient potential to cover the evaluation cost of the next step and the potential of the next program state.

To illustrate the idea, we informally explain the linear potential method for the functions in Figure 3. We will use the allocation/heap metric which simply counts the number of cons constructor calls during the evaluation.[3] With this metric, the cost of evaluating `append(l1,l2)` is m, where m is the number of cons constructors in `l1`, and the resource annotated type of `append` is $L^1(\texttt{int}) \times L^0(\texttt{int}) \xrightarrow{0/0} L^0(\texttt{int})$. This type says that to type `append(l1,l2)`, we need `l1` to have 1 potential per element, `l2` to have 0 per element, and the result will be a list with 0 potential per element. Additionally, the function uses 0 constant potential, and leaves 0 constant potential after evaluating. This translates to a bound which states that the number of allocations `append` makes is bounded by 1 times size of the first list. For `appTwice(l)`, the cost under the heap metric is $2m$, where m is the number of cons constructors in `l`. This is because we have to share the input list across two calls of `append`, which each requires lists with unit potential per element. For example, if `l` : $L^2(\texttt{int})$, then `l1` and `l2` both get 1 potential per element so that `l1` : $L^1(\texttt{int})$, `l2` : $L^1(\texttt{int})$, which covers the cost of the next 2 calls to `append`, and the resulting pair of lists both have 0 potential per element.

More generally, we can give the following types to `append` and `appTwice`:

$$\texttt{append} : L^p(\texttt{int}) \times L^q(\texttt{int}) \xrightarrow{r/r'} L^s(\texttt{int}),\ \text{where } p \geq s+1,\ q \geq s \text{ and } r \geq r'$$

$$\texttt{appTwice} : L^p(\texttt{int}) \xrightarrow{q/q'} L^r(\texttt{int}) \times L^s(\texttt{int}),\ \text{where } p \geq r+s+2 \text{ and } q \geq q'$$

Notice that the constant potentials r and q are unconstrained since the functions don't use any potential in the base cases. With AARA, the type system keeps track of this collection of constraints on resource annotations and passes them to an off-the-shelf LP-solver which finds the minimum solution. This is then translated to concrete resource bounds like the ones we derived by hand. It has been shown that this technique can be extended to polynomial potential functions, user-defined data types, and higher-order functions while still relying on linear constraint solving [21, 22].

```
let rec append (l1, l2) =
  match l1 with
  | [] -> l2
  | x::xs -> x::(append (xs, l2))

let appTwice l =
    share l as l1,l2 in
    let l1' = append (l1, []) in
    let l2' = append (l2, []) in
    (l1',l2')
```

Figure 3: Functions `append` and `appTwice`

Linear Potential Functions Before giving the type rules, we need to formalize linear potential as explained above. Since potential is associated with the *structure* of a value and not the particular heap locations, it is helpful to introduce a mapping from heap values to semantic values of a type. First, we give a denotational semantics for (define the structures of) the first-order types:

$$[\![\texttt{unit}]\!] = \{\texttt{val(Null)}\} \qquad [\![A_1 \times A_2]\!] = [\![A_1]\!] \times [\![A_2]\!]$$
$$[\![\texttt{bool}]\!] = \{\texttt{val(T)}, \texttt{val(F)}\} \qquad \texttt{nil} \in [\![L(A)]\!]$$
$$[\![\texttt{nat}]\!] = \mathbb{N} \qquad \texttt{cons}(a;l) \in [\![L(A)]\!] \text{ if } a \in [\![A]\!] \text{ and } l \in [\![L(A)]\!]$$

The meaning of each type is the least set such that the above holds. As usual, we write $[a_1, ..., a_n]$ for $\texttt{cons}(a_1;, ..., \texttt{cons}(a_n;\texttt{nil}))$.

In Figure 4 we give the judgements relating heap values to semantic values, in the form $\boxed{H \vDash v \mapsto a : A}$, which can be read as follows: Under heap H, heap value v defines the semantic value $a \in [\![A]\!]$. Given a stack V, we write $H \vDash V : \Gamma$ if $dom(V) \subseteq dom(\Gamma)$ and for every $x \mapsto v \in V$, $H \vDash V(x) \mapsto a : \Gamma(x)$ for some $a \in [\![A]\!]$.

We introduce linear potential for structures corresponding to the base types. The definition of linear potential is standard [20]. Below is the grammar for resource-annotated types:

$$\begin{array}{lll} \textsf{BTypes} & A & ::= \\ & & \ldots \\ & & \texttt{list}^p(A) \quad L^p(A) \end{array} \qquad \begin{array}{lll} \textsf{FTypes} & \rho & ::= \\ & & \texttt{arr}(A_1;A_2;p;q) \quad A_1 \xrightarrow{p/q} A_2 \end{array}$$

[3]This is in contrast to the highwater mark for the GC semantics $\mathcal{E}_{\mathsf{gc}}$ that is targeted by our new analysis.

$$\frac{n \in \mathbb{Z}}{H \vDash \mathtt{val}(n) \mapsto n : \mathtt{nat}}\text{(V:ConstI)} \qquad \frac{}{H \vDash \mathtt{val}(\mathtt{Null}) \mapsto \mathtt{val}(\mathtt{Null}) : \mathtt{unit}}\text{(V:ConstI)}$$

$$\frac{A \in \mathsf{BType}}{H \vDash \mathtt{val}(\mathtt{Null}) \mapsto \mathtt{val}(\mathtt{Null}) : L(A)}\text{(V:Nil)} \qquad \frac{}{H \vDash \mathtt{val}(\mathtt{T}) \mapsto \mathtt{val}(\mathtt{T}) : \mathtt{bool}}\text{(V:True)}$$

$$\frac{}{H \vDash \mathtt{val}(\mathtt{F}) \mapsto \mathtt{val}(\mathtt{F}) : \mathtt{bool}}\text{(V:False)} \qquad \frac{H \vDash v_1 \mapsto a_1 : A_1 \qquad H \vDash v_2 \mapsto a_2 : A_2}{H \vDash \langle v_1, v_2 \rangle \mapsto \langle a_1, a_2 \rangle : A_1 \times A_2}\text{(V:Pair)}$$

$$\frac{l \in \mathsf{Loc} \qquad H(l) = \langle v_h, v_t \rangle \qquad H \vDash v_h \mapsto a_1 : A \qquad H \vDash v_t \mapsto [a_2, \dots, a_n] : L(A)}{H \vDash l \mapsto [a_1, \dots, a_n] : L(A)}\text{(V:Cons)}$$

Figure 4: Mapping Locations to Semantic Values

The intended meaning is that a list of $L^p(A)$ has p units of potential per cons cell, and a function of type $A \xrightarrow{p/q} B$ takes constant potential p to run and q is the constant potential left afterwards.

With linear potential, each component of a structure is associated with a constant amount of potential. Given a structure a in a heap H, where $H \vDash v \mapsto a : A$, we define its potential $\Phi_H(a : A)$ by recursion on A:

$$\begin{aligned}
&\Phi_H(v : A) = 0 && \text{if } A \in \{\mathtt{unit}, \mathtt{bool}, \mathtt{nat}\} \\
&\Phi_H(\langle v_1, v_2 \rangle : A_1 \times A_2) = \Phi_H(v_1 : A_1) + \Phi_H(v_2 : A_2) && \\
&\Phi_H(l : L^p(A)) = p + \Phi_H(v_h : A) + \Phi_H(v_T : L^p(A)) && \text{if } H(l) = \langle v_h, v_h \rangle
\end{aligned}$$

Write $\Phi_{V,H}(\Gamma)$ for $\Sigma_{x \in dom(V)} \Phi_H(V(x) : \Gamma(x))$.
Now define $A \curlyvee A_1, A_2, n$ as the sharing relation for resource-annotated types:

$$\begin{aligned}
&L^p(A) \curlyvee^n L^q(A_1), L^r(A_2) && \text{if } p = q + r + n \text{ and } A \curlyvee^n A_1, A_2 \\
&A \times B \curlyvee^n A_1 \times B_1, A_2 \times B_2 && \text{if } A \curlyvee^n A_1, A_2 \text{ and } B \curlyvee^n B_1, B_2 \\
&A \curlyvee^n A, A && \text{if } A \in \{\mathtt{unit}, \mathtt{bool}, \mathtt{nat}\}
\end{aligned}$$

The sharing relation captures the amount of potential needed to copy a type A where each cons node in any structure in $[\![A]\!]$ has a copying overhead n.

Type Rules The type system FO^{gc} consists of rules of the form $\boxed{\Sigma; \Gamma \,\vert\!\frac{q}{q'}\, e : A}$, read as under signature $\Sigma : \mathsf{Var} \to \mathsf{FTypes}$, typing environment $\Gamma : \mathsf{Var} \to \mathsf{BTypes}$, e has type A starting with q units of constant potential and ending with q' units.

Our type system is based on the one of classic linear AARA [24]. We give a review of the rules in Figure 5. Since we are interested in the number of heap locations, there is an implicit side condition in all rules which ensures all constants are assumed to be nonnegative.

For example, L:Cons states that to add an element to a list with p potential per element, we need $p+1$ units of constant potential: p to maintain the potential of the list, and 1 for allocating the cons cell. L:MatL states that matching on a list with type $L^p(A)$, we need to type the nil case with the same constant potentials, and we need to type the cons case with an additional p units of constant potential, since we get the spill of p from the definition of linear potential. As the last example, we look at L:Share, which states that to share a variable x of type A, we need to split the potential between A_1 and A_2, and type the rest of the expression with the two new variables $x_1 : A_1$, $x_2 : A_2$.

New Rules The new type system for programs with garbage collection replaces the rules L:MatL and L:Share. The observation is that if we ensure that locations are used linearly, we can use destructive pattern

$$\frac{}{\Sigma; x : B \mid_{q}^{q} x : B}\text{(L:Var)} \qquad \frac{\Sigma(f) = A \xrightarrow{q/q'} B}{\Sigma; x : A \mid_{q'}^{q} f(x) : B}\text{(L:Fun)} \qquad \frac{\Sigma; \Gamma \mid_{q'}^{q} e_t : B \qquad \Sigma; \Gamma \mid_{q'}^{q} e_f : B}{\Sigma; \Gamma, x : \texttt{bool} \mid_{q'}^{q} \texttt{if}\ x\ \texttt{then}\ e_t\ \texttt{else}\ e_f : B}\text{(L:Cond)}$$

$$\frac{}{\Sigma; x_1 : A_1, x_2 : A_2 \mid_{q}^{q} \langle x_1, x_2 \rangle : A_1 \times A_2}\text{(L:Pair)} \qquad \frac{\Sigma; \Gamma, x_1 : A_1, x_2 : A_2 \mid_{q'}^{q} e : B}{\Sigma; \Gamma, x : (A_1, A_2) \mid_{q'}^{q} \texttt{match}\ x\ \{(x_1; x_2) \hookrightarrow e\} : B}\text{(L:MatP)}$$

$$\frac{}{\Sigma; \emptyset \mid_{q}^{q} \texttt{nil} : L^p(A)}\text{(L:Nil)} \qquad \frac{}{\Sigma; x_h : A, x_t : L^p(A) \mid_{q}^{q+p+1} \texttt{cons}(x_h; x_t) : L^p(A)}\text{(L:Cons)}$$

$$\frac{\Sigma; \Gamma \mid_{q'}^{q} e_1 : B \qquad \Sigma; \Gamma, x_h : A, x_t : L^p(A) \mid_{q'}^{q+p} e_2 : B}{\Sigma; \Gamma, x : L^p(A) \mid_{q'}^{q} \texttt{match}\ x\ \{\texttt{nil} \hookrightarrow e_1 \mid \texttt{cons}(x_h; x_t) \hookrightarrow e_2\} : B}\text{(L:MatL)}$$

$$\frac{A \curlyvee A_1, A_2 \qquad \Sigma; \Gamma, x_1 : A_1, x_2 : A_2 \mid_{q'}^{q} e : B}{\Sigma; \Gamma, x : A \mid_{q'}^{q} \texttt{share}\ x\ \texttt{as}\ x_1, x_2\ \texttt{in}\ e : B}\text{(L:Share)} \qquad \frac{\Sigma; \Gamma_1 \mid_{p}^{q} e_1 : A \qquad \Sigma; \Gamma_2, x : A \mid_{q'}^{p} e_2 : B}{\Sigma; \Gamma_1, \Gamma_2 \mid_{q'}^{q} \texttt{let}(e_1; x : \tau.e_2) : B}\text{(L:Let)}$$

Figure 5: Type Rules of Classic AARA [24]

matching to model local garbage collection by returning the potential associated with the constructor location (notice the extra $+1$ in the second premise):

$$\frac{\Sigma; \Gamma \mid_{q'}^{q} e_1 : B \qquad \Sigma; \Gamma, x_h : A, x_t : L^p(A) \mid_{q'}^{q+p+1} e_2 : B}{\Sigma; \Gamma, x : L^p(A) \mid_{q'}^{q} \texttt{match}\ x\ \{\texttt{nil} \hookrightarrow e_1 \mid \texttt{cons}(x_h; x_t) \hookrightarrow e_2\} : B}\text{(L:MatLD)}$$

This is validated by the fact (Lemma 10) that in the auxiliary copying semantics (introduced in later), once a cons-cell is matched on, there can be no live references from the root set to it, and thus we are justified in restituting the potential to type the subexpression e_2.

However, the rule L:MatLD is not sound for programs with aliasing of data. We address this issue by replacing the rule L:Share with the rule L:ShareCopy:

$$\frac{A \curlyvee^1 A_1, A_2 \qquad \Sigma; \Gamma, x_1 : A_1, x_2 : A_2 \mid_{q'}^{q} e : B}{\Sigma; \Gamma, x : A \mid_{q'}^{q} \texttt{share}\ x\ \texttt{as}\ x_1, x_2\ \texttt{in}\ e : B}\text{(L:ShareCopy)}$$

To share a variabe of type A, we need to split the potential between two new annotated types A_1 and A_2 as usual. In addition, we have to pay an "overhead" of 1 for every cons node in any structure in $[\![A]\!]$. The idea is that we treat data as if it is actually copied. This is sound w.r.t. the copying semantics because the size of the domain of the reachable set of a value v is exactly the linear potential of $v : A$ with all resource annotations set to 1.

For example, Figure 6 contains derivations for `append` and `appTwice`. Here, A is short for `int` and $\Sigma = [\mathsf{append} \mapsto L^p(A) \times L^p(A) \xrightarrow{q/q} L^p(A)]$ is the program signature.

From these derivations, we get the improved space overhead bound to `append` and `appTwice`:

$$\texttt{append} : L^p(\texttt{int}) \times L^q(\texttt{int}) \xrightarrow{r/r'} L^s(\texttt{int}),\ \text{where } p \geq s,\ q \geq s,\ \text{and } r \geq r'$$
$$\texttt{appTwice} : L^p(\texttt{int}) \xrightarrow{q/q'} L^r(\texttt{int}) \times L^s(\texttt{int}),\ \text{where } p \geq r + s + 1 \text{ and } q \geq q'$$

Cost Metrics In previous versions of AARA [27, 21], the typing judgment and cost semantics are parametrized by a *cost metric* $m : \mathsf{res\_const} \to \mathbb{Q}$, which assigns a constant cost to each step in the semantics. Recall the heap metric introduced above; formally, this is the function $k \mapsto \mathbb{1}_{k=k^{\mathsf{cons}}}$. We instantiate

$$
\dfrac{\dfrac{\Sigma(\mathsf{append}) = L^p(A) \times L^p(A) \xrightarrow{q/q} L^p(A)}{\Sigma; l2 : L^p(A), xs : L^p(A) \,\big|^{q+p+1}_{q+p+1}\, \mathsf{append}(xs, l2) : L^p(A)}\,\text{L:App} \qquad \dfrac{}{\Sigma; x : A, r : L^p(A) \,\big|^{q+p+1}_{q}\, x :: r : L^p(A)}\,\text{L:Cons}}{\Sigma; l2 : L^p(A), x : A, xs : L^p(A) \,\big|^{q+p+1}_{q}\, \mathtt{let}\,(\mathsf{append}(xs, l2); r.x :: r) : L^p(A)}\,\text{L:Let}
$$

$$
\dfrac{\dfrac{}{\Sigma; l2 : L^p(A) \,\big|^{q}_{q}\, l2 : L^p(A)}\,\text{L:Var} \qquad \vdots}{\Sigma; l1 : L^p(A), l2 : L^p(A) \,\big|^{q}_{q}\, \mathtt{match}_{\mathtt{L}}\{l1\}(l2; x, xs.\mathtt{let}\,(\mathsf{append}(xs, l2); r.x :: r)) : L^p(A)}\,\text{L:MatL}
$$

$$
\dfrac{}{\Sigma; l1' : L^r(A), l2' : L^s(A) \,\big|^{q}_{q}\, \langle l1', l2' \rangle : L^r(A) \times L^s(A)}\,\text{L:Pair}
$$

$$
\dfrac{\vdots \qquad \dfrac{\langle l2, [] \rangle : L^s(A) \times L^s(A) \qquad \Sigma(\mathsf{append}) = L^s(A) \times L^s(A) \xrightarrow{q/q} L^s(A)}{\Sigma; l2 : L^s(A) \,\big|^{q}_{q}\, \mathsf{append}(l2, []) : L^s(A)}\,\text{L:Fun}}{\Sigma; l2 : L^s(A), l1' : L^r(A) \,\big|^{q}_{q}\, e_0 : L^r(A) \times L^s(A)}\,\text{L:Let}
$$

$$
\dfrac{\dfrac{\langle l1, [] \rangle : L^r(A) \times L^r(A) \quad \Sigma(\mathsf{append}) = L^r(A) \times L^r(A) \xrightarrow{q/q} L^r(A)}{\Sigma; l1 : L^r(A) \,\big|^{q}_{q}\, \mathsf{append}(l1, []) : L^r(A)}\,\text{L:Fun} \qquad \vdots}{\Sigma; l1 : L^r(A), l2 : L^s(A) \,\big|^{q}_{q}\, \mathtt{let}\,(\mathsf{append}(l1, []); l1'.e_0) : L^r(A) \times L^s(A)}\,\text{L:Let}
$$

$$
\dfrac{L^{r+s+1}(A) \curlyvee^1 L^r(A), L^s(A) \qquad \vdots}{\Sigma; l : L^{r+s+1}(A) \,\big|^{q}_{q}\, \mathtt{share}(l; l1, l2.\,\mathtt{let}\,(\mathsf{append}(l1, []); l1'.e_0)) : L^r(A) \times L^s(A)}\,\text{L:ShareCopy}
$$

Figure 6: Type derivations for the functions `append` and `appTwice`. In the derivation for `appTwice` we write e_0 for $\mathtt{let}\,(\mathsf{append}(l2, []); l2'.\langle l1', l2' \rangle)$.

the previous type system with this metric (which only accounts for heap *allocations*), resulting in a concrete type system RaML$^{\mathsf{heap}}$. We give a full evaluation of the improvements of FO^{gc} over RaML$^{\mathsf{heap}}$ in Section 7. Although we defined the constructor to cost 1 heap location (as shown in L:Cons and L:MatLD), it can be any constant as long as the introduction and elimination rules agree on the constant. Thus we can extend the type system to accurately track constructors which vary in size depending on the argument.

Type Inference One of the benefits of AARA is efficient type inference using off-the-shelve LP solvers [24], even for non-linear potential functions [21, 22]. The new rules do not complicate inference and previous techniques still apply. In a nutshell, inference is performed in three steps: First, perform a standard Hindley-Milner type inference for the base types. Then, annotate the type derivation with (yet unknown) variables for the potential annotations and collect linear constraints that are derived from the type rules. Finally, solve the constraints with an LP solver and minimize the potential annotations of the inputs. Details can be found in previous work [24, 22].

5 Soundness of FO^{gc}

We seek to prove the following theorem.

Theorem 2 (Soundness)**.** *Let* $H \vDash V{:}\Gamma$, $\Sigma; \Gamma \,\big|^{q}_{q'}\, e : B$, *and* $V, H \vdash^{\mathcal{E}_{\mathsf{oper}}} e \Downarrow v, H'$. *Then for all configurations* W, Y, F, R, *if* $V, H \sim W, Y$ *and* $|F| \geq \Phi_{V,H}(\Gamma) + q$, *there exists a value* w, *and a freelist* F' *such that*

$$
W, Y, R, F \vdash^{\mathcal{E}_{\mathsf{gc}}} e \Downarrow w, Y', F' \qquad \textit{and} \qquad v \sim^{H'}_{Y'} w\,.
$$

Here, $\mathcal{E}_{\mathsf{oper}}$ is a standard big-step semantics, with judgments of the form $\boxed{V, H \vdash e \Downarrow v, H'}$ derived from $\mathcal{E}_{\mathsf{gc}}$, $V, H \sim W, Y$ is context equivalence, and $v \sim^{H'}_{Y'} w$ is value equivalence (these are defined below). The theorem states that, given a terminating expression and a freelist that is sufficiently large (as predicated by the type derivation), a run with $\mathcal{E}_{\mathsf{gc}}$will normalize to an equivalent value.

To facilitate the proof, we define an intermediate semantics $\mathcal{E}_{\mathsf{copy}}$ which is semantically linear. The proof has two stages: First, we show $\mathcal{E}_{\mathsf{copy}}$ over-approximates $\mathcal{E}_{\mathsf{gc}}$, meaning that any computation that succeeds with $\mathcal{E}_{\mathsf{gc}}$ will succeed with an equally-sized or smaller freelist with $\mathcal{E}_{\mathsf{gc}}$. Then we show FO^{gc} is sound with respect to $\mathcal{E}_{\mathsf{copy}}$, and thus by the previous step sound with respect to $\mathcal{E}_{\mathsf{gc}}$.

As mentioned above, we introduce a big step semantics $\mathcal{E}_{\mathsf{oper}}$ that does not use freelists or account for garbage collection. We use it to characterize expressions that normalize to values when initialized with a sufficient freelist. This technique has also been employed in earlier work on AARA [25]. In the judgment $V, H \vdash e \Downarrow v, H'$, the "freelist" is the whole ambient set of locations Loc, thus we never run out of locations during evaluation. This introduces a problem for value and context equivalence: when comparing evaluation results between a run with $\mathcal{E}_{\mathsf{copy}}$ and $\mathcal{E}_{\mathsf{oper}}$, the return values might not be syntactically equal. Consider the following expression $e = \texttt{let \_ = [4] in [5]}$. Let $\mathsf{Loc} = \mathbb{N}$, the natural numbers. Consider the evaluation $\emptyset, \emptyset, \emptyset, \{1\} \vdash^{\mathcal{E}_{\mathsf{copy}}} e \Downarrow v_1, H_1, F_1$. First, 1 is allocated and mapped to `[4]`. Then, since the first subexpression `[4]` is not used afterwards, we collect 1, and reuse it and map again to `[5]`. Thus $v_1 = 1$. In an evaluation $\emptyset, \emptyset, \emptyset, \{1\} \vdash^{\mathcal{E}_{\mathsf{oper}}} e \Downarrow v_2, H_2, F_2$, we also first map 1 to `[4]`, but then allocate a new location, say 2, and map it to `[5]`, and $v_2 = 2$. Due to the difference in allocation strategies and the fact that both are nondeterministic, we need a more robust notion of equality for values. Luckily, the structures from the denotational semantics (defined in Section 4) does the job. In both runs, the return value maps to the semantic value `[5]`. Thus we use semantical equality as the basis for value and context equivalence:

Definition 2 (Value Equivalence)**.** Two values v_1, v_2 are equivalent (with the presupposition that they are well-formed w.r.t. heaps H_1, H_2), iff $H_1 \vDash v_1 \mapsto a : A$ and $H_2 \vDash v_2 \mapsto a : A$. Write value equivalence as $v_1 \sim^{H_1}_{H_2} v_2$.

Definition 3 (Context Equivalence)**.** Two contexts $(V_1, H_1), (V_2, H_2)$ are equivalent iff $\mathit{dom}(V_1) = \mathit{dom}(V_2)$ and for all $x \in \mathit{dom}(V_1)$, $V_1(x) \sim^{H_1}_{H_2} V_2(x)$. Write context equivalence as $(V_2, H_2) \sim (V_2, H_2)$

Stated simply, two contexts are equivalent when they have the same domain and equal variables bind equal semantic values.

Linear Garbage Collection Cost Semantics To establish the soundness of the type system, we need an intermediary semantics $\mathcal{E}_{\mathsf{copy}}$, which is *semantically linear*. As mentioned in Section 2, this means that locations are treated linearly, that is, no location can be used twice in a program. Variable sharing is achieved via *copying*: the shared value is created by allocating a fresh set of locations from the freelist and copying the locations of the original value one by one. This is also sometimes referred to as deep copying. Let $\mathit{copy}(H, L, v, H', v')$ be a 5-place relation on $\mathsf{Heap} \times \mathcal{P}(\mathsf{Loc}) \times \mathsf{Val} \times \mathsf{Heap} \times \mathsf{Val}$. Similar to reachability, we write this as $H', v = \mathit{copy}(H, L, v)$ to signify the intended mode for this predicate: $(+, +, +, -, -)$.

$$\frac{v \in \{n, \texttt{T}, \texttt{F}, \texttt{Null}\}}{H, v = \mathit{copy}(H, L, v)} \qquad \frac{l' \in L \qquad H', v = \mathit{copy}(H, L \setminus \{l'\}, H(l))}{H'\{l' \mapsto v\}, l' = \mathit{copy}(H, L, l)}$$

$$\frac{\begin{array}{c} L_1 \sqcup L_2 \subseteq L \qquad |L_1| = |\mathit{dom}(\mathit{reach}_H(v_1)| \\ |L_2| = |\mathit{dom}(\mathit{reach}_H(v_2)| \qquad H_1, v_1' = \mathit{copy}(H, L_1, v_1) \qquad H_2, v_2' = \mathit{copy}(H_1, L_2, v_2) \end{array}}{H_2, \langle v_1', v_2' \rangle = \mathit{copy}(H, L, \langle v_1, v_2 \rangle)}$$

Primitives require no cells to copy; a location value is copied recursively; a pair of values is copied sequentially, and the total number of cells required is the size of the reachable set of the value. Now, consider $\mathcal{E}_{\mathsf{gc}}$ with the share rule F:Share replaced with the following rule.

$$\frac{\begin{array}{c} V(x) = v' \\ L \subseteq F \qquad |L| = |\mathit{dom}(\mathit{reach}_H(v'))| \qquad H', v'' = \mathit{copy}(H, L, v') \qquad V' = (V[x_1 \mapsto v', x_2 \mapsto v'']) \upharpoonright_{FV(e)} \\ F' = F \setminus L \qquad g = \{l \in H \mid l \notin F' \cup R \cup \mathit{locs}_{V',H}(e)\} \qquad V', H', R, F' \cup g \vdash e \Downarrow v, H'', F'' \end{array}}{V, H, R, F \vdash \texttt{share } x \texttt{ as } x_1, x_2 \texttt{ in } e \Downarrow v, H'', F''} \text{(E:Share)}$$

To share a variable, we first copy the shared value. The number of cells required is equal to the size of the reachable set from the value. This copying sharing semantics is what justifies the analysis to restitute the potential when matching on a cons node, since even if the node was shared, we had to pay for the cost by copying the node when sharing the original value. Next, we restrict the stack to the appropriate variables. Lastly, any locations not reachable from the current subexpression e are collected. This is for the case when a variable is shared but not used later.

Recall that a *computation* is a pair $(\mathcal{C}, e)$ consisting of a configuration $\mathcal{C} = (V, H, R, F)$ and an expression e. Since the cost semantics can only *preserve* the linearity of a computation, we restrict our attention to computations that are linear initially, and show that $\mathcal{E}_{\mathsf{copy}}$ respects the linearity of any initially linear computation. This motivates the following definitions:

Definition 4. (Linear context) Given a context (V, H), let $x, y \in dom(V)$, $x \neq y$, and $r_x = reach_H(V(x))$, $r_y = reach_H(V(y))$. It is *linear* given that $\mathsf{set}(r_x)$, $\mathsf{set}(r_y)$, and $r_x \cap r_y = \emptyset$.

Where $\mathsf{set}(S)$ means S a proper set $(\forall x, S(x) \leq 1)$. Denote this by $\mathsf{linearCtxt}(V, H)$. Whenever $\mathsf{linearCtxt}(V, H)$ holds, there is at most one path from a variable on the stack V to any location in H. Now we can formalize our intuition for linear computations:

Definition 5 (Linear computation)**.** Given a configuration $\mathcal{C} = (V, H, R, F)$ and an expression e, we say the 5-tuple $(\mathcal{C}, e)$ is a *computation*; it is a *linear computation* given that $dom(V) = FV(e)$, $\mathsf{linearCtxt}(V, H)$, and $\mathsf{disjoint}(\{R, F, locs_{V,H}(e)\})$. And we write $\mathsf{linearComp}(V, H, R, F, e)$ (equivalently $\mathsf{linearComp}(\mathcal{C}, e)$) to denote this fact.

Intuitively, we expect that any terminating compuation with $\mathcal{E}_{\mathsf{copy}}$ has a corresponding run with $\mathcal{E}_{\mathsf{gc}}$ that can be instantiated with an equally-sized or smaller freelist. Although this seems quite straightforward to prove, a complete characterization of the relationship between the space allocations of two runs with each semantics is necessary. To demonstrate the difficulties involved, consider the following proof attempt:

Attempt 1. *Let $\mathcal{C}_2 = (V, H, R, F)$ be a configuration and $(\mathcal{C}_2, e)$ be a linear computation. Given that $\mathcal{C}_2 \vdash^{\mathcal{E}_{\mathsf{copy}}} e \Downarrow v, H', F'$, for all configurations $\mathcal{C}_1 = (W, Y, R, M)$ such that $W, Y \sim V, H$ and $|M| = |F|$, there exists a triple $(w, Y', M') \in \mathsf{Val} \times \mathsf{Heap} \times \mathsf{Loc}$ such that*

$$\mathcal{C}_1 \vdash^{\mathcal{E}_{\mathsf{gc}}} e \Downarrow w, Y', M' \qquad \textit{and} \qquad v \sim^{H'}_{Y'} w \qquad \textit{and} \qquad |M'| \geq |F'| \ .$$

We proceed with induction on the derivation of the judgment in $\mathcal{E}_{\mathsf{copy}}$. Almost every case goes through, save for E:Let. First, we get $W_1, Y \sim V_1, H$ and we have the following from induction on the first premise:

$$W_1, Y, R', M \vdash^{\mathcal{E}_{\mathsf{gc}}} e \Downarrow w_1, Y_1, M_1 \qquad \text{and} \qquad v_1 \sim^{H_1}_{Y_1} w_1 \qquad \text{and} \qquad |M_1| \geq |F_1|$$

To instantiate the induction hypothesis on the second premise, we need to show that, among other things, $|M_1 \cup j| \geq |F_1 \cup g|$, where j is the set of collected locations in the $\mathcal{E}_{\mathsf{gc}}$ judgment. We cannot show this precisely because g might contain more cells then j due to the linearity of $\mathcal{E}_{\mathsf{copy}}$, thus preventing a piecewise comparison. But of course $|j|$ is always less than $|g|$, since $\mathcal{E}_{\mathsf{gc}}$ doesn't copy to share values! This shows that there is a mismatch between the induction hypothesis and the relationship between the sizes of the respective freelists and the garbage sets. Specifically, we need to know exactly how much larger M_1 is compared to F_1 at any given step.

Having a sense of what is missing, we formulate the criteria which characterize the required equivalence between two configurations, which we call *copy extension*.

Definition 6. A configuration $\mathcal{C}_2 = (V_2, H_2, R_2, F_2)$ is a *copy extension* of another configuration $\mathcal{C}_1 = (V_1, H_1, R_1, F_1)$ iff

1. $V_1, H_1 \sim V_2, H_2$
2. There is a proper partition $\gamma : dom(H_1) \setminus F_1 \to \mathcal{P}(dom(H_2) \setminus F_2)$ such that for all $l \in dom(\gamma)$, $|\gamma(l)| = reach_{H_1}(V_1)(l) + R_1(l)$

3. For all $l \in dom(\gamma)$, $x \in dom(V_1)$, sequence of directions P which is valid w.r.t. $V_1(x)$, $|reach_{H_2}(V_2(x; P)) \cap \gamma(l)| = reach_{H_1}(V_1(x; P))(l)$.

4. For all $l \in dom(\gamma)$, $|\gamma(l) \cap R_2| = R_1(l)$

5. $|F_1| = |F_2| + |\oslash(\gamma)|$, where $\oslash(\gamma) = \bigcup_{P \in ec(\gamma)} P \setminus \{rep(P)\}$

Write this as $\mathcal{C}_1 \preceq \mathcal{C}_2$.

The intention is that $\mathcal{C}_2$ is a configuration for an evaluation using $\mathcal{E}_{\mathsf{copy}}$, and $\mathcal{C}_1$ a configuration for $\mathcal{E}_{\mathsf{gc}}$. The first condition is the straightforward context equivalence. The second condition requires the existence of a mapping γ that tells us given a location in $H_1 \setminus F_1$, which locations in $H_2 \setminus F_2$ are shared instances.

For example, consider the expression $\mathtt{share}\ x\ \mathtt{as}\ x_1, x_2\ \mathtt{in}\ e$ and assume the stack is $[\mathtt{x} \mapsto 1]$, and the heap equals $[1 \mapsto \langle 0, \mathtt{Null}\rangle]$, i.e. $\mathtt{x}$ is the list $\mathtt{[0]}$. In an evaluation with $\mathcal{E}_{\mathsf{gc}}$, the stack becomes $[\mathtt{x1} \mapsto 1, \mathtt{x2} \mapsto 1]$, and the heap does not change. With $\mathcal{E}_{\mathsf{copy}}$, we allocate a new location in the heap: $[1 \mapsto \langle 0, \mathtt{Null}\rangle, 2 \mapsto \langle 0, \mathtt{Null}\rangle]$, and the stack changes accordingly: $[\mathtt{x1} \mapsto 1, \mathtt{x2} \mapsto 2]$. Now γ would map 1 to $\{1, 2\}$, since both are shared instances of the former.

Thus, the image of γ is a collection of disjoint subsets whose union is $dom(H_2) \setminus F_2$, and each location in $dom(H_2) \setminus F_2$ belongs to a unique class whose preimage is the unique representative in $dom(H_1) \setminus F_1$. Furthermore, we noticed it is crucial to include the fact that the size of $\gamma(l)$ must be the sum of the number of references from the stack and the continuation set. Furthermore, we also require each subset $\gamma(l)$ (also referred to as class) to be nonempty (this is the *proper* partition condition).

While γ gives us a relation between the active regions of two respective heaps, we still need to know exactly how variables on the stack factor in this relationship. Let $l \in H_1$. Specifically, we need to know that the number of references to l from every *sub*value in V_1 is equal to the size of the corresponding part of the class $\gamma(l)$. First, we need to access subvalues of a value using directions:

Definition 7. Let Dir be the set $\{\mathsf{L}, \mathsf{R}, \mathsf{N}\}$, denoting left, right, and next respectively. We define the function $get_H : (1 \oplus \mathsf{Val}) \times \mathsf{Dir} \to 1 \oplus \mathsf{Val}$ which indexes values via directions:

$$\begin{array}{rclcrcl} get_H(Just(\langle v_1, v_2\rangle), \mathsf{L}) & = & Just(v_1) & \quad & get_H(Just(l), \mathsf{N}) & = & Just(H(l)) \\ get_H(Just(\langle v_1, v_2\rangle), \mathsf{R}) & = & Just(v_2) & \quad & get_H(\_, \_) & = & None \end{array}$$

Let $P \in \mathcal{S}(\mathsf{Dir})$, where $\mathcal{S}(X)$ denotes the set of sequence with elements from X. We define $find_H : (1 \oplus \mathsf{Val}) \times \mathcal{S}(\mathsf{Dir}) \to 1 \oplus \mathsf{Val}$ extending get_H to sequences of directions:

$$find_H(v, D :: P) = find_H(get_H(v, D), P)$$
$$find_H(v, []) = v$$

Call P valid w.r.t. a value v if $find_H(v, P) = Just(v')$ for some v'. Given a valid sequence P w.r.t. $V(x)$, write $V_H(x; P)$ for $fromJust(find_H(V(x), P))$ and $reach_H(V(x; P))$ for $reach_H(V_H(x; P))$. A map $m_V : X \to \mathcal{S}(\mathsf{dir})$ is a *subvalue map* given that $X \subseteq dom(V)$ and each $x \in X$ is mapped to a sequence P which is valid w.r.t. $V(x)$. Given a subvalue map m_V, define its action as $reachPath_{V,H}(X, m) = \biguplus_{x \in X} reach_H(V(x; m(x)))$.

With this, the third condition gives us a more fine grained restriction: for any subvalue in V_1, the number of references from it to l is equal to the size of the intersection of the reachable set of the corresponding subvalue in V_2 with the appropriate class $\gamma(l)$.

The next condition simply states that the continuation sets respect γ. Lastly, we have that F_1 is greater than F_2, with the overhead $\oslash(\gamma)$ being exactly the sum $\sum_{l \in \gamma} |\gamma(l)| - 1$. Here $ec(\gamma)$ is the image of γ: $\{\gamma(l) \mid l \in dom(\gamma)\}$. Since each class $\gamma(l)$ is non-empty, we use $rep(l)$ to choose an arbitrary element from the class.

Below are some expected properties of a copy extension:

Lemma 3. *Let $V_1, H_1 \sim V_2, H_2$. Then for all $x \in dom(V_1)$ and sequence of directions P, Either $find_{H_1}(V_1(x), P) = find_{H_2}(V_2(x), P) = None$ or $find_{H_1}(V_1(x), P) = v_1$, $find_{H_2}(V_2(x), P) = v_2$ and $v_1 \sim^{H_1}_{H_2} v_2$*

Lemma 4. *Let $V_2, H_2, R_2, F_2 \vdash^{\mathsf{copy}} e \Downarrow v, H', F'$, and $V_1, H_1, R_1, F_1 \preceq V_2, H_2, R_2, F_2$, where γ is the partition satisfying the copy extension property. Then for all $l \in dom(\gamma)$ and subvalue map $m_V : X \to \mathcal{S}(\mathsf{dir})$, $\gamma(l) \subseteq collect(R_2, reachPath_{V_2,H_2}(X,m), H_2, F_2)$ iff $l \in collect(R_1, reachPath_{V_1,H_1}(X,m), H_1, F_1)$.*

Now we can state the key lemma:

Lemma 5. *Let $(\mathcal{C}_2, e) = (V, H, R, F, e)$ be a linear computation. Given that $\mathcal{C}_2 \vdash^{\mathcal{E}_{\mathsf{copy}}} e \Downarrow v, H', F'$, for all configurations $\mathcal{C}_1$ such that $\mathcal{C}_1 \preceq \mathcal{C}_2$, there exists a triple $(w, Y', M') \in \mathsf{Val} \times \mathsf{Heap} \times \mathsf{Loc}$ and $\gamma' : dom(Y') \setminus M' \to \mathcal{P}(dom(H') \setminus F')$ s.t.*

1. *$\mathcal{C}_1 \vdash^{\mathcal{E}_{\mathsf{gc}}} e \Downarrow w, Y', M'$*
2. *$v \sim^{H'}_{Y'} w$*
3. *γ' is a proper partition, and for all $l \in dom(\gamma')$, $|\gamma'(l)| = |reach_{Y_1}(w_1)(l)| + S(l)$*
4. *For all $l \in dom(\gamma')$ and $P \in \mathcal{S}(\mathsf{Dir})$ that is valid w.r.t. v, $|reach_{H'}(find_{H'}(v; P)) \cap \gamma'(l)| = reach_{Y'}(find_{Y'}(w; P))(l)$*
5. *For all $l \in dom(\gamma')$, $\gamma'(l) \cap R = \gamma(l) \cap R$*
6. *$|M'| = |F'| + |\oslash(\gamma')|$*

Proof. Induction on the evaluation judgment $\vdash^{\mathcal{E}_{\mathsf{copy}}}$. We illustrate the ideas with the case CondT:

$$\textbf{Case: } \frac{V' = V\restriction_{FV(e_1)} \qquad g = \{l \in H \mid l \notin F \cup R \cup locs_{V',H}(e_1)\} \qquad \overset{V(x) = \mathtt{T}}{V', H, R, F \cup g \vdash e_1 \Downarrow v, H', F'}}{V, H, R, F \vdash \mathtt{if}(x; e_1; e_2) \Downarrow v, H', F'}$$

Let W, Y, S, M be a configuration such that $W, Y, S, M \preceq V, H, R, F$. Define $W' = W\restriction_{dom(V')}$ and $j = \{l \in Y | l \notin M \cup S \cup locs_{W,Y}(e_1)\}$. To instantiate the induction hypothesis, we need to show that $W', Y, S, M \cup j \preceq V', H, R, F \cup g$, giving us 5 obligations:

(1) $W', Y \sim V', H$
(2) Give a proper partition $\gamma' : dom(Y) \setminus (M \cup j) \to \mathcal{P}(dom(H) \setminus (F \cup g))$
(3) For all $l \in dom(\gamma'), x \in dom(W')$, $P \in \mathcal{S}(\mathsf{Dir})$ a valid sequence w.r.t. $W'(x)$, $|reach_H(V'(x; P)) \cap \gamma'(l)| = reach_Y(W'(x; P))(l)$
(4) For all $l \in S, |\gamma'(l) \cap R| = S(l)$
(5) $|M \cup j| = |F \cup g| + |\oslash(\gamma')|$

(1) is satisfied since $W, Y \sim V, H$. For *(2)*, we take $\gamma'(l) = \gamma\restriction_{dom(Y)\setminus(M\cup j)}(l) \setminus g$. First, we show that γ' is a partition. Let $l, l' \in dom(Y) \setminus (M \cup j)$ be two arbitrary locations. Then $\gamma'(l) \cap \gamma'(l') = \emptyset$ since γ is a partition. Now Consider the image of γ':

$$\begin{aligned}
\gamma'(dom(Y) \setminus (M \cup j)) &= \gamma(dom(Y) \setminus (M \cup j)) \setminus g \\
&= ((\bigcup_{l \in dom(Y)\setminus M} \gamma(l)) \setminus \bigcup_{l \in j} \gamma(l)) \setminus g \\
&= ((dom(H) \setminus F) \setminus \bigcup_{l \in j} \gamma(l)) \setminus g && (\gamma \text{ is a partition}) \\
&= (dom(H) \setminus F) \setminus g && (\textstyle\bigcup_{l\in j} \gamma(l) \subseteq g \text{ by Lemma 4}) \\
&= dom(H) \setminus (F \cup g)
\end{aligned}$$

Hence γ' is a partition. Next we need to show it is proper, or that every class is nonempty. Let $l \in dom(Y) \setminus (M \cup j)$ be any location. Since $\gamma'(l) = \gamma(l) \setminus g$, it suffices to show that the class $\gamma(l)$ is not all collected. For the sake of contradiction, assume $\gamma(l) \subseteq g$. But then $l \in j$ by Lemma 4, and we have a contradiction since we assumed $l \notin j$. Lastly, $|\gamma'(l)| = reach_Y(W')(l) + S(l)$ follows from the definition of g and the second and third condition of copy extension. Conditions *(3)* and *(4)* follow similarly.

Lastly, we need to show that the overhead is preserved: $|M| + |j| = |F| + |g| + |\oslash(\gamma')|$. By assumption, $|M| = |F| + |\oslash(\gamma)|$, so it suffices to show $|j| + |\oslash(\gamma)| = |g| + |\oslash(\gamma')|$. Since g and the image of γ' are disjoint, it suffices to define a bijection $f : j \oplus \oslash(\gamma) \to g \sqcup \oslash(\gamma')$. First, we separate the classes $ec(\gamma)$ into those that are completely collected into g and those that are only partially collected: $\mathcal{C}_1 = \{\gamma(l) \mid l \in j\}$ and $\mathcal{C}_2 = ec(\gamma) \setminus \mathcal{C}_1$. Further, let $D_1 = \bigcup_{C\in\mathcal{C}_1} C \setminus \{rep(C)\}$ and $D_2 = \bigcup_{C\in\mathcal{C}_2} C \setminus \{rep(C)\}$. Then we have $\oslash(\gamma) = D_1 \sqcup D_2$ and by Lemma 4, $g = (\bigcup_{C\in\mathcal{C}_1} C) \sqcup L$ for some L. Therefore, we need to find a bijection $f : j \oplus (D_1 \sqcup D_2) \to (\bigcup_{C\in\mathcal{C}_1} C) \sqcup L \sqcup \oslash(\gamma')$. It suffices to find the bijections $f_1 : j \oplus D_1 \to \bigcup_{C\in\mathcal{C}_1} C$ and $f_2 : D_2 \to L \sqcup \oslash(\gamma')$. First, we define f_1:

$$f_1(x) = \begin{cases} rep(\gamma(l)) & x = (\mathsf{inl}, l) \\ l & x = (\mathsf{inr}, l) \end{cases}$$

Note that f_1 simply maps the set of representatives into their respective classes and is the identity on the rest of the class. Thus f_1 is a bijection, and $|j| + |D_1| = |\bigcup_{C\in\mathcal{C}_1} C|$. Next, note that

$$|\mathcal{C}_2| = |ec(\gamma) \setminus \{\gamma(l) \mid l \in j\}| = |ec(\gamma \restriction_{dom(Y)\setminus(M\cup j)})| = |ec(\gamma')|$$

Which means that $\mathcal{C}_2$ has the same number of classes as γ' (even though the actual classes might be different). Since both γ and γ' are proper partitions, we can keep class representatives for each class when defining the bijection:

$$\begin{aligned} &|D_2| = |L \sqcup \oslash(\gamma')| \\ \iff& |\bigcup_{C\in\mathcal{C}_2} C \setminus \{rep(C)\}| = |L \sqcup \bigcup_{C\in ec(\gamma')} C \setminus \{rep(C)\}| \\ \iff& |\bigcup_{C\in\mathcal{C}_2} C| = |L \sqcup \bigcup_{C\in ec(\gamma')} C| \end{aligned}$$

In fact, the latter two sets are equal. Let $l \in \bigcup_{C\in\mathcal{C}_2} C$. If l is collected into g, l must be in L since it is not from a class in $\mathcal{C}_1$. If l is not collected, it remains in $\bigcup_{C\in ec(\gamma')} C$. For the other direction, let $l \in L \sqcup \bigcup_{C\in ec(\gamma')} C$. If $l \in L$, then it is in a class that was not completely collected, which means $l \in \bigcup_{C\in\mathcal{C}_2} C$. Otherwise, l is in a class that is disjoint from g, which means again that $l \in \bigcup_{C\in\mathcal{C}_2} C$. Hence f_2 is simply the identity.

Finally, we have the copy extension $W', Y, S, F \cup j \preceq V', H, R, F \cup g$. Now we instantiate the induction hypothesis to obtain the triple (w, Y', M') and new partition γ'' with the expected properties. Applying the rule F:CondT to the evaluation $W', Y, S, F \cup j \vdash e_1 \Downarrow w, Y', M'$ given by the first property from induction, we get $W, Y, S, F \vdash \texttt{if}(x; e_1; e_2) \Downarrow w, Y', M'$, which inherits the required conditions *(2)* - *(6)* from induction. ☐

Thus, we have shown that we can execute a computation using $\mathcal{E}_{\mathsf{gc}}$ given that the computation suceeded in a run with $\mathcal{E}_{\mathsf{copy}}$, which means that $\mathcal{E}_{\mathsf{copy}}$ is an over approximation of $\mathcal{E}_{\mathsf{gc}}$.

Soundness of FO^{gc} For the second part of the proof, we show FO^{gc} is sound w.r.t. $\mathcal{E}_{\mathsf{copy}}$. Below are selected lemmas used in the soundness theorem:

Definition 8 (Stability)**.** Given heaps H, H', a set of locations is *stable* if $\forall l \in R.\ H(l) = H'(l)$. Denote this by $\mathsf{stable}(R, H, H')$.

Lemma 6. *Let $H \vDash v \mapsto a : A$. For all sets of locations R, if $reach_H(v) \subseteq R$ and $\mathsf{stable}(R, H, H')$, then $H' \vDash v \mapsto a : A$ and $reach_H(v) = reach_{H'}(v)$.*

Lemma 7 (Stability of copying)**.** *Let $H', v' = copy(H, L, v)$. For all $l \in H$, if $l \notin L$, then $H(l) = H'(l)$. Further, $reach_{H'}(v') \subseteq L$.*

Lemma 8 (Copy is copy)**.** *Let $H', v' = copy(H, L, v)$. If $H \vDash v \mapsto a : A$, then $H' \vDash v' \mapsto a : A$.*

Lemma 9. *Let $A \curlyvee^n A_1, A_2$, $H \vDash v : A$, $v \sim_H^H v_1$, and $v \sim_H^H v_2$. Then $\Phi_H(v : A) = \Phi_H(v_1 : A_1) + \Phi_H(v_2 : A_2) + n \cdot |dom(reach_H(v))|$*

Lemma 10 (Linearity of $\mathcal{E}_{\mathsf{copy}}$)**.** *Let $\mathcal{C}$ be a configuration, $\mathcal{C} \vdash^{\mathcal{E}_{\mathsf{copy}}} e \Downarrow v, H', F'$, and $\Sigma; \Gamma \vdash e : B$. Given that $\mathsf{linearComp}(\mathcal{C}, e)$, we have that $\mathsf{set}(reach_{H'}(v))$ and $\mathsf{disjoint}(\{R, F', reach_{H'}(v)\})$.*

Theorem 11 (Soundness)**.** *let $H_o \vDash V_o : \Gamma$, $\Sigma; \Gamma \left|\frac{q}{q'}\right. e : B$, $V_o, H_o \ \vdash e \Downarrow v_o, H'_o$. Then $\forall C \in \mathbb{Q}^+$ and configuration (V, H, R, F) s.t.*

1. $V_o, H_o \sim V, H$
2. $\mathsf{linearComp}(V, H, R, F, e)$
3. $|F| \geq \Phi_{V,H}(\Gamma) + q + C$

then there exists a triple (v, H', F'), and a freelist F' s.t.

1. $V, H, R, F \vdash^{\mathcal{E}_{\mathsf{copy}}} e \Downarrow v, H', F'$
2. $v_o \sim_{H'}^{H'_o} v$
3. $|F'| \geq \Phi_{H'}(v : B) + q' + C$

In other words, given a terminating expression (verified by succeeding with the run using $\mathcal{E}_{\mathsf{oper}}$) and given a freelist that is sufficiently large (as predicated by the type derivation), a run with $\mathcal{E}_{\mathsf{copy}}$will normalize to an equivalent value, and the resulting freelist will be sufficiently large (as predicated by the type derivation).

6 Implementation and Evaluation

Implementation We have implemented the novel cost semantics and the type system in Resource Aware ML (RaML). The implementation covers full RaML, including user-defined data types, higher-order functions, and polynomial potential functions. However, there is no destructive match for function closures and analyzing the heap-space usage of closures still amounts to counting allocations only. The main changes that where necessary have been in the rules for sharing and pattern matching as described earlier. We also needed to change some elaboration passes that were no longer cost preserving with the GC cost model.

The garbage collection cost semantics is implemented as an alternative evaluation module inside RaML. As mentioned before, RaML leverages the syntax of OCaml programs. First, we take the OCaml type checked abstract syntax tree and perform a series of transformations. The evaluation modules operate on the resulting RaML syntax tree. In the gc evaluation module, `evaluate` has the following signature:

```
evaluate : ('a, unit) Expressions.expression -> int -> (('a value * 'a heap * Int.Set.t) option)
```

Here, the second argument `int` specifies the size of the initial freelist. The result is an option triple of the return value, heap, and freelist; `None` is returned in case the freelist was not sufficient for the evaluation. Whereas the normal evaluation boxes every value (everything evaluates to a location), the gc module follows the cost semantics and only boxes data constructors. The rationale is that the size for other values can be computed statically and thus stack allocated. One difference between the cost semantics and its implementation is that while in the language presented here list is the only data type, our implementation supports user defined data types. The extension is straightforward except the treatment of the nil constructor, or generally "empty" constructors that have arity zero. For simplicity of presentation, we evaluate all nil constructors to the same null value in the cost semantics. This is natural for lists because all nil constructors are the same, and every list has at most one nil node. However, for custom data types that have more than one kind of empty constructor, it is not possible to map every constructor to the same null value. Thus, the implementation treats all constructors uniformly, so each empty constructor also costs one heap location.

As mentioned before, all functions used in a program are declared in a global mutually recursive block, and we do not account for the constant space overhead for this block in the cost semantics. In order to implement this global function block, we allow closure creation during program evaluation. However, we allocate all closures from a separate freelist into a separate heap. This ensures that data constructors are allocated from the correct freelist and no space overhead is created by allocating closures for function declarations.

Evaluation We evaluated our new analysis on a number of functions. Table 1 contains a representative compilation. It shows the type signature for each function. Table 2 presents the test data that showcase the difference between RaML$^{\mathsf{heap}}$, the previous RaML type system instantiated with the heap metric (the old analysis which only counts heap allocations), and FO^{gc}, which includes deallocations and copying cost for sharing. For each type system, we show the heap space bound computed by RaML, the number of constraints generated, and the time elapsed during analysis. The last column gives the expression for the exact heap high watermark derived by hand and verified by running the cost semantics.

function	type
quicksort	['a ->'a ->bool; 'a list] ->'a list
mergesort	[['a; 'a] ->bool; 'a list] ->'a list
ocamlsort	[['a; 'a] ->bool; 'a list] ->'a list
selection sort	int list ->int list
eratosthenes	int list ->int list
dfs	[btree; int] ->btree option
bfs	[btree; int] ->btree option
transpose	'a list list ->'a list list
map_it	['a ->'b; 'a list list] ->'b list list * 'b list list
pairs	'a list ->('a * 'a) list

Table 1: Signature of Test Functions

Except for bfs and dfs, all functions in the table take a *principal* argument of type list. The variables in the table refer to this argument (for example, the type of the principal argument of quicksort is 'a list). In general, M refers to the number of cons constructors of the principal argument (or the number of *outer* cons nodes in case of nested lists); L refers to the maximum number of cons nodes of the inner lists.

For the sorting functions, aside from mergesort, the new analysis using the gc metric derived asymptotically better bounds when compared to the heap metric. Furthermore, all bounds are *exact* with respect to the cost semantics. In regards to mergesort, the analysis was not able to derive a tight bound due to the limitations of AARA in deriving logarithmic bounds. A particularly nice result is that for quicksort, we derive that the space usage is exactly 0, which justifies its use as a zero space-overhead (or "in place") sorting algorithm.

Next, we have have the graph search algorithms operating on a binary tree. Again, the gc metric was able to derive exact space overheads, while the heap metric derived linear bounds for both. For transpose, the gc metric derived an asymptotically better bound, but was not able to derive the exact overhead. We implement matrices as lists-of-lists in row-major order. The transpose function is implemented tail-recursively, with the accumulator starting as the empty list. When "flipping" the first row r of the input and appending this to the accumulator, we need to create $|r|$ many new nil and cons constructors to store the row as a column. While this overhead only occurs once, RaML is unable to infer this from the source code, and thus the cost is repeated over the entire input matrix, resulting in the linear bound (w.r.t. the size of the matrix). This artifact is unrelated to the new extension; it is a limitation due to the implementation of RaML.

The last two functions demonstrate how the gc metric performs when there is variable sharing. map_it maps the input function across each list in the principal argument twice, returning a tuple of nested lists. The gc metric dictates that every outer data constructor in the principal argument needs to be copied, and thus gives the linear bound M + 1. In this case, the bound is exact. The function "pairs" takes a list and outputs all pairs of the input list which are ordered ascending in input position. For example, pairs [1;2;3;4] = [(1,2);(1,3);(1,4);(2,3);(2.4);(3,4)]. For pairs, the gc metric derived a bound that is asymptotically the same as the heap metric, but with better constants. An exact bound could not be derived because the deallocation potential from the pattern match in the definition of pairs is wasted since the matched body could already be typed with zero cost. However, this deallocation is used as usual in the cost semantics. Thus the slack in the bound totals to the size of the input.

	RaML$^{\textsf{heap}}$			FO^{gc}			
function	computed bound	constraints	time	computed bound	constraints	time	optimal
quicksort	$1.00 + 3.50M + 1.50M^2$	8515	0.52	0	8519	0.48	0
mergesort	$1.00 - 4.67M + 6.33M^2$	9572	0.64	$-0.50M + 0.50M^2$	9578	0.58	$\lfloor \log(M) \rfloor$
ocamlsort	$7.50 + 5.50M + 1.00M^2$	8565	0.51	$1.00 + 1.00M$	8573	0.50	$M + 1$
selection sort	$2.00 + 3.00M + 1.00M^2$	639	0.06	0	642	0.05	0
eratosthenes	$1.00 + 1.50M + 0.50M^2$	515	0.06	0	517	0.04	0
dfs	$3.00 + 2.00M$	5481	0.90	2	5483	0.36	2
bfs	$5.00 + 10.00M$	24737	4.15	4	24742	1.62	4
transpose	$1.00 + 3.50LM + 0.50LM^2$	10680	0.50	$1.00 + 2.00LM$	10684	0.50	$\max(0, 2L - 1)$
map_it	$2.00 + 2.00LM + 4.00M$	30699	1.58	$1.00M + 1.00$	30703	1.57	M + 1
pairs	$1.00 + 1.00M^2$	10214	0.60	$0.50M + 0.50M^2$	10217	0.64	$0.5M^2 - 1.5M + 2$

Table 2: Automatic Bound Analysis with RaML

7 Conclusion and Future Work

In this article, we introduced a novel operational cost semantics that models a perfect tracing garbage collector and an extension to AARA that is sound with respect to the new semantics. We implemented the new semantics and analysis as modules in RaML and found through experimental testing that the extended AARA was able to derive asymptotically better bounds for several commonly used functions and programming patterns; often, the bounds are optimal with respect to the cost semantics.

One direction for future work is using the *cost free* metric `cf` to model global garbage collection. In `cf`, all resource constants, including constructor nodes, are set to 0. A cost-free typing judgment then captures how an expression manipulates the structures in the context into the structure induced by its type. Using this fact, we could express the *maximum* space usage in the sequential composition $\texttt{let}(e_1; x : \tau.e_2)$ by analyzing e_1 twice—once with the cost-free metric and once with the regular metric—and assign potential to x using the result type in the cost-free typing. In prior work [23], the authors have successfully employed this cost-free metric to analyze parallel programs. Here, the difficulty is showing the simultaneous soundness of both destructive pattern matching and the cost-free composition. Another complication is the choice between local variable sharing and global context sharing. We leave the exploration of this area to future work.

Another direction for future work are function closures. The current treatment in our implementation is unsatisfactory since there is no equivalent to the destructive pattern match for closures. As a result, the GC metric in RaML only accounts for allocation of closures, which is not an improvement over the existing implementation. Ideally, we would like to account for deallocation at function applications and treat closures similar to other data structures in sharing. However, the size of closures cannot be determined easily statically and closures can not capture potential and are currently shared freely in RaML. As a result, the techniques we developed here do not directly carry over to closures.

Finally, we are interested in exploring if our work can be used to improve the efficiency of garbage collection in languages like OCaml. A guaranteed upper bound on the heap space can be used in different ways to control the frequency of the collections and the total memory that is requested from the operating system.

Acknowledgments

This article is based on research supported by the United States Air Force under DARPA AA Contract FA8750-18-C-0092 and DARPA STAC Contract FA8750-15-C-0082, and by the National Science Foundation under SaTC Award 1801369 and SHF Award 1812876. Yue Niu has been supported by Carnegie Mellon's Undergraduate Research Office through a Summer Undergraduate Research Fellowship (SURF). Any opinions, findings, and conclusions contained in this document are those of the authors and do not necessarily reflect the views of the sponsoring organizations.

References

[1] Elvira Albert, Jesús Correas Fernández, and Guillermo Román-Díez. Non-cumulative Resource Analysis. In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, (TACAS'15)*, 2015.

[2] Elvira Albert, Samir Genaim, and Miguel Gomez-Zamalloa. Heap space analysis for java bytecode. In *Proceedings of the 6th International Symposium on Memory Management (ISMM'07)*, 2007.

[3] Elvira Albert, Samir Genaim, and Miguel Gómez-Zamalloa Gil. Live Heap Space Analysis for Languages with Garbage Collection. In *Proceedings of the 2009 International Symposium on Memory Management (ISMM'09)*, 2009.

[4] Elvira Albert, Samir Genaim, and Miguel Gómez-Zamalloa. Heap space analysis for garbage collected languages. *Science of Computer Programming*, 78(9):1427 – 1448, 2013.

[5] Robert Atkey. Amortised Resource Analysis with Separation Logic. In *19th Euro. Symp. on Prog. (ESOP'10)*, 2010.

[6] Martin Avanzini and Georg Moser. A Combination Framework for Complexity. In *24th International Conference on Rewriting Techniques and Applications (RTA'13)*, 2013.

[7] Régis Blanc, Thomas A. Henzinger, Thibaud Hottelier, and Laura Kovács. ABC: Algebraic Bound Computation for Loops. In *Logic for Prog., AI., and Reasoning - 16th Int. Conf. (LPAR'10)*, 2010.

[8] Víctor A. Braberman, Federico Fernández, Diego Garbervetsky, and Sergio Yovine. Parametric prediction of heap memory requirements. In *7th Int. Symp. on Memory Management (ISMM'08)*, pages 141–150, 2008.

[9] Quentin Carbonneaux, Jan Hoffmann, Thomas Reps, and Zhong Shao. Automated Resource Analysis with Coq Proof Objects. In *29th International Conference on Computer-Aided Verification (CAV'17)*, 2017.

[10] Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. Compositional Certified Resource Bounds. In *36th Conference on Programming Language Design and Implementation (PLDI'15)*, 2015. Artifact submitted and approved.

[11] Arthur Charguéraud and François Pottier. Machine-Checked Verification of the Correctness and Amortized Complexity of an Efficient Union-Find Implementation. In *Interactive Theorem Proving - 6th International Conference (ITP'15)*, 2015.

[12] Wei-Ngan Chin, Huu Hai Nguyen, Corneliu Popeea, and Shengchao Qin. Analysing Memory Resource Bounds for Low-level Programs. In *Proceedings of the 7th International Symposium on Memory Management (ISMM'08)*, 2008.

[13] Wei-Ngan Chin, Huu Hai Nguyen, Shengchao Qin, and Martin Rinard. Memory Usage Verification for OO Programs. In *Proceedings of the 12th International Conference on Static Analysis (SAS'05)*, 2005.

[14] Norman Danner, Daniel R. Licata, and Ramyaa Ramyaa. Denotational Cost Semantics for Functional Languages with Inductive Types. In *29th Int. Conf. on Functional Programming (ICFP'15)*, 2012.

[15] Ankush Das, Jan Hoffmann, and Frank Pfenning. Parallel complexity analysis with temporal session types. In *23rd International Conference on Functional Programming (ICFP'18)*, 2018. Conditionally accepted.

[16] Antonio Flores-Montoya and Reiner Hähnle. Resource Analysis of Complex Programs with Cost Equations. In *Programming Languages and Systems - 12th Asian Symposiu (APLAS'14)*, 2014.

[17] Florian Frohn, M. Naaf, Jera Hensel, Marc Brockschmidt, and Jürgen Giesl. Lower Runtime Bounds for Integer Programs. In *Automated Reasoning - 8th International Joint Conference (IJCAR'16)*, 2016.

[18] Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *36th ACM Symp. on Principles of Prog. Langs. (POPL'09)*, 2009.

[19] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2016.

[20] Jan Hoffmann. *Types with Potential: Polynomial Resource Bounds via Automatic Amortized Analysis*. PhD thesis, Ludwig-Maximilians-Universität München, 2011.

[21] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate Amortized Resource Analysis. In *38th Symposium on Principles of Programming Languages (POPL'11)*, 2011.

[22] Jan Hoffmann, Ankush Das, and Shu-Chun Weng. Towards Automatic Resource Bound Analysis for OCaml. In *44th Symposium on Principles of Programming Languages (POPL'17)*, 2017.

[23] Jan Hoffmann and Zhong Shao. Automatic static cost analysis for parallel programs. In *Proceedings of the 24th European Symposium on Programming on Programming Languages and Systems - Volume 9032*, pages 132–157, New York, NY, USA, 2015. Springer-Verlag New York, Inc.

[24] Martin Hofmann and Steffen Jost. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *30th ACM Symp. on Principles of Prog. Langs. (POPL'03)*, 2003.

[25] Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, pages 185–197, New York, NY, USA, 2003. ACM.

[26] Martin Hofmann and Steffen Jost. Type-Based Amortised Heap-Space Analysis. In *15th Euro. Symp. on Prog. (ESOP'06)*, 2006.

[27] Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. Static Determination of Quantitative Resource Usage for Higher-Order Programs. In *37th ACM Symp. on Principles of Prog. Langs. (POPL'10)*, 2010.

[28] Zachary Kincaid, Jason Breck, Ashkan Forouhi Boroujeni, and Thomas Reps. Compositional recurrence analysis revisited. In *Conference on Programming Language Design and Implementation (PLDI'17)*, 2017.

[29] Ugo Dal Lago and Marco Gaboardi. Linear Dependent Types and Relative Completeness. In *26th IEEE Symp. on Logic in Computer Science (LICS'11)*, 2011.

[30] Yasuhiko Minamide. Space-profiling semantics of the call-by-value lambda calculus and the CPS transformation. *Electr. Notes Theor. Comput. Sci.*, 26:105–120, 1999.

[31] Greg Morrisett, Matthias Felleisen, and Robert Harper. Abstract models of memory management. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture (FPCA'95)*, 1995.

[32] Van Chan Ngo, Mario Dehesa-Azuara, Matthew Fredrikson, and Jan Hoffmann. Verifying and Synthesizing Constant-Resource Implementations with Types. In *38th IEEE Symposium on Security and Privacy (S&P '17)*, 2017.

[33] Tobias Nipkow. Amortized Complexity Verified. In *Interactive Theorem Proving - 6th International Conference (ITP'15)*, 2015.

[34] Lars Noschinski, Fabian Emmes, and Jürgen Giesl. Analyzing Innermost Runtime Complexity of Term Rewriting by Dependency Pairs. *J. Autom. Reasoning*, 51(1):27–56, 2013.

[35] Ivan Radiček, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Florian Zuleger. Monadic Refinements for Relational Cost Analysis. *Proc. ACM Program. Lang.*, 2(POPL), 2017.

[36] Hugo R. Simões, Pedro B. Vasconcelos, Mário Florido, Steffen Jost, and Kevin Hammond. Automatic Amortised Analysis of Dynamic Memory Allocation for Lazy Functional Programs. In *17th Int. Conf. on Funct. Prog. (ICFP'12)*, 2012.

[37] Moritz Sinn, Florian Zuleger, and Helmut Veith. A Simple and Scalable Approach to Bound Analysis and Amortized Complexity Analysis. In *Computer Aided Verification - 26th Int. Conf. (CAV'14)*, 2014.

[38] Daniel Spoonhower, Guy E. Blelloch, Robert Harper, and Phillip B. Gibbons. Space profiling for parallel functional programs. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 253–264, New York, NY, USA, 2008. ACM.

[39] Leena Unnikrishnan and Scott D. Stoller. Parametric Heap Usage Analysis for Functional Programs. In *Proceedings of the 2009 International Symposium on Memory Management (ISMM '09)*, 2009.

[40] Leena Unnikrishnan, Scott D. Stoller, and Yanhong A. Liu. Optimized Live Heap Bound Analysis. In *Verification, Model Checking, and Abstract Interpretation, 4th International Conference (VMCAI'03)*, pages 70–85, 2003.

[41] Pedro B. Vasconcelos, Steffen Jost, Mário Florido, and Kevin Hammond. Type-Based Allocation Analysis for Co-recursion in Lazy Functional Languages. In *24th European Symposium on Programming (ESOP'15)*, 2015.

[42] P. Wang, D. Wang, and A. Chlipala. TiML: A Functional Language for Practical Complexity Analysis with Invariants. In *Object-Oriented Prog., Syst., Lang., and Applications (OOPSLA'17)*, 2017.

[43] Ezgi Çiçek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. Relational Cost Analysis. In *44th Symposium on Principles of Programming Languages (POPL'17)*, 2017.

A Notation

For a finite mapping $f : A \to B$, we write dom for the defined values of f. Sometimes we shorten $x \in dom(f)$ to $x \in f$. We write $f[x \mapsto y]$ for the extension of f where x is mapped to y, with the constraint that $x \notin dom(f)$.

Given possibly non-disjoint sets A, B, let the disjoint union be $A \oplus B$ defined by $\{(\mathsf{inl}, a) \mid a \in A\} \cup \{(\mathsf{inr}, b) \mid b \in B\}$.

Let a multiset be a function $S : A \to \mathbb{N}$, i.e. a map of the multiplicity of each element in the domain. Write $x \in S$ iff $S(x) \geq 1$. If for all $s \in S$, $\mu(s) = 1$, then S is a property set, and we denote this by $\mathsf{set}(S)$. Addtionally, $A \uplus B$ denotes counting union of sets where $(A \uplus B)(s) = A(s) + B(s)$, similarly, $(A \cap B)(s) = \min A(s), B(s)$. Furthermore, $A \cup B$ denotes the usual union where $(A \cup B)(s) = \max(A(s), B(s))$. For the union of disjoint multi-sets A and B, we write $A \sqcup B$ to emphasize the disjointness. For a collection of pairwise disjoint multi-sets $\mathcal{C}$, i.e. $\forall X, Y \in \mathcal{C}.\ X \cap Y = \emptyset$, we write $\mathsf{disjoint}(\mathcal{C})$.

In the rest of the paper, we sometimes treat a set A sets as multiset $A : A \to \mathbb{N}$ via $x \mapsto \begin{cases} 1 & \text{if } x \in A \\ 0 & \text{o.w.} \end{cases}$ when convenient. For instance, if an operation defined on multisets is used on sets and multisets, the set is thus promoted.

Given a set A, let $\mathcal{P}(A)$ be the powerset of A. Given a multiset A, let $\wp(A)$ be the power multiset of A, i.e. the set of all submultisets of A.

For a partition $f : A \to \mathcal{P}(B)$, we write the set of equivalence classes as $ec(f) = \{f(x) \mid x \in A\} = f(A)$, i.e. the image of f on its domain A. Furthermore, a partition is *proper* if for any $x \in A$, $f(x) \neq \emptyset$.

Given a proper partition $f : A \to \mathcal{P}(B)$, for every $a \in A$, we can choose an arbitrary $b \in f(a)$ to be the representative for that part; call this $rep(a)$.

B Linearity of Copy Semantics

In the soundness proof of FO^{gc}, we used an important lemma: that $\mathcal{E}_{\mathsf{copy}}$is semantically linear, i.e. locations are used linearly. To see why, consider the second premise in the rule L:MatLD. In addition to the p units of potential justified by the definition of linear potential, we get 1 unit from deallocating the cons cell itself. This is only sound if in the corresponding rule in $\mathcal{E}_{\mathsf{copy}}$a location was actually collected. Consider the evaluation in question:

$$\frac{\begin{array}{c} V(x) = l \qquad H(l) = \langle v_h, v_t \rangle \qquad V'' = (V[x_h \mapsto v_h, x_t \mapsto v_t]) \upharpoonright_{FV(e_2)} \\ g = \{l \in H \mid l \notin F \cup R \cup locs_{V'',H}(e_2)\} \qquad V'', H, R, F \cup g \ \vdash e_2 \Downarrow v, H', F' \end{array}}{V, H, R, F \ \vdash \mathtt{match}\, x\, \{\mathtt{nil} \hookrightarrow e_1 \mid \mathsf{cons}(x_h; x_t) \hookrightarrow e_2\} \Downarrow v, H', F'}\,(\mathrm{S}_1)$$

If all the variables in V was mapped to values with disjoint reachable sets, then we see that l is only in the reachable set of x (assuming that well-typed expressions don't have duplicate occurences of variables, i.e. $x \notin FV(e_1) \cup FV(e_2)$. Then it follows that $l \in g$ given that locations in V, R, and F are also all disjoint, and this is what we needed to justify the rule L:MatL. Thus we have to show that $\mathcal{E}_{\mathsf{copy}}$preserves the linearity invariant: given a *linear* computation, the evaluation result is also linear.

First, we characterize semantically linear contexts:

Definition 9. (Linear context) Given a context (V, H), let $x, y \in dom(V)$, $x \neq y$, and $r_x = reach_H(V(x))$, $r_y = reach_H(V(y))$. It is *linear* given that:

1. $\mathsf{set}(r_x), \mathsf{set}(r_y)$
2. $r_x \cap r_y = \emptyset$

Denote this by $\mathsf{linearCtxt}(V, H)$.

Whenever $\mathsf{linearCtxt}(V, H)$ holds, visually, one can think of the stack as a collection of disjoint, directed trees with locations as nodes; consequently, there is at most one path from a variable on the stack V to any location in H. Now we can formalize our intuition for linear computations:

Definition 10 (Linear computation)**.** Given a configuration $\mathcal{C} = (V, H, R, F)$ and an expression e, we say the 5-tuple $(\mathcal{C}, e)$ is a *computation*; it is a *linear computation* given the following:

1. $dom(V) = FV(e)$
2. $\mathsf{linearCtxt}(V, H)$
3. $\mathsf{disjoint}(\{R, F, locs_{V,H}(e)\})$

And we write $\mathsf{linearComp}(V, H, R, F, e)$ to denote this fact.

Given a semantically linear computation (one with no sharing between the underlying locations), the resulting value is linear (expressed by item 1. and 2. below):

Lemma 12 (Linearity of $\mathcal{E}_{\mathsf{copy}}$)**.** *For all stacks V and heaps H, let $V, H, R, F \vdash e \Downarrow v, H', F'$ and $\Sigma; \Gamma \vdash e : B$. Then given that $\mathsf{linearComp}(V, H, R, F, e)$, we have the follwoing:*

1. $\mathsf{set}(reach_{H'}(v))$
2. $\mathsf{disjoint}(\{R, F', reach_{H'}(v)\})$, *and*
3. $\mathsf{stable}(R, H, H')$

Where stable is a predicate on $\mathcal{P}(\mathsf{Loc}) \times \mathsf{Heap} \times \mathsf{Heap}$, defined below. The premises of this lemma is a subset of the premises of the soundness theorem. Thus, we could have merged the proof of this lemma directly into the soundness proof. However, we think this makes the presentation clearer; furthermore, the linearity of $\mathcal{E}_{\mathsf{copy}}$is an interesting in itself, regardless of the accompanying type system. Some auxiliary lemmas:

Define $\dagger : L^p(A) \mapsto L(A)$ as the map that erases resource annotations. This gives a simplified jugdment $\boxed{\Sigma^\dagger; \Gamma^\dagger \vdash e : B^\dagger}$ used in proofs where the resource annotations are not necessary.

Lemma 13. *If $\Sigma; \Gamma \,\Big|\!\frac{q}{q'}\, e : B$, then $\Sigma^\dagger; \Gamma^\dagger \vdash e : B^\dagger$.*

Proof. Induction on the typing judgement. □

Define $FV^\star : \mathsf{Exp} \to \wp(\mathsf{Var})$, the multiset of free variables of expressions, as the usual FV inductively over the structure of e. This version of FV reflects the multiplicity of variable occurences.

Lemma 14. *If $\Sigma; \Gamma \,\Big|\!\frac{q}{q'}\, e : B$, then $\mathsf{set}(FV^\star(e))$ and $dom(\Gamma) = FV(e)$.*

Proof. Induction on the typing judgement. □

Definition 11 (Stability)**.** Given heaps H, H', a set of locations is *stable* if $\forall l \in R.\ H(l) = H'(l)$. Denote this by $\mathsf{stable}(R, H, H')$.

Lemma 15. *Let $H \vDash v \mapsto a : A$. For all sets of locations R, if $reach_H(v) \subseteq R$ and $\mathsf{stable}(R, H, H')$, then $H' \vDash v \mapsto a : A$ and $reach_H(v) = reach_{H'}(v)$.*

Proof. Induction on the structure of $H \vDash v \mapsto a : A$. □

Corollary 16. *Let $H \vDash V : \Gamma$. For all sets of locations R, if $\bigcup_{x \in V} reach_H(V(x)) \subseteq R$ and $\mathsf{stable}(R, H, H')$, then $H' \vDash V : \Gamma$.*

Proof. Follows from Lemma 6. □

Lemma 17 (stability of copying)**.** *Let $H', v' = copy(H, L, v)$. For all $l \in H$, if $l \notin L$, then $H(l) = H'(l)$. Further, $reach_{H'}(v') \subseteq L$.*

Lemma 18 (copy is copy)**.** *Let $H', v' = copy(H, L, v)$. If $H \vDash v \mapsto a : A$, then $H' \vDash v' \mapsto a : A$.*

Lemma 19. *Let $A \curlyvee^n A_1, A_2$, $H \vDash v_1 : A_1$, $H \vDash v_2 : A_2$, and $H \vDash v : A$. Then $\Phi_H(v : A) = \Phi_H(v_1 : A_1) + \Phi_H(v_2 : A_2) + n \cdot |dom(reach_H(v))|$*

Lemma 20. *Let $H \vDash V : \Gamma$, $\Sigma; \Gamma \vdash e : A$, and $V, H \vdash e \Downarrow v, H'$. Then $H' \vDash v : A$.*

Now the proof for linearity of $\mathcal{E}_{\mathsf{copy}}$:

Proof. Nested induction on the evaluation judgement and the typing judgement.

Case 1: E:Var

Suppose $H \vDash V : \Gamma, dom(V) = FV(e), \mathsf{linearCtxt}(V, H), \mathsf{disjoint}(\{R, F, locs_{V,H}(e)\})$
$\mathsf{set}(reach_H(v))$ $\qquad (\mathsf{linearCtxt}(V, H))$
$\mathsf{disjoint}(\{R, F, reach_H(v)\})$ $\qquad (\mathsf{disjoint}(\{R, F, locs_{V,H}(e)\}))$
$\mathsf{linearCtxt}(V, H)$ $\qquad$ (Sp.)
$\mathsf{stable}(R, H, H')$ $\qquad (H = H')$

Case 2: E:Const* Due to similarity, we show only for E:ConstI

Suppose $H \vDash V : \Gamma, dom(V) = FV(e), \mathsf{linearCtxt}(V, H), \mathsf{disjoint}(\{R, F, locs_{V,H}(e)\})$
$\mathsf{set}(reaach_H(v))$ $\qquad (reach_H(v) = \emptyset)$
$\mathsf{disjoint}(\{R, F, \emptyset\})$ $\qquad (\mathsf{disjoint}(R, F))$
$\mathsf{linearCtxt}(V, H)$ $\qquad$ (Sp.)
$\mathsf{stable}(R, H, H')$ $\qquad (H = H')$

Case 4: E:App

Case 5: E:CondT Similar to E:MatNil

Case 6: E:CondF Similar to E:CondT

Case 7: E:Let

$V, H, R, F \ \vdash \mathtt{let}(e_1; x : \tau.e_2) \Downarrow v_2, H_2, F_2$ $\qquad$ (case)
$V_1, H, R', F \vdash e_1 \Downarrow v_1, H_1, F_1$ $\qquad$ (ad.)
$\Sigma; \Gamma_1, \Gamma_2 \vdash \mathtt{let}(e_1; x : \tau.e_2) : B$ $\qquad$ (case)
$\Sigma; \Gamma_1 \vdash e_1 : A$ $\qquad$ (ad.)
Suppose $H \vDash V : \Gamma, dom(V) = FV(e), \mathsf{linearCtxt}(V, H), \mathsf{disjoint}(\{R, F, locs_{V,H}(e)\})$
$H \vDash V_1 : \Gamma_1$ $\qquad$ (def of W.D.E and Lemma 14)
By IH, we have invariant on the first premise
NTS (1) - (3) to instantiate invariant on the first premise
(1) $\quad dom(V_1) = FV(e_1)$ $\qquad$ (def of V_1)
(2) $\quad \mathsf{linearCtxt}(V_1, H)$ $\qquad$ ($\mathsf{linearCtxt}(V, H)$ and $V_1 \subseteq V$)
(3) $\quad \mathsf{disjoint}(R', F, locs_{V,H}(e_1))$
$F \cap R' = \emptyset$ $\qquad$ ($F \cap locs_{V,H}(e) = \emptyset$ and $locs_{V_2,H}(\mathtt{lam}(x : \tau.e_2)) \subseteq locs_{V,H}(e)$)
$FV(e_1) \cap FV(\mathtt{lam}(x : \tau.e_2)) = \emptyset$ $\qquad$ (Lemma 14)
$locs_{V,H}(e_1) \cap locs_{V_2,H}(\mathtt{lam}(x : \tau.e_2)) = \emptyset$ $\qquad (\mathsf{linearCtxt}(V, H))$
$R' \cap locs_{V,H}(e_1) = \emptyset$ $\qquad (\mathsf{disjoint}(\{R, locs_{V,H}(e)\}))$
$F \cap locs_{V,H}(e_1) = \emptyset$ $\qquad$ (Sp.)
Thus we have $\mathsf{disjoint}(R', F, locs_{V,H}(e_1))$

By IH,

1.(1) $\mathsf{set}(reach_{H_1}(v_1))$

1.(2) $\mathsf{disjoint}(\{R', F_1, reach_{H_1}(v_1)\})$

1.(3) $\mathsf{stable}(R', H, H_1)$

$V_2', H_1, R, F_1 \cup g \vdash e_2 \Downarrow v_2, H_2, F_2$ (ad.)

$\Sigma; \Gamma_2, x : A \vdash e_2 : B$ (ad.)

$H_1 \vDash V_2' : (\Gamma_2, x : A)$ (Lemma **??**)

By IH, we have invariant on the second premise

NTS (1) - (3) to instantiate invariant on the second premise

(1) $dom(V_2') = FV(e_2)$ (def of V_2')

(2) $\mathsf{linearCtxt}(V_2', H_1)$

Let $x_1, x_2 \in V2', x_1 \neq x_2$ be arb.

case: $x_1 \neq x, x_2 \neq x$

$reach_H(V_2'(x_1)) \subseteq R'$ $(reach_H(V_2'(x_1)) \subseteq locs_{V_2',H}(\mathtt{lam}(x : \tau.e_2)))$

$reach_H(V_2'(x_2)) \subseteq R'$ $(reach_H(V_2'(x_2)) \subseteq locs_{V_2',H}(\mathtt{lam}(x : \tau.e_2)))$

$reach_H(V_2'(x_1)) = reach_{H_1}(V_2'(x_1)), reach_H(V_2'(x_2)) = reach_{H_1}(V_2'(x_2))$
($\mathsf{stable}(R', H, H_1)$ and Lemma 6)

$reach_{H_1}(V_2'(x_1)) = reach_H(V(x_1)), reach_{H_1}(V_2'(x_2)) = reach_H(V(x_2))$
($\mathsf{stable}(R', H, H_1)$ and Lemma 6)

$\mathsf{linearCtxt}(V_2', H_1)$ ($\mathsf{linearCtxt}(V, H)$)

case: $x_1 = x, x_2 \neq x$

$reach_{H_1}(V_2'(x_1)) = reach_{H_1}(v_1)$ (def of V_2')

$reach_{H_1}(V_2'(x_2)) \subseteq R'$ (same as above)

$\mathsf{set}(reach_{H_1}(v_1))$ (IH 1.1)

$reach_{H_1}(V_2'(x_2)) = reach_H(V(x_2))$ (same as above)

$\mathsf{set}(reach_{H_1}(V_2'(x_2)))$ ($\mathsf{linearCtxt}(V, H)$)

$reach_{H_1}(V_2'(x_1)) \cap reach_{H_1}(V_2'(x_2)) = \emptyset$ ($\mathsf{disjoint}(\{R', reach_{H_1}(v_1)\})$)

Thus we have $\mathsf{linearCtxt}(V_2', H_1)$

(3) $\mathsf{disjoint}(\{R, F_1 \cup g, locs_{V_2',H_1}(e_2)\})$

$R \cap F_1 = \emptyset$ ($\mathsf{disjoint}(\{R', F_1\})$ from 1.2 and $R \subseteq R'$)

$R \cap (F_1 \cup g) = \emptyset$ (def of g)

NTS $(F_1 \cup g) \cap locs_{V_2',H_1}(e_2) = \emptyset$

Let $l \in locs_{V_2',H_1}(e_2)$ be arb.

$l \in reach_{H_1}(V_2'(x'))$ for some $x' \in V_2'$

case: $x' \neq x$

$reach_H(V_2(x')) = reach_{H_1}(V_2'(x'))$ (same as above)

$reach_{H_1}(V_2'(x')) \subseteq R'$ (def of R')

$reach_{H_1}(V_2'(x')) \cap F_1 = \emptyset$ ($\mathsf{disjoint}(\{R', F_1\})$ from 1.2)

case: $x' = x$

$reach_{H_1}(V_2'(x')) = reach_{H_1}(v_1)$ (def of V_2')

$reach_{H_1}(V_2'(x')) \cap F_1 = \emptyset$ ($\mathsf{disjoint}(\{F_1, reach_{H_1}(v_1)\})$ from 1.2)

$reach_{H_1}(V_2'(x')) \subseteq locs_{V_2',H_1}(e_2)$ (def of $locs_{V,H}$)

$reach_{H_1}(V_2'(x')) \cap g = \emptyset$ (def of g)

Thus $reach_{H_1}(V_2'(x')) \cap (F_1 \cup g) = \emptyset$

NTS $R \cap locs_{V_2',H_1}(e_2) = \emptyset$

Let $l \in locs_{V_2',H_1}(e_2)$ be arb.

$l \in reach_{H_1}(V_2'(x'))$ for some $x' \in V_2'$

case: $x' \neq x$

$reach_H(V_2(x')) = reach_{H_1}(V_2'(x'))$ (same as above)

$l \in locs_{V,H}(e)$ (def of $locs_{V,H}$)

$l \notin R$ ($\mathsf{disjoint}(\{R, locs_{V,H}(e)\})$ from 0.3)

case: $x' = x$

$reach_{H_1}(V_2'(x')) = reach_{H_1}(v_1)$ (def of V_2')

$reach_{H_1}(V_2'(x')) \cap R = \emptyset$ ($\mathsf{disjoint}(\{R', reach_{H_1}(v_1)\})$ from 1.2 and $R \subseteq R'$)

Thus $reach_{H_1}(V_2'(x')) \cap R = \emptyset$

Hence we have *(3)* $\mathsf{disjoint}(R, F_1 \cup g, locs_{V_2',H_1}(e_2))$

By instantiating the invariant on the second premise , we have

2.(1) $\mathsf{set}(reach_{H_2}(v_2))$

2.(2) $\mathsf{disjoint}(\{R, F_2, reach_{H_2}(v_2)\})$

2.(3) $\mathsf{stable}(R, H_1, H_2)$

Lastly, showing (1) - (3) holds for the original case :

(1) $\mathsf{set}(reach_{H_2}(v_2))$ (By 2.1)

(2) $\mathsf{disjoint}(\{R, F_2, reach_{H_2}(v_2)\})$ (By 2.2)

(3) $\mathsf{stable}(R, H_1, H_2)$

Let $l \in R$ be arb.

$H(l) = H_1(l)$ ($\mathsf{stable}(R', H, H_1)$ from 1.3)

$H_1(l) = H_2(l)$ ($\mathsf{stable}(R, H_1, H_2)$ from 2.3)

$H(l) = H_2(l)$

Hence $\mathsf{stable}(R, H, H_2)$

Case 8: E:Pair Similar to E:Var

Case 9: E:MatP Similar to E:MatCons

Case 10: E:Nil Similar to E:Const*

Case 11: E:Cons

$V, H, R, F \vdash e \Downarrow l, H'', F'$ (case)

Suppose $H \vDash V : \Gamma, dom(V) = FV(e), \mathsf{linearCtxt}(V, H), \mathsf{disjoint}(\{R, F, locs_{V,H}(e)\})$

NTS (1) - (3) holds after evaluation

(1) $\mathsf{set}(reach_{H''}(l))$

$\mathsf{stable}(\{locs_{V,H}(e)\}, H, H'')$ ($\mathsf{disjoint}(\{F, locs_{V,H}(e)\})$ and *copy* only updates $l \in L \subseteq F$)

$reach_H(V(x_i)) = reach_{H''}(V(x_i))$ ($reach_H(V(x_i)) \subseteq locs_{V,H}(e)$ and 6 for $i = 1, 2$)

$reach_{H''}(l) = \{l\} \cup reach_{H''}(V(x_1)) \cup reach_{H''}(V(x_2))$ (def of $reach_H$)

$\mathsf{set}(reach_{H''}(l))$ ($l \notin locs_{V,H}(e)$ and $\mathsf{linearCtxt}(V, H)$)

(2) $\mathsf{disjoint}(\{R, F', reach_{H''}(l)\})$

$R \cap F' = \emptyset$ ($F' \subseteq F$ and $\mathsf{disjoint}(\{R, F\})$)

$R \cap reach_{H''}(l) = \emptyset$ ($l \in F$ and $\mathsf{disjoint}(\{R, locs_{V,H}(e)\})$)

$F' \cap reach_{H''}(l) = \emptyset$ ($F' \subseteq F$ and $\mathsf{disjoint}(\{F, locs_{V,H}(e)\})$)

Thus we have *(2)* $\mathsf{disjoint}(\{R, F', reach_{H''}(l)\})$

(3) $\mathsf{stable}(R, H, H'')$ (since copy only updates $l \in L \subseteq F$ and $F \cap R = \emptyset$)

Case 12: E:MatNil

Suppose $H \vDash V : \Gamma, dom(V) = FV(e), \mathsf{linearCtxt}(V, H), \mathsf{disjoint}(\{R, F, locs_{V,H}(e)\})$

$\Sigma; \Gamma' \vdash e_1 : B$ (ad.)

$V', H, R, F \cup g \ \vdash e_1 \Downarrow v, H', F'$ (ad.)

$H \vDash V' : \Gamma'$ (def of W.D.E)

By IH, we have invariant on the premise

NTS (1) - (3) to instantiate invariant on the premise

(1) $dom(V') = FV(e_1)$ (def of V')

(2) $\mathsf{linearCtxt}(V', H)$ ($\mathsf{linearCtxt}(V, H)$ and $V' \subseteq V$)

(3) $\mathsf{disjoint}(\{R, F, locs_{V',H}(e_1)\})$ ($\mathsf{disjoint}(\{R, F, locs_{V,H}(e)\})$ and $locs_{V',H}(e_1) \subseteq locs_{V,H}(e)$)

Instantiating invariant on premise ,

(1) $\mathsf{set}(reach_{H'}(v))$

(2) $\mathsf{disjoint}(\{R, F_1, reach_{H'}(v)\})$

(3) $\mathsf{stable}(R, H, H')$

Case 13: E:MatCons

$V(x) = l$ (ad.)

$H(l) = \langle v_h, v_t \rangle$ (ad.)

$\Gamma = \Gamma', x : L(A)$ (ad.)

$\Sigma; \Gamma', x_h : A, x_t : L(A) \vdash e_2 : B$ (ad.)

$V'', H, R, F \cup g \ \vdash e_2 \Downarrow v_2, H_2, F'$ (ad.)

Suppose $H \vDash V : \Gamma, dom(V) = FV(e), \mathsf{linearCtxt}(V, H), \mathsf{disjoint}(\{F, R, locs_{V,H}(e)\})$

$H \vDash V(x) : L(A)$ (def of W.D.E)

$H \vDash v_h : A, \ H \vDash v_t : L(A)$

$H \vDash V'' : \Gamma', x_h : A, x_t : L(A)$ (def of W.D.E)

By IH, we have invariant on the premise

NTS (1) - (3) to instantiate invariant on the premise

(1) $dom(V'') = FV(e_2)$ (def of V'')

(2) $\mathsf{linearCtxt}(V'', H)$

Let $x_1, x_2 \in V'', x_1 \neq x_2, r_{x_1} = reach_H(V''(x_1)), r_{x_2} = reach_H(V''(x_2))$

case: $x_1 \notin \{x_h, x_t\}, x_2 \notin \{x_h, x_t\}$

(1), (2) from $\mathsf{linearCtxt}(V, H)$

case: $x_1 = x_h, x_2 \notin \{x_h, x_t\}$

$\mathsf{set}(r_{x_1})$ (since $\mathsf{set}(reach_H(V(x)))$ from $\mathsf{linearCtxt}(V, H)$)

$\mathsf{set}(r_{x_2})$ (since $\mathsf{linearCtxt}(V, H)$)

$x_2 \in FV(e)$ (def of FV)

$reach_H(V(x)) \cap r_{x_2} = \emptyset$ (def of $reach$ and $\mathsf{linearCtxt}(V, H)$)

hence $r_{x_1} \cap r_{x_2} = \emptyset$

case: $x_1 = x_h, x_2 = x_t$

$\mathsf{set}(r_{x_1})$ since $\mathsf{set}(reach_H(V(x)))$ from $\mathsf{linearCtxt}(V, H)$

$\mathsf{set}(r_{x_2})$ since $\mathsf{set}(reach_H(V(x)))$ from $\mathsf{linearCtxt}(V, H)$

$r_{x_1} \cap r_{x_2} = \emptyset$ ($\mathsf{set}(reach_H(V(x)))$)

case: otherwise

similar to the above

Thus we have $\mathsf{linearCtxt}(V'', H)$

(3) $\mathsf{disjoint}(\{R, F \cup g, locs_{V'',H}(e_2)\})$

$(F \cup g) \cap R = \emptyset$ (since $F \cap R = \emptyset$ and by def of g)

NTS $R \cap locs_{V'',H}(e_2) = \emptyset$

Let $l' \in locs_{V'',H}(e_2)$ be arb.

case: $l' \in reach_H(V''(x'))$ for some $x' \in FV(e_2)$ where $x' \notin \{x_h, x_t\}$

$x' \in V$ (def of V'')

$l' \in reach_H(V(x'))$

$x' \in FV(e)$ (def of FV)

$l' \in locs_{V,H}(e)$ (def of $locs_{V,H}$)

$l' \notin R$ ($\mathsf{disjoint}(\{R, F, locs_{V,H}(e)\})$)

case: $l' \in reach_H(V''(x_h))$

$l' \in reach_H(v_h)$

$l' \in reach_H(V(x))$ (def of $reach$)

$l' \in locs_{V,H}(e)$ (def of $locs_{V,H}$)

$l' \notin R$ (since $\mathsf{disjoint}(\{F, R, locs_{V,H}(e)\})$)

case: $l' \in reach_H(V''(x_t))$

similar to above

Hence $R \cap locs_{V'',H}(e_2) = \emptyset$

$F \cap locs_{V'',H}(e_2) = \emptyset$ (Similar to above)

$g \cap locs_{V'',H}(e_2) = \emptyset$ (def. of g)

$(F \cup g) \cap locs_{V'',H}(e_2) = \emptyset$

Thus $\mathsf{disjoint}(\{R, F \cup g, locs_{V'',H}(e_2)\})$

Instantiating invariant on the premise,

(1) $\mathsf{set}(reach_{H'}(v))$

(2) $\mathsf{disjoint}(\{R, F', reach_{H'}(v)\})$

(3) $\mathsf{stable}(R, H, H')$

Case 13: E:Share

$e = \texttt{share}\ x\ \texttt{as}\ x_1, x_2\ \texttt{in}\ e'$ (case)

Suppose $H \vDash V : \Gamma, dom(V) = FV(e), \mathsf{linearCtxt}(V, H), \mathsf{disjoint}(\{R, F, locs_{V,H}(e)\})$ (def. of wfc)

Let $V_2 = (V[x_1 \mapsto v', x_2 \mapsto v'']) \restriction_{FV(e')}$

We show the subsequent computation is also well-formed to invocate the IH:

(1) $dom(V_2) = FV(e')$ ($dom(V) = FV(e)$ and def of FV)

(2) $\mathsf{linearCtxt}((V[x_1 \mapsto v', x_2 \mapsto v'']) \restriction_{FV(e')}, H)$

Let $x' \mapsto v''' \in V'[x_1 \mapsto v']$.STS $reach_{H'}(v''') \cap reach_{H'}(v'') = \emptyset$

$reach_{H'}(v'') \subseteq L \subseteq F$ (definition of copy)

$reach_{H'}(v''') \subseteq locs_{V'[x_1 \mapsto v'],H'}(e') \subseteq locs_{V,H}(e)$ (By Lemma 7)

but since $F \cap locs_{V,H}(e) = \emptyset$, we have the result. ($\mathsf{linearComp}(V, H, R, F, e)$)

(3) $\mathsf{disjoint}(\{R, F \setminus L, locs_{V_2,H'}(e')\})$

Disjointedness involving F follows from assumption. Last one follows since $locs_{V_2,H'}(e') \subseteq locs_{V,H}(e) \cup L$

By IH:

(1) $\mathsf{set}(reach_{H''}(v))$

(2) $\mathsf{disjoint}(\{R, F', reach_{H''}(v)\})$

(3) $\mathsf{stable}(R, H', H'')$

STS $\mathsf{stable}(R, H, H')$, which follows from $L \cap R = \emptyset$ and Lemma 7

□

C Soundness

Theorem 21 (Soundness). *let $H_o \vDash V_o : \Gamma$, $\Sigma; \Gamma \left|\frac{q}{q'}\right. e : B$, $V_o, H_o \vdash e \Downarrow v_o, H'_o$. Then $\forall C \in \mathbb{Q}^+$ and configuration V, H, R, F s.t.*

1. $V_o, H_o \sim V, H$
2. $dom(V) = FV(e)$
3. $\mathsf{linearCtxt}(V, H)$
4. $\mathsf{disjoint}(\{R, F, locs_{V,H}(e)\})$
5. $|F| \geq \Phi_{V,H}(\Gamma) + q + C$

then there exists a triple (v, H', F'), and a freelist F' s.t.

1. $V, H, R, F \vdash e \Downarrow v, H', F'$
2. $v_o \sim^{H'_o}_{H'} v$
3. $|F'| \geq \Phi_{H'}(v : B) + q' + C$

Call this the existence clause.

Proof. Nested induction on the evaluation judgement and the typing judgement.

Case 1: E:Var

$V_o, H_o \vdash x \Downarrow V_o(x), H_o$ (case)

$\Sigma; x : B \left|\frac{q}{q}\right. x : B$ (case)

Let $C \in \mathbb{Q}^+$, well-formed configuration V, H, R, F s.t. $V_o, H_o \sim V, H$ and $|F| \geq \Phi_{V,H}(x : B) + q + C$

NTS the conclusions for the existence clause:

(1) $V, H, R, F \vdash e \Downarrow V(x), H, F$ (E:Var)

(2) $V_o(x) \sim_H^{H_o} V(x)$ (assumption)

(3) And we have $F \geq \Phi_{V,H}(x : B) + q + C$

$= \Phi_H(V(x) : B) + q + C$ (definition of Φ)

Case 2: E:Const* Similar to E:Var

Case 4: E:App

$V_o, H_o \vdash f(x) \Downarrow v_o, H_o'$ (case)

$V_o(x) = v_o'$ (admissibility)

Let $P(f) = (y_f, e_f)$

$[y_f \mapsto v_o'], H_o \vdash e_f \Downarrow v_o, H_o'$ (admissibility)

$\Sigma; x : A \left|\frac{q}{q'}\right. f(x) : B$ (case)

$\Sigma(f) = A \xrightarrow{q/q'} B$ (admissibility)

Let $C \in \mathbb{Q}^+$, well-formed configuration V, H, R, F s.t. $V_o, H_o \sim V, H$ and $|F| \geq \Phi_{V,H}(x : A) + q + C$

$\Sigma; y_f : A \left|\frac{q}{q'}\right. e_f : B$

Let $V(x) = v', g = \mathit{collect}(R, \mathit{locs}_{V,H}(e_f), H, F)$

By IH on $[y_f \mapsto v'], H, R, F \cup g$, have the existence clause. NTS the following conditions:

(1) $[y_f \mapsto v_o'], H_o \sim [y_f \mapsto v'], H$ ($v_o' \sim_H^{H_o} v'$ by assumption)

(2) - *(4)* Have by assumption

(5) NTS $|F \cup g| \geq \Phi_{[y_f \mapsto v'],H}(x : A) + q + C$

STS $|F| \geq \Phi_{[y_f \mapsto v'],H}(y_f : A) + q + C$

$|F| \geq \Phi_{V,H}(x : A) + q + C$

$= \Phi_H(v' : A) + q + C$

$= \Phi_{[y_f \mapsto v'],H}(y_f : A) + q + C$

Applying the existence clause and E:App, we're done.

Case 5: E:CondT

$\Gamma = \Gamma', x : \texttt{bool}$ (ad.)

$H \vDash V : \Gamma'$ (def of W.F.E)

$\Sigma; \Gamma' \left|\frac{q}{q'}\right. e_t : B$ (ad.)

$V, H, R, F \cup g \vdash e_t \Downarrow v, H', F'$ (ad.)

$|F \cup g| - |F'| \leq \Phi_{V,H}(\Gamma) + q - (\Phi_{H'}(v : B) + q')$ (IH)

$|F| - |F'| \leq \Phi_{V,H}(\Gamma) + q - (\Phi_{H'}(v : B) + q')$

Case 6: E:CondF Similar to E:CondT

Case 7: E:Let

$V_o, H_o \vdash e \Downarrow v_o'', H_o''$ (case)

$V_o' = V_o \restriction_{dom(FV(e_1)}$ (ad.)

$V_o', H_o \vdash e_1 \Downarrow v_o', H_o'$ (ad.)

$\Sigma; \Gamma \left|\frac{q}{q'}\right. \mathtt{let}(e_1; x : \tau.e_2) : B$ (case)

$\Gamma = \Gamma_1, \Gamma_2$ (ad.)

$\Sigma; \Gamma_1 \left|\frac{q}{p}\right. e_1 : A$ (ad.)

$H \vDash V_o' : \Gamma_1$ (def of $\vDash$)

Let $C \in \mathbb{Q}^+$, well-formed configuration V, H, R, F s.t. $V_o, H_o \sim V, H$ and $|F| \geq \Phi_{V,H}(\Gamma) + q + C$

NTF v_2, H_2, F_2 s.t.

1.$V, H, R, F \vdash e \Downarrow v_2, H_2, F_2$ and

2.$v_o'' \sim_{H_2}^{H_o''} v_2$

3.$|F_2| \geq \Phi_{H_2}(v_2 : B) + q' + C$

Let $R' = R \cup locs_{V,H}(FV(e_2) \setminus \{x\})$

$\mathsf{disjoint}(\{R', F, locs_{V,H}(e_1)\})$ (Similar to case in Lemma 10)

Let $V_1 = V \restriction_{dom(FV(e_1)}$

Instantiate IH with $C = C + \Phi_{V_2,H}(\Gamma_2), V = V_1, H = H, F = F, R = R'$

we get existence clause on the first premise

NTS (1) - (4) to instantiate existence clause on the first premise:

(1) $V_o', H_o \sim V_1, H$ (assumption)

(2) - (4) Same as in 10

(4) $|F| \geq \Phi_{V_1,H}(\Gamma_1) + q + C + \Phi_{V,H}(\Gamma_2)$

($|F| \geq \Phi_{V,H}(\Gamma) + q + C$ and $\Phi_{V,H}(\Gamma) \geq \Phi_{V_1,H}(\Gamma_1) + \Phi_{V,H}(\Gamma_2)$)

Instantiating existence clause on the first premise , we get v_1, H_1, F_1 s.t.

Fact 1. $V_1, H, R', F \vdash e_1 \Downarrow v_1, H_1, F_1$

Fact 2. $v_o' \sim_{H_1}^{H_o'} v_1$

Fact 3. $|F_1| \geq \Phi_{H_1}(v_1 : A) + p + C + \Phi_{V_2,H_1}(\Gamma_2)$

For the second premise:

$V_o'' = (V_o[x \mapsto v_o']) \restriction_{FV(e_2)}$ (ad.)

$V_o'', H_o' \vdash e_2 \Downarrow v_o'', H_o''$ (ad.)

$\Sigma; \Gamma_2, x : A \left|\frac{p}{q'}\right. e_2 : B$ (ad.)

$H_1 \vDash v_o' : A$ (By Lemma 6 and **??**)

$H_1 \vDash V_2 : \Gamma_2, x : A$ (def of $\vDash$)

Let $V_2 = (V[x \mapsto v_1]) \restriction_{FV(e_2)}$

Let $g = \{l \in H_1 \mid l \notin F_1 \cup R \cup locs_{V_2,H_1}(e_2)\}$

Instantiate IH with $C = C, V = V_2, H = H_1, F = F_1 \cup g, R = R,$ we get existence clause on the second premise

NTS (1) - (4) to instantiate existence clause on the second premise:

(1) $V_o'', H_o' \sim V_2, H_1$ (assumption and Fact 2.)

(2) - (4) Same as Lemma 10

(5) $|F_1 \cup g| \geq \Phi_{V_2,H_1}(\Gamma_2, x : A) + p + C$

STS $|F_1| \geq \Phi_{V_2,H_1}(\Gamma_2, x : A) + p + C$

Have $|F_1| \geq \Phi V_2, H_1(\Gamma_2) + \Phi_{H_1}(v_1 : A) + p + C$ (By Fact 3.)

Instantiate the existence clause on the second premise and apply E:Let and we're done

Case 8: E:Pair Similar to E:Const*

Case 9: E:MatP Similar to E:MatCons

Case 10: E:Nil Similar to E:Const*

Case 11: E:Cons

$V_o, H_o \vdash \texttt{cons}(x_1; x_2) \Downarrow l_o, H'_o$ (case)

$H'_o(l_o) = \langle V_o(x_1), V_o(x_2) \rangle$ (admissibility)

$\Sigma; x_1 : A, x_2 : L^p(A) \left|\frac{q+p+1}{q}\right. \texttt{cons}(x_1; x_2) : L^p(A)$

Let $C \in \mathbb{Q}^+$, well-formed configuration V, H, R, F s.t.

$V_o, H_o \sim V, H$ and $|F| \geq \Phi_{V,H}(x_1 : A, x_2 : L^p(A)) + (q + p + 1) + C$

Let $v = \langle V(x_1), V(x_2) \rangle, l \in F, H' = H\{l \mapsto v\}, F' = F \setminus \{l\}$

(1) $V, H, R, F \vdash \texttt{cons}(x_1; x_2) \Downarrow l, H', F'$ (E:Cons)

(2) $l_o \sim^{H'_o}_{H'} l$ (assumption)

(3 $|F'| \geq \Phi_{H'}(l : L^p(A)) + q + C$

$|F| \geq \Phi_{V,H}(x_1 : A, x_2 : L^p(A)) + (p + q + 1) + C$

$= \Phi_{H'}(l : L^p(A)) + q + 1 + C$

$|F'| = |F| - 1 \geq \Phi_{H'}(l : L^p(A)) + q + C$

Case 12: E:MatNil Similar to E:Cond*

Case 13: E:MatCons

$V_o(x) = l_o$ (ad.)

$H_o(l_o) = \langle v'_o, v''_o \rangle$ (ad.)

$\Gamma = \Gamma', x : L^p(A)$ (ad.)

$\Sigma; \Gamma', x_h : A, x_t : L^p(A) \left|\frac{q+p+1}{q'}\right. e_2 : B$ (ad.)

$V'_o = (V_o[x_h \mapsto v'_o, x_t \mapsto v''_o]) \restriction_{FV(e_2)}$ (ad.)

$V'_o, H_o \vdash e_2 \Downarrow v_o, H'$ (ad.)

$H \vDash V(x) : L^p(A)$ (def of $\vDash$)

$H \vDash v_h : A,\ H \vDash v_t : L^p(A)$ (Inversion on $\vDash$)

$H \vDash V'_o : \Gamma', x_h : A, x_t : L^p(A)$ (def of W.D.E)

Let $C \in \mathbb{Q}^+$, well-formed configuration V, H, R, F s.t. $V_o, H_o \sim V, H$ and $|F| \geq \Phi_{V,H}(\Gamma) + q + C$

NTF v, H', F' s.t.

1.$V, H, R, F \vdash e \Downarrow v, H', F'$ and

2.$v_o \sim^{H'_o}_{H'} v$

2.$|F'| \geq \Phi_{H'}(v : B) + q' + C$

Let $V(x) = l$

$H(l) = \langle v_h, v_t \rangle$ ($l_o \sim_H^{H_o} l$ by assumption)

Let $V' = (V[x_h \mapsto v_h, x_t \mapsto v_t]) \upharpoonright_{FV(e_2)}$

Let $g = \{l \in H \mid l \notin F \cup R \cup locs_{V',H}(e_2)\}$

NTS g nonempty, in particular, that $l \in g$

$l \notin F \cup R$ ($l \in locs_{V,H}(e)$ and $\mathsf{disjoint}(\{R, F, locs_{V,H}(e)\})$)

AFSOC $l \in locs_{V',H}(e_2)$

Then $l \in reach_H(V'(x'))$ for some $x' \in dom(V')$

case $x' \in \{x_h, x_t\}$:

WLOG let $x' = x_h$

But then $reach_H(V(x))(l) \geq 2$ and $\mathsf{set}(reach_(\overline{V}(x)))$ doesn't hold

case $x' \notin \{x_h, x_t\}$:

$x' \in dom(V)$

$x = x'$ ($l \in reach_H(V(x))$ and $\mathsf{linearCtxt}(V, H)$)

Contradiction, $x \notin domv(V')$

$l \notin locs_{V'',H}(e_2)$

Hence $l \in g$

By IH with $C' = C, V = V', H = H, R = R, F_1 = F \cup g$ we have the existence clause. NTS the following:

(1) $V_o', H_o \sim V', H$ (assumption)

(2) - (4) Same as Lemma 10

(5) $|F_1| \geq \Phi_{V',H}(\Gamma', x_h : A, x_t : L^p(A)) + q + p + 1 + C$:

$|F_1| = |F \cup g|$

$= |F| + |g|$ (F and g disjoint)

$\geq \Phi_{V,H}(\Gamma) + q + C + |g|$ (assumption)

$= \Phi_{V,H}(\Gamma', x_h : A, x_t : L^p(A)) + q + p + C + |g|$ (definition of Φ)

$= \Phi_{V,H}(\Gamma', x_h : A, x_t : L^p(A)) + q + p + 1 + C$ (g nonempty)

By existence clause have v, H', F' s.t.

Fact 1 .$V', H, R, F \cup g \vdash e_2 \Downarrow v, H', F'$

Fact 2 .$|F'| \geq \Phi_{H'}(v : B) + q' + C$

Apply E:MatCons and we're done

Case 14: E:Share

$V_o, H_o \vdash \mathtt{share}\ x\ \mathtt{as}\ x_1, x_2\ \mathtt{in}\ e' \Downarrow v_o, H_o'$ (case)

$V_o(x) = v_o'$ (ad.)

$V_o' = (V_o[x_1 \mapsto v_o', x_2 \mapsto v_o']) \upharpoonright_{FV(e')}$

$V_o', H_o \vdash e' \Downarrow v_o, H_o'$ (ad)

$\Sigma; \Gamma, x : A \,\Big|^{q}_{q'}\, \mathtt{share}\ x\ \mathtt{as}\ x_1, x_2\ \mathtt{in}\ e' : B$ (case)

$A \curlyvee A_1, A_2, 1$ (ad.)

$\Sigma; \Gamma, x_1 : A_1, x_2 : A_2 \,\Big|^{q}_{q'}\, e' : B$ (ad.)

Let $C \in \mathbb{Q}^+$, well-formed configuration V, H, R, F s.t. $V_o, H_o \sim V, H$ and $|F| \geq \Phi_{V,H}(\Gamma, x : A) + q + C$

NTF v, H'', F'' s.t.

1.$V, H, R, F \vdash e \Downarrow v, H'', F''$ and

2.$v_o \sim^{H'_o}_{H''} v$

3.$|F''| \geq \Phi_{H''}(v : B) + q' + C$

Let $V(x) = v', L \subseteq F, |L| = |dom(reach_H(v'))|, H', v'' = copy(H, L, v'), V' = (V[x_1 \mapsto v', x_2 \mapsto v'']) \restriction_{FV(e)}$
$F' = F \setminus L, g = \{l \in H \mid l \notin F' \cup R \cup locs_{V',H'}(e)\}$

By IH with $C, V', H', R, F' \cup g$, have the existence clause. NTS the following:

(1) $V'_o, H_o \sim V', H'$ (By Lemma 7 and assumption)

(2) - (4) same as lemma 10

(5) $|F' \cup g| \geq \Phi_{V',H'}(\Gamma, x_1 : A_1, x_2 : A_2) + q + C$

STS $|(F \setminus L) \cup g| \geq \Phi_{V',H'}(\Gamma, x_1 : A_1, x_2 : A_2) + q + C$

$\iff (|F| - |L|) + |g| \geq \Phi_{V',H'}(\Gamma) + \Phi_{V',H'}(x_1 : A_1) + \Phi_{V',H'}(x_2 : A_2) + q + C$

STS $|F| \geq \Phi_{V',H'}(\Gamma) + \Phi_{V',H'}(x_1 : A_1) + \Phi_{V',H'}(x_2 : A_2) + |L| + q + C$

$\iff |F| \geq \Phi_{V',H'}(\Gamma) + \Phi_{V,H}(x : A) + q + C$ (By Lemma 9)

$\iff |F| \geq \Phi_{V,H}(\Gamma, x : A) + q + C$ (By Lemma 7)

Instantiate the existence clause and apply E:Share, and we're done

□

D $\mathcal{E}_{\mathsf{copy}}$ over-approximates $\mathcal{E}_{\mathsf{gc}}$

Definition 12. A configuration (V, H, R, F) is well-formed if

1. $dom(H) \subseteq reach_H(V) \cup R \cup F$
2. $reach_H(V) \cup R \subseteq dom(H) \setminus F$
3. $collect(R, reach_H(V), H, F) = \emptyset$

For a context (V, H, R, F), denote the garbage w.r.t. a set of locations L as $collect(R, L, H', F') = \{l \in H' \mid l \notin F' \cup R \cup L\}$.

Lemma 22. *Given a well-formed context (V, H, R, F) and $V, H, R, F \vdash e \Downarrow v, H', F'$, $dom(H') \subseteq reach_H(v) \cup R \cup F'$, $reach_H(v) \cup R \subseteq dom(H') \setminus F'$ and $collect(R, reach'_H(v), H', F') = \emptyset$.*

Now consider two well-formed configurations $\mathcal{C}_1 = (V_1, H_1, R_1, F_1), \mathcal{C}_2 = (V_2, H_2, R_2, F_2)$.

A mapping $f : A \to \mathcal{P}(B)$ is a *partition* on B the image of A forms a disjoint union of B (e.g. $\forall x, y \in A, f(x) \cap f(y) = \emptyset \wedge \bigcup f(A) = B$). Furthermore, a partition is *proper* if for any $x \in A$, $f(x) \neq \emptyset$. Given a proper partition f, we can choose an arbitrary $b \in f(a)$ to be the representation for that part; call this singlet set $\{b\}$ $rep(a)$.

A simple corollary is the fact that if V_2, H_2 is a linear context (e.g. $\mathsf{linearCtxt}(V_2, H_2)$ holds), then $|\gamma(l)| = |(reach_{H_1}(V_1))(l)|$, where $reach_{H_1}(V_1) = \biguplus_{x \in dom(V)} reach_{H_1}(x)$. In general for a multiset S, when this holds, we say that γ is a *counting partition* for S.

For a partition $f : A \to \mathcal{P}(B)$, we write the set of equivalence classes as $ec(f) = \{f(x) \mid x \in A\} = f(A)$, i.e. the image of f on its domain A.

Definition 13. Let dir be the set $\{\mathsf{L},\mathsf{R},\mathsf{N}\}$, denoting left, right, and next respectively. We can index values via directions:

$$\begin{aligned} get_H(Just(\langle v_1, v_2\rangle, \mathsf{L})) &= Just(v_1) \\ get_H(Just(\langle v_1, v_2\rangle, \mathsf{R})) &= Just(v_2) \\ get_H(Just(\langle v_1, v_2\rangle),_{)} &= None \\ get_H(Just(l), \mathsf{N}) &= Just(H(l)) \\ get_H(Just(l), \_) &= None \\ get_H(r, \_) &= r \end{aligned}$$

Let P be a sequence of directions. Extend *get* to sequence of directions:

$$\begin{aligned} find_H(v, D :: P) &= find_H(get_H(v, D), P) \\ find_H(v, []) &= v \end{aligned}$$

Call P valid w.r.t. a value v if $find_H(v, P) = Just(v')$ for some v'. Write $V_H(x; P)$ for $fromJust(find_H(V(x), P))$ given a valid sequence P w.r.t. $V(x)$, and $reach_H(V(x; P))$ for $reach_H(V_H(x; P))$. Given a map $m : X \to \mathcal{S}(\mathsf{dir})$ from varibles to valid sequences of directions, Define $reachPath_{V,H}(X, m) = \biguplus_{x\in X} reach_H(V(x; m(x)))$.

Lemma 23. *Let $V_1, H_1 \sim V_2, H_2$. Then for all $x \in dom(V_1)$ and sequence of directions P, Either $find_{H_1}(V_1(x), P) = find_{H_2}(V_2(x), P) = None$ or $find_{H_1}(V_1(x), P) = v_1$, $find_{H_2}(V_2(x), P) = v_2$ and $v_1 \sim^{H_1}_{H_2} v_2$*

Proof. Induction on length of P and then $H \vdash v \mapsto a : A$. ☐

Definition 14. A configuration $\mathcal{C}_2 = (V_2, H_2, R_2, F_2)$ is a *copy extension* of another configuration $\mathcal{C}_1 = (V_1, H_1, R_1, F_1)$ iff

1. $V_1, H_1 \sim V_2, H_2$
2. There is a proper partition $\gamma : dom(H_1) \setminus F_1 \to \mathcal{P}(dom(H_2) \setminus F_2)$ such that for all $l \in dom(\gamma)$, $|\gamma(l)| = reach_{H_1}(V_1)(l) + R_1(l)$
3. for all $l \in dom(\gamma)$, $x \in dom(V_1)$, valid sequence of directions P w.r.t. $V_1(x)$, $|reach_{H_2}(V_2(x; P)) \cap \gamma(l)| = reach_{H_1}(V_1(x; P))(l)$.
4. for all $l \in dom(\gamma)$, $|\gamma(l) \cap R_2| = R_1(l)$
5. $|F_1| = |F_2| + |\oslash(\gamma)|$, where $\oslash(\gamma) = \bigcup_{P\in ec(\gamma)} C \setminus rep(C)$

Write this as $\mathcal{C}_1 \preceq \mathcal{C}_2$.

Note that $\preceq$ is reflexive. Now the key lemma:

Lemma 24. *Let $(\mathcal{C}_2, e)$ be a linear computation. Given that $\mathcal{C}_2 \vdash^{\mathsf{copy}} e \Downarrow v, H', F'$, for all well-formed configurations $\mathcal{C}_1$ such that $\mathcal{C}_1 \preceq \mathcal{C}_2$, there is exists a triple $(w, Y', M') \in \mathsf{Val} \times \mathsf{Heap} \times \mathsf{Loc}$ and $\gamma' : dom(Y') \setminus M' \to \mathcal{P}(dom(H') \setminus F')$ s.t.*

1. *$\mathcal{C}_1 \vdash^{\mathsf{free}} e \Downarrow w, Y', M'$*
2. *$v \sim^{H'}_{Y'} w$*
3. *γ' is a proper partition, such that for all $l \in dom(\gamma')$, $|\gamma'(l)| = reach_{Y_1}(w_1)(l) + S(l)$*
4. *For all P valid w.r.t. v, $|reach_{H'}(find_{H'}(v; P)) \cap \gamma'(l)| = reach_{Y'}(find_{Y'}(w; P))(l)$*
5. *$l \in dom(\gamma')$, $\gamma'(l) \cap R = \gamma(l) \cap R$*

6. $|M'| = |F'| + |\oslash(\gamma')|$

Lemma 25. *Let* $V_2, H_2, R_2, F_2 \vdash^{\mathsf{copy}} e \Downarrow v, H', F'$*, and* $V_1, H_1, R_1, F_1 \preceq V_2, H_2, R_2, F_2$ *because* $(-, \gamma, \eta, -, -)$*. Then the following hold: for all* $l \in dom(H_1) \setminus F_1$*,* $X \subseteq dom(V)$*,* $m : X \to \mathcal{S}(\mathsf{dir})$*,* $l \in dom(\gamma)$*,* $\gamma(l) \subseteq collect(R_2, \mathcal{C}_2, reachPath_{V_2,H_2}(X, m), H_2, F_2)$ *iff* $l \in collect(R_1, reachPath_{V_1,H_1}(X, m), H_1, F_1)$*.*

Proof. $\Longrightarrow$

$\gamma(l) \cap (F_2 \cup R_2 \cup reachPath_{V_2,H_2}(X, m)) = \emptyset$ (definition of *clean*)

NTS $l \notin F_1 \cup R_1 \cup reach_{H_1}(X)$

(1) $l \notin F_1$ (assumption)

(2) $l \notin R_1$

Know $\gamma(l) \cap R_2 = \emptyset$

$|\gamma(l) \cap R_2| = R_1(l) = 0$ (condition 4. of $\preceq$)

$l \notin R_1$

(3) $l \notin reachPath_{V_1,H_1}(X, m)$

Know $\gamma(l) \cap reachPath_{V_2,H_2}(X, m) = \emptyset$

Let $x \in X$

$\gamma(l) \cap reach_{H_2}(V_2(x; m(x))) = \emptyset$

$|\gamma(l) \cap reach_{H_2}(V_2(x; m(x)))| = reach_{H_1}(V_1(x; m(x)))(l)$ (condition 3. of $\preceq$)

$reach_{H_1}(V_1(x; m(x)))(l) = 0$

$reachPath_{V_1,H_1}(X, m)(l) = 0$

$l \notin reachPath_{V_1,H_1}(X, m)$

Thus, $l \in collect(R_1, reachPath_{V_1,H_1}(X, m), H_1, F_1)$

$\Longleftarrow$

$l \notin (F_1 \cup R_1 \cup reachPath_{V_1,H_1}(X, m))$

NTS $\gamma(l) \cap (F_2 \cup R_2 \cup reachPath_{V_2,H_2}(X, m)) = \emptyset$

(1) $\gamma(l) \cap F_2 = \emptyset$ (assumption)

(2) $\gamma(l) \cap R_2 = \emptyset$

$|\gamma(l) \cap R_2| = R_1(l)$ (condition 4. of $\preceq$)

$= 0$ (assumption)

$\gamma(l) \cap R_2 = \emptyset$

(3) $\gamma(l) \cap reachPath_{V_2,H_2}(X, m) = \emptyset$

Let $x \in X$

$|\gamma(l) \cap reach_{H_2}(V_2(x; m(x)))| = reach_{H_1}(V_1(x; m(x)))(l)$ (condition 3. of $\preceq$)

$= 0$ (assumption)

$\gamma(l) \cap reach_{H_2}(V_2(x; m(x))) = \emptyset$

$\gamma(l) \cap reachPath_{V_2,H_2}(X, m) = \emptyset$

$\gamma(l) \cap (F_2 \cup R_2 \cup reachPath_{V_2,H_2}(X, m)) = \emptyset$

$\gamma(l) \in collect(R_2, reachPath_{V_2,H_2}(X, m), H_2, F_2)$

$\square$

Lemma 26. *Given a proper partition* $\gamma : dom(Y) \setminus M \to \mathcal{P}(dom(H) \setminus F)$*, equivalent values* $v \sim_Y^H w$*, and the following:*

1. $L \subseteq F$
2. $|L| = |dom(reach_H(v))|$
3. $H', v' = copy(H, L, v)$
4. $F' = F \setminus L$

there is a proper partition $\gamma' : dom(Y) \setminus M \to \mathcal{P}(dom(H) \setminus F')$ *s.t.* $|\gamma'(l)| = |\gamma(l)| + reach_Y(w)(l)$, $|reach_H(v) \cap \gamma'(l)| = |reach_H(v') \cap \gamma'(l)| = |reach_H(v) \cap \gamma(l)|$.

Proof. Induction on *copy*. $\square$

Now the proving $\mathcal{E}_{\mathsf{copy}}$ over approximates $\mathcal{E}_{\mathsf{gc}}$:

Proof. Induction on the evaluation judgement.

Case 1: E:Var Trivial

Case 2: E:Const* Trivival

Case 4: E:App Similar to E:CondT

Case 5: E:CondT

$V, H, R, F \vdash^{\mathcal{E}_{\mathsf{copy}}} \mathtt{if}(x; e_1; e_2) \Downarrow v, H', F'$ (case)

Let $W, Y, S, M \preceq V, H, R, F$

Let $W' = W \restriction_{dom(V')}$

Let $j = \{l \in Y | l \notin M \cup S \cup locs_{W,Y}(e_1)\}$

NTS $W', Y, S, M \cup j \preceq V', H, R, F \cup g$

(1) $W', Y \sim V', H$ $(W, Y \sim V, H)$

(2) NTF a proper partition $\gamma' : dom(Y) \setminus (M \sqcup j) \to \mathcal{P}(dom(H) \setminus (F \cup g))$

Let $\gamma'(l) = \gamma \restriction_{dom(Y) \setminus (M \sqcup j)} (l) \setminus g$

First, show γ' is a partition

Let $l, l' \in dom(Y) \setminus (M \sqcup j)$

$\gamma'(l) \cap \gamma'(l') = \emptyset$ (γ is partition)

$\gamma'(dom(Y) \setminus (M \sqcup j)) = \gamma(dom(Y) \setminus (M \sqcup j)) \setminus g$

$= (\bigsqcup_{l \in dom(Y) \setminus M} \gamma(l) \setminus \bigsqcup_{l \in j} \gamma(l)) \setminus g$

$= ((dom(H) \setminus F) \setminus (\bigsqcup_{l \in j} \gamma(l))) \setminus g$ (γ is partition)

$= (dom(H) \setminus F) \setminus g$ ($\bigsqcup_{l \in j} \gamma(l) \subseteq g$ by lemma 4)

$= dom(H) \setminus (F \sqcup g)$

Hence γ' is a partition

γ' is also proper:

Let $l \in dom(Y) \setminus (M \sqcup j)$

$\gamma'(l) = \gamma(l) \setminus g$

AFSOC $\gamma(l) \subseteq g$

$l \in j$ (By Lemma 4)

Contradiction since assumed $l \notin j$

Now, NTS $|\gamma'(l)| = reach_Y(W')(l) + S(l)$

Let $l \in dom(\gamma')$

$\gamma'(l) = \gamma(l) \setminus g$ (definition)

$|\gamma'(l)| = |\gamma(l)| - |\gamma(l) \cap g|$

$= reach_Y(W)(l) + S(l) - |\gamma(l) \cap g|$

$g = reach_H(V \restriction_{dom(V)\setminus FV(e_1)})$

$|\gamma(l) \cap g| = reach_Y(W \restriction_{dom(W)\setminus FV(e_1)})(l)$ (condition 3. of $\preceq$)

Thus, $|\gamma'(l)| = reach_Y(W)(l) + S(l) - reach_Y(W \restriction_{dom(W)\setminus FV(e_1)})(l)$

$= reach_Y(W \restriction_{FV(e_1)})(l) + S(l)$

$= reach_Y(W')(l) + S(l)$

(3) Let $l \in dom(\gamma'), x \in dom(W')$, P valid sequence w.r.t. $W'(x)$. NTS $|reach_H(V'(x;P)) \cap \gamma'(l)| = reach_Y(W'(x;P))(l)$

STS $|reach_H(V'(x;P)) \cap (\gamma(l) \setminus g)| = reach_Y(W'(x;P))(l)$

$g \cap reach_H(V'(x;P)) = \emptyset$ (definition of g)

$|reach_H(V'(x;P)) \cap (\gamma(l) \setminus g)| = |(reach_H(V'(x;P)) \setminus g) \cap \gamma(l)|$

$= |reach_H(V'(x;P)) \cap \gamma(l)|$

$= |reach_H(V(x;P)) \cap \gamma(l)|$

$= reach_Y(W(x;P))(l)$ (condition 3. of $\preceq$)

$= reach_Y(W'(x;P))(l)$

(4) $S \subseteq dom(Y) \setminus (M \cup j)$ since $S \cap j = \emptyset$

Let $l \in S$. NTS $|\gamma'(l) \cap R| = S(l)$

STS $|(\gamma(l) \setminus g) \cap R| = S(l)$

$(\gamma(l) \setminus g) \cap R = \gamma(l) \cap (R \setminus g)$

$= \gamma(l) \cap R$ ($g \cap R = \emptyset$)

$|\gamma(l) \cap R| = S(l)$ (condition 4. of $\preceq$)

(5) NTS $|M \cup j| = |F \cup g| + |\oslash(\gamma')|$

STS $|M| + |j| = |F| + |g| + |\oslash(\gamma')|$

By assumption , $|M| = |F| + |\oslash(\gamma)|$

STS $|j| + |\oslash(\gamma)| = |g| + |\oslash(\gamma')|$

NTF a bijection $f : j \oplus \oslash(\gamma) \to g \sqcup \oslash(\gamma')$

First, we know that $g = (\bigsqcup_{l\in j} \gamma(l)) \sqcup L$ for some L (By 4)

Let $\mathcal{C}_1 = \{\gamma(l) \mid l \in j\}, \mathcal{C}_2 = ec(\gamma) \setminus \mathcal{C}_1$

Clearly, $\oslash(\gamma) = \bigsqcup_{C\in\mathcal{C}_1} C \setminus rep(C) \sqcup \bigsqcup_{C\in\mathcal{C}_2} C \setminus rep(C)$

Let $D_1 = \bigsqcup_{C\in\mathcal{C}_1} C \setminus rep(C), D_2 = \bigsqcup_{C\in\mathcal{C}_2} C \setminus rep(C)$

We define the bijection f by parts: $f_1 : j \oplus D_1 \to \bigsqcup_{C\in\mathcal{C}_2} C, f_2 : D_2 \to L \sqcup \oslash(\gamma')$

$$f_1(x) = \begin{cases} rep(\gamma(l)) & x = (\mathsf{inl}, l) \\ l & x = (\mathsf{inr}, l) \end{cases}$$

Clearly, f_1 is a bijection, and $|j| + |D_1| = |\bigsqcup_{C \in \mathcal{C}_1} C|$

To avoid the problem of maintaining a single representative for a class (which might be collected), note the following:

$|\mathcal{C}_2| = |ec(\gamma) \setminus \{\gamma(l) \mid l \in j\}|$

$= |ec(\gamma \restriction_{dom(Y) \setminus (M \sqcup j)})|$

$= |ec(\gamma')|$

Meaning that $\mathcal{C}_2$ has the same number of classes as γ'(note these classes might be different)

Since both γ, γ' are proper partitions we have the following:

$|D_2| = |L \sqcup \oslash(\gamma')|$ iff $|\bigsqcup_{C \in \mathcal{C}_2} C \setminus rep(C)| = |L \sqcup \bigsqcup_{C \in ec(\gamma')} C \setminus rep(C)|$ iff $|\bigsqcup_{C \in \mathcal{C}_2} C| = |L \sqcup \bigsqcup_{C \in ec(\gamma')} C|$

In fact, the latter two sets are equal:

let $l' \in \bigsqcup_{C \in \mathcal{C}_2} C$

$l' \in H \setminus F$ (Def. of partition)

case $l \in g$

$l' \notin \bigsqcup_{l \in j} \gamma(l)$ (Def. of $\mathcal{C}_2$)

$l' \in L$ (Def. of g)

$l' \in L \sqcup \bigsqcup_{C \in ec(\gamma')} C$

case $l' \notin g$

$l' \in H \setminus (F \sqcup g)$

Exists $C \in ec(\gamma')$ s.t. $l' \in C$ (Def. of partition)

$l' \in \bigsqcup_{C \in ec(\gamma')} C$

$l' \in L \sqcup \bigsqcup_{C \in ec(\gamma')} C$

For the other direction, let $l' \in L \sqcup \bigsqcup_{C \in ec(\gamma')} C$

case $l' \in L$

$l' \in H \setminus F$ (Def. of L)

Exists $C \in ec(\gamma)$ s.t. $l' \in C$ (Def. of partition)

$l' \in \bigsqcup_{C \in ec(\gamma)} C$

case $l' \in \bigsqcup_{C \in ec(\gamma')} C$

$l' \in H \setminus (F \sqcup g)$ (Def. of partition)

$l' \in H \setminus F$

Exists $C \in ec(\gamma)$ s.t. $l' \in C$ (Def. of partition)

$l' \in \bigsqcup_{C \in ec(\gamma)} C$

Hence we show that $|D_2| = |L \sqcup \oslash(\gamma')|$, and together with the previous equality, $|j| + |\oslash(\gamma)| = |g| + |\oslash(\gamma')|$

Thus we have $W', Y, S, F \cup j \preceq V', H, R, F \cup g$

$V', H, R, F \cup g \vdash^{\mathsf{copy}} e_1 \Downarrow v, H', F'$ (case)

By IH on $(V', H, R, F \cup g)$, we have (w, Y', M', γ'') such that

(1) $W', Y, S, M \cup j \vdash^{\mathcal{E}_{\mathsf{gc}}} e_1 \Downarrow w, Y', M'$

(2) $v \sim^{H'}_{Y'} w$

(3) γ' is a proper partition, and $|\gamma''(l)| = reach_{Y'}(w)(l) + R(l)$

(4) $\gamma''(l) \cap R = \gamma'(l) \cap R$

$\gamma'(l) \cap R = (\gamma(l) \setminus g) \cap R = \gamma(l) \cap R$

(5) $|M'| = |F'| + |\oslash(\gamma'')|$

Apply F:CondT to *(1)*, we are done.

Case 6: E:CondF Similar to E:CondT

Case 7: E:Let

$V, H, R, F \vdash^{\mathcal{E}_{\mathsf{copy}}} \mathtt{let}(e_1; x : \tau.e_2) \Downarrow v_2, H_2, F_2$ (case)

$W, Y, S, M \preceq V, H, R, F$ (assumption)

$V_1, H, R', F \vdash^{\mathcal{E}_{\mathsf{copy}}} e_1 \Downarrow v_1, H_1, F_1$ (admissibility)

Let $W_1 = W \restriction_{FV(e_1)}, S' = S \uplus locs_{W,Y}(\mathtt{lam}(x : \tau.e_2))$

NTS $W_1, Y, S', M \preceq V_1, H, R', F$

(1) $W_1, Y \sim V_1, H$ (condition 1. of $\preceq$)

(2)a. γ is a proper partition (condition 2. of $\preceq$)

(2).b NTS $|\gamma(l)| = reach_Y(W')(l) + S'(l)$

$|\gamma(l)| = reach_Y(W)(l) + S(l)$ (condition 2. of $\preceq$)

$= reach_Y(W')(l) + reach_Y(W \restriction_{FV(e_2)\setminus\{x\}})(l) + S(l)$

$= reach_Y(W')(l) + S'(l)$

(3) Let. $l \in dom(\gamma), x \in dom(W'), P$ valid sequence of directions. Have

$|reach_H(V'(x; P)) \cap \gamma(l) = reach_Y(W'(x; P))(l)$ (condition 3. of $\preceq$)

(4).a NTS $S' \subseteq dom(Y) \setminus M$

$S \subseteq dom(Y) \setminus M$ (condition 4. of $\preceq$)

$locs_{W,Y}(\mathtt{lam}(x : \tau.e_2)) \subseteq reach_Y(W) \subseteq dom(Y) \setminus M$ (well-formed configuration)

$S' \subseteq dom(Y) \setminus M$

(4).b Let $l \in S'$. NTS $|\gamma(l) \cap R'| = S'(l)$

STS $|\gamma(l) \cap (R \cup reach_H(FV(e_2) \setminus \{x\})| = S(l) + reach_Y(FV(e_2) \setminus \{x\})(l)$

STS $|(\gamma(l) \cap R) \cup (\gamma(l) \cap reach_H(V(FV(e_2) \setminus \{x\})))| = S(l) + reach_Y(W(FV(e_2) \setminus \{x\}))(l)$

STS $|(\gamma(l) \cap R)| + |\gamma(l) \cap reach_H(V(FV(e_2) \setminus \{x\}))| = S(l) + reach_Y(W(FV(e_2) \setminus \{x\}))(l)$

$(R \cap reach_H(V(FV(e_2) \setminus \{x\})) = \emptyset)$

STS $|\gamma(l) \cap reach_H(V(FV(e_2) \setminus \{x\}))| = reach_Y(W(FV(e_2) \setminus \{x\}))(l)$ (condition 4. of $\preceq$)

STS $|\gamma(l) \cap \bigcup_{x' \in FV(e_2)\setminus\{x\}} reach_H(V(x'))| = (\biguplus_{x' \in FV(e_2)\setminus\{x\}} reach_Y(W(x')))(l)$

Let $x' \in FV(e_2) \setminus \{x\}$

$|\gamma(l) \cap reach_H(V(x'))| = reach_Y(W(x'))(l)$ (condition 3. of $\preceq$)

(5) Have $|M| = |F| + |\oslash(\gamma)|$ (condition 5. of $\preceq$)

By IH on first premise, there is (w_1, Y_1, M_1) and γ_1 s.t.

Fact 1.$W', Y, S', M \vdash^{\mathcal{E}_{gc}} e \Downarrow w_1, Y_1, M_1$

Fact 2.$v \sim_{Y_1}^{H_1} w$

Fact 3.γ_1 is a proper partition, such that for all $l \in \mathit{dom}(\gamma_1)$,

$|\gamma_1(l)| = |\mathit{reach}_{Y_1}(w_1)(l)| + S(l)$

Fact 4.For all $P, |\mathit{reach}_{H_1}(\mathit{find}_{H_1}(v_1; P)) \cap \gamma_1(l)| = \mathit{reach}_{Y_1}(\mathit{find}_{Y_1}(w_1; P))(l)$

Fact 5.$\gamma_1(l) \cap R' = \gamma(l) \cap R'$ and $S' \subseteq \mathit{dom}(Y_1) \setminus M_1$

Fact 6.$|M_1| = |F_1| + |\oslash(\gamma_1)|$

$V_2, H_1, R, F_1 \cup g \vdash e_2 \Downarrow v_2, H_2, F_2$ (admissibility)

Let $W_2 = (W[x \mapsto w_1]) \restriction_{FV(e_2)}, j = \{l \in H_1 \mid l \notin M_1 \cup S \cup \mathit{locs}_{W_2, Y_1}(e_2)\}$

NTS $W_2, Y_1, S, M_1 \cup j \preceq V_2, H_1, R, F_1 \cup g$

(1) $W_2, Y_1 \sim V_2, H_1$ (condition 1. of $\preceq$ and Fact 1)

(2) Let $\gamma_2 : \mathit{dom}(Y_1) \setminus (M_1 \cup j) \to \mathcal{P}(\mathit{dom}(H_1) \setminus (F_1 \cup g))$ be defined by $\gamma_2(l) = \gamma_1(l) \setminus g$

NTS $|\gamma_2(l)| = \mathit{reach}_{Y_1}(W_2)(l) + S(l)$

STS $|\gamma_1(l)| = |\gamma_1(l) \cap g| + \mathit{reach}_{Y_1}(W_2)(l) + S(l)$

case $x \in FV(e_2)$:

$g = \mathit{collect}(R', v_1, H_1, F_1)$

$= \emptyset$ (By 22)

$|\gamma_1(l)| = \mathit{reach}_{Y_1}(w_1)(l) + S'(l)$ (Fact 3)

$= \mathit{reach}_{Y_1}(w_1)(l) + \mathit{reach}_Y(W \restriction_{FV(e_2)\setminus\{x\}})(l) + S(l)$

$= \mathit{reach}_{Y_1}(W_2)(l) + S(l)$

case $x \notin FV(e_2)$:

$g = \mathit{collect}(R', v_1, H_1, F_1) \sqcup \mathit{reach}_{H_1}(v_1)$

$= \mathit{reach}_{H_1}(v_1)$ (By 22)

$|\gamma_1(l) \cap g| = \mathit{reach}_{Y_1}(w_1)$ (Fact 3)

STS $|\gamma_1(l)| = \mathit{reach}_{Y_1}(w_1)(l) + \mathit{reach}_{Y_1}(W_2)(l) + S(l)$

$= \mathit{reach}_{Y_1}(w_1)(l) + S'(l)$ (Fact 3)

$= \mathit{reach}_{Y_1}(w_1)(l) + \mathit{reach}_Y(W \restriction_{FV(e_2)\setminus\{x\}})(l) + S(l)$

$= \mathit{reach}_{Y_1}(w_1)(l) + \mathit{reach}_{Y_1}(W \restriction_{FV(e_2)\setminus\{x\}})(l) + S(l)$ (By 6)

$= \mathit{reach}_{Y_1}(W_2)(l) + S(l)$

(3) Let $l \in \mathit{dom}(\gamma_2), x' \in \mathit{dom}(W_2)$, P valid sequence w.r.t. $W_2(x')$. NTS

$|\mathit{reach}_{H_1}(V_2(x'; P)) \cap \gamma_2(l)| = \mathit{reach}_{Y_1}(W_2(x'; P))(l)$

case $x' = x$:

$|\mathit{reach}_{H_1}(V_2(x'; P)) \cap \gamma_2(l)| = \mathit{reach}_{Y_1}(W_2(x'; P))(l)$ (Fact 4)

case $x' \neq x$:

$|\mathit{reach}_{H_1}(V_2(x'; P)) \cap \gamma(l)| = \mathit{reach}_{Y_1}(W_2(x'; P))(l)$ (stability and condition 3. of $\preceq$)

$|\mathit{reach}_{H_1}(V_2(x'; P)) \cap \gamma_1(l)| = \mathit{reach}_{Y_1}(W_2(x'; P))(l)$ ($\mathit{reach}_{H_1}(V_2(x'; P)) \subseteq R'$ and Fact 5.)

$|\mathit{reach}_{H_1}(V_2(x'; P)) \cap \gamma_2(l)| = \mathit{reach}_{Y_1}(W_2(x'; P))(l)$ ($g \cap R' = \emptyset$)

(4).a NTS $S \subseteq \mathit{dom}(Y_1) \setminus (M_1 \cup j)$

$S' \subseteq \mathit{dom}(Y_1) \setminus M_1$ (Fact 5.)

$S \subseteq dom(Y_1) \setminus M_1$

$S \cap j = \emptyset$

$S \subseteq dom(Y_1) \setminus (M_1 \cup j)$

(4).b Let $l \in S$. NTS $|\gamma_2(l) \cap R| = S(l)$

$|\gamma_2(l) \cap R| = |(\gamma_1(l) \setminus g) \cap R|$

$= |\gamma_1(l) \cap R|$ $(g \cap R = \emptyset)$

$= |\gamma(l) \cap R|$ (Fact 5.)

$= S(l)$ (condition 4. of $\preceq$)

(5) NTS $|M_1 \sqcup j| = |F_1 \sqcup g| + \oslash(\gamma_2)$

Exactly the same as in E:CondT

Thus $W_2, Y_1, S, M_1 \cup j \preceq V_2, H_1, R, F_1 \cup g$

By IH on the second premise, we have (w_2, Y_2, M_2) and γ_3 s.t.

Fact 1'.$W_2, Y_1, S, M_1 \cup j \vdash^{\mathcal{E}_{\text{gc}}} e_2 \Downarrow w_2, Y_2, M_2$

Fact 2'.$v_2 \sim^{H_2}_{Y_2} w_2$

Fact 3'.γ_3 is a proper partition, such that for all $l \in dom(\gamma_3)$,

$|\gamma_3(l)| = |reach_{Y_2}(w_2)(l)| + S(l) + |\gamma_3(l)|$

Fact 4'.For all $P, |reach_{H_2}(find_{H_2}(v_2; P)) \cap \gamma_3(l)| = reach_{Y_2}(find_{Y_2}(w_2; P))(l)$

Fact 5'.$\gamma_3(l) \cap R = \gamma_2(l) \cap R$ and $S \subseteq dom(Y_2) \setminus M_2$

$\gamma_2(l) \cap R = \gamma_1(l) \cap R$ $(g \cap R = \emptyset)$

$= \gamma(l) \cap R$ (Fact 5. from first premise)

Fact 6'.$|M_2| = |F_2| + |\oslash(\gamma_3)|$

Apply F:Let to *(1)*, we are done.

Case 8: E:Pair Similar to E:Const*

Case 9: E:MatP Similar to E:CondT

Case 10: E:Nil Similar to E:Const*

Case 11: E:Cons

$V, H, R, F \vdash \texttt{cons}(x_1; x_2) \Downarrow l, H', F \setminus \{l\}$ (case)

$W, Y, S, M \preceq V, H, R, F$ (assumption)

Let $w = \langle W(x_1), W(x_2) \rangle$

Let $m \in M, Y' = Y\{m \mapsto w\}$

(1) $V, H, R, F \vdash \texttt{cons}(x_1; x_2) \Downarrow m, Y', M \setminus \{m\}$ (F:Cons)

(2) $v \sim^H_Y w$ (assumption)

(3) Let $\gamma' : dom(Y') \setminus M' \to \mathcal{P}(dom(H') \setminus F')$ be defined by $\gamma'(l') = \gamma[m \mapsto \{l\}](l')$

γ' is a proper partition (γ proper and $l \notin dom(H) \setminus F$)

NTS $|\gamma'(l')| = reach_{Y'}(m)(l') + S(l')$

STS $|\gamma[m \mapsto \{l\}](l')| = reach_{Y'}(m)(l') + S(l')$

case $l' = m$:

$|\gamma[m \mapsto \{l\}](l')| = |\{l\}| = 1$

$reach_{Y'}(m)(l') + S(l') = 1 + reach_{Y'}(w_1)(m) + reach_{Y'}(w_2)(m) + S(l')$

$= 1 + S(m)$ $(M \cap reach_Y(W) = \emptyset)$

$= 1$ $(M \cap S = \emptyset)$

case $l' \neq m:$

$|\gamma[m \mapsto \{l\}](l')| = |\gamma(l')| = reach_H(W)(l') + S(l')$ (condition 2. of $\preceq$)

$= reach_{H'}(w_1)(l') + reach_{H'}(w_2)(l') + S(l')$

$= \{m \mapsto 1\}(l') + reach_{H'}(w_1)(l') + reach_{H'}(w_2)(l') + S(l')$

$= reach_{H'}(m)(l') + S(l')$

(4) Let $l \in dom(\gamma')$, P be a valid path w.r.t. l. NTS $|reach_{H'}(find_{H'}(l; P)) \cap \gamma'(l')| = reach_{Y'}(find_{Y'}(m; P))(l')$

case $P = []$

$|reach_{H'}(find_{H'}(l; P)) \cap \gamma'(l')| = |reach_{H'}(l) \cap \gamma'(l')|$

$= |(\{l\} \cup reach_{H'}(v_1) \cup reach_{H'}(v_2)) \cap \gamma'(l')|$

$= |\{l\} \cap \gamma(l')| + |reach_{H'}(v_1) \cap \gamma'(l')| + |reach_{H'}(v_2) \cap \gamma'(l')|$

$= \mathbb{1}_{l'=m} + reach_{Y'}(w_1)(l') + reach_{Y'}(w_2)(l')$ (condition 3. of $\preceq$)

$= reach_{Y'}(m)(l')$

$= reach_{Y'}(find_{Y'}(m; P))(l')$

case $P = \mathsf{N} :: P'$

$|reach_{H'}(find_{H'}(l; \mathsf{N} :: P')) \cap \gamma'(l')| = |reach_{H'}(\langle v_1, v_2 \rangle) \cap \gamma'(l')|$

$= reach_{Y'}(find_{Y'}(m; P))(l')$ (similar to above)

(5) NTS $S \subseteq dom(\gamma')$ (condition 4. of $\preceq$ and $dom(\gamma) \subseteq dom(\gamma')$)

Let $l' \in dom(\gamma')$. NTS $\gamma(l') \cap R = \gamma(l) \cap R$

STS $\gamma[m \mapsto \{l\}](l') \cap R = \gamma(l') \cap R$

case $l' = m:$

$\gamma[m \mapsto \{l\}](m) \cap R = \{l\} \cap R$

$= \emptyset$ (well-formed configuration)

$= \gamma(l') \cap R$ (well-formed configuration)

case $l' \neq m:$

$\gamma[m \mapsto \{l\}](m) \cap R = \gamma(l') \cap R$

(6) NTS $|M'| = |F'| + \oslash(\gamma')$

$|M'| = |M| - 1$

$|F'| = |F| - 1$

$\oslash(\gamma') = \oslash(\gamma[m \mapsto \{l\}])$

$= \bigcup C \in ec(\gamma')C \setminus (rep(C))$

$= (\{l\} \setminus \{l\}) \cup \bigcup C \in ec(\gamma)C \setminus (rep(C))$

$= \bigcup C \in ec(\gamma)C \setminus (rep(C))$

$= \bigcup C \in ec(\gamma)C \setminus (rep(C))$

$= \oslash(\gamma)$

STS $|M| - 1 = |F| - 1 + \oslash(\gamma)$

Have $|M| = |F| + \oslash(\gamma)$ (condition 5. of $\preceq$)

Case 12: E:MatNil Similar to E:Cond*

Case 13: E:MatCons

$V, H, R, F \vdash^{\mathcal{E}_{\mathsf{copy}}} \mathtt{match}\, x\, \{\mathtt{nil} \hookrightarrow e_1 \mid \mathtt{cons}(x_h; x_t) \hookrightarrow e_2\} \Downarrow v, H', F'$ (case)

$W, Y, S, M \preceq V, H, R, F$ (assumption)

$V', H, R, F \cup g \vdash e_2 \Downarrow v, H', F'$ (admissibility)

Let $W' = W \restriction_{dom(V')}$

Let $j = \{l \in Y | l \notin M \cup S \cup locs_{W',Y}(e_2)\}$

NTS $W', Y, S, M \cup j \preceq V', H, R, F \cup g$

Let $\gamma' : dom(Y) \setminus (M \sqcup j) \to \mathcal{P}(dom(H) \setminus (F \sqcup g))$ be defined by $\gamma'(l) = \gamma(l) \setminus g$

(1) $W', Y \sim V', H$ (similar to E:CondT)

Let $W(x) = m$

$H(m) = \langle w_h, w_t \rangle, v_h \sim^H_Y w_h, v_t \sim^H_Y w_t$ ((1))

(2) $\gamma'(l)$ is a proper partition . (similar to E:CondT)

Let $l' \in dom(\gamma')$

Now, NTS $|\gamma'(l')| = reach_Y(W')(l) + S(l')$

Consider $|\gamma(l') \cap reach_H(V(x))|$

$= |\gamma(l') \cap \{l\}| + |\gamma(l') \cap reach_H(v_h)| + |\gamma(l') \cap reach_H(v_t)|$

$= reach_Y(W(x))(l')$ (condition 3. of $\preceq$)

$= (\{m \mapsto 1\} \uplus reach_Y(w_h) \uplus reach_Y(w_t))(l')$

$= \{m \mapsto 1\}(l') + reach_Y(w_h)(l') + reach_Y(w_t)(l')$

Note $|\gamma(l') \cap reach_H(v_h)| = |\gamma(l') \cap reach_H(V(x; [\mathsf{N}, \mathsf{L}]))|$

$= reach_Y(W(x; [\mathsf{N}, \mathsf{L}]))(l')$

$= reach_Y(w_h)(l')$

Similarly, $|\gamma(l') \cap reach_H(v_h)| = reach_Y(w_h)(l')$

Thus $|\gamma(l') \cap \{l\}| = \{m \mapsto 1\}(l') = \mathbb{1}_{l'=m}$

Back to the NTS:

$\gamma'(l') = \gamma(l') \setminus g$ (definition)

$|\gamma'(l')| = |\gamma(l')| - |\gamma(l') \cap g|$

$= reach_Y(W)(l') + S(l') - |\gamma(l') \cap g|$

case $\{x_h, x_t\} \subseteq FV(e_2)$:

$g = \{l\} \cup reach_H(V \restriction_{dom(V) \setminus (FV(e_2) \cup \{x\})})$ (definition of g and 10)

$|\gamma(l') \cap g| = |(\gamma(l') \cap \{l\}) \cup (\gamma(l') \cap reach_H(V \restriction_{dom(V) \setminus (FV(e_2) \cup \{x\})}))|$

(condition 3. of $\preceq$)

Thus, $|\gamma'(l')| = reach_Y(W)(l') + S(l') - (\mathbb{1}_{l'=m} + reach_Y(W \restriction_{dom(W) \setminus (FV(e_2) \cup \{x\})})(l'))$

$= reach_Y(W \restriction_{FV(e_2) \cup \{x\}})(l') + S(l') - \mathbb{1}_{l'=m}$

$= reach_Y(W \restriction_{FV(e_2)})(l') + reach_Y(m)(l') + reach_Y(w_h)(l') + reach_Y(w_t)(l') + S(l') - \mathbb{1}_{l'=m}$

$= reach_Y(W \restriction_{FV(e_2)})(l') + reach_Y(w_h)(l') + reach_Y(w_t)(l') + S(l')$

$= reach_Y(W')(l') + S(l')$

case $x_h \in FV(e_2), x_t \notin FV(e_2)$:

$g = \{l\} \cup reach_H(V \restriction_{dom(V) \setminus (FV(e_2) \cup \{x\})}) \cup reach_H(v_t)$ (definition of g and 10)

$|\gamma(l') \cap g| = |\gamma(l') \cap \{l\}| + |\gamma(l') \cap reach_H(V \restriction_{dom(V) \setminus (FV(e_2) \cup \{x\})})| + |\gamma(l') \cap reach_H(v_t)|$

(condition 3. of $\preceq$)

$= \mathbb{1}_{l'=m} + reach_H(V \restriction_{dom(V)\setminus(FV(e_2)\cup\{x\})})(l') + reach_Y(w_t)(l')$ (condition 3. of $\preceq$)

Thus, $|\gamma'(l')| = reach_Y(W)(l') + S(l') - (\mathbb{1}_{l'=m} + reach_Y(W \restriction_{dom(W)\setminus(FV(e_2)\cup\{x\})})(l') + reach_Y(w_t)(l))$

$= reach_Y(W \restriction_{FV(e_2)})(l') + reach_Y(m)(l') + reach_Y(w_h)(l') + reach_Y(w_t)(l') + S(l') - \mathbb{1}_{l'=m} - reach_Y(w_t)(l))$

$= reach_Y(W \restriction_{FV(e_2)})(l') + reach_Y(w_h)(l') + S(l')$

$= reach_Y(W')(l') + S(l')$

case $x_t \in FV(e_2), x_h \notin FV(e_2)$ and $\{x_h, x_t\} \cap FV(e_2) = \emptyset :$ symmetric to above

(3) Let $l' \in dom(\gamma'), x' \in dom(W')$, P valid sequence w.r.t. $W'(x')$. NTS

$|reach_H(V'(x';P)) \cap \gamma'(l')| = reach_Y(W'(x';P))(l')$

case $x' \notin \{x_h, x_t\} :$

$|reach_H(V'(x';P)) \cap \gamma'(l')| = |reach_H(V'(x';P)) \cap (\gamma(l') \setminus g)|$

$= |reach_H(V'(x';P)) \cap \gamma(l')|$ ($g \cap reach_H(V'(x)) = \emptyset$)

$= reach_Y(W'(x;P))(l')$

$= reach_Y(W(x;P))(l')$

case $x' = x_h :$

$|reach_H(V'(x';P)) \cap \gamma'(l')| = |reach_H(find_H(v_h, P)) \cap \gamma'(l')|$

$|reach_H(find_H(v_h, P)) \cap (\gamma(l') \setminus g)|$

$|reach_H(find_H(v_h, P)) \cap \gamma(l')|$ ($g \cap reach_H(V') = \emptyset$)

$= |reach_H(find_H(l, \mathsf{N} :: \mathsf{L} :: P)) \cap \gamma(l')|$

$= |reach_H(V(x, \mathsf{N} :: \mathsf{L} :: P)) \cap \gamma(l')|$

$= reach_Y(W(x, \mathsf{N} :: \mathsf{L} :: P))(l')$ (condition 3. of $\preceq$)

$= reach_Y(find_H(m, \mathsf{N} :: \mathsf{L} :: P))(l')$

$= reach_Y(find_H(v_h, P))(l')$

$= reach_Y(W'(x_h; P))(l')$

(4) Similar to E:CondT

(5) Similar to E:CondT

Apply IH and F:MatCons and we're done

Case 14: E:Share

$V, H, R, F \ \vdash^{\mathcal{E}_{\mathsf{copy}}} \texttt{share}\ x\ \texttt{as}\ x_1, x_2\ \texttt{in}\ e \Downarrow v, H'', F'$ (case)

$W, Y, S, M \preceq V, H, R, F$ (assumption)

$V', H', R, F' \sqcup g \ \vdash e \Downarrow v, H'', F'$ (admissibility)

Let $W(x) = w'$

$v' \sim^Y_H w'$ (condition 1 of. $\preceq$)

Let $W' = (W[x_1 \mapsto w', x_2 \mapsto w']) \restriction_{FV(e)}$

Let $j = \{l \in Y | l \notin M \cup S \cup locs_{W',Y}(e)\}$

NTS $W', Y, S, M \cup j \preceq V', H, R, F \cup g$

Let $\gamma' : dom(Y) \setminus (M \sqcup j) \to \mathcal{P}(dom(H) \setminus (F \sqcup g))$ be defined by $\gamma'(l) = \gamma(l) \setminus g$

(1) $W', Y \sim V', H$ (Similar to E:CondT)

(2) Have proper partition $\gamma' : dom(Y) \setminus M \to dom(H') \setminus F'$ (By 26)

Let $j = \{l \in H \mid l \notin M \cup S \cup locs_{W',Y}(e)\}$

Let $\gamma''(l) : dom(Y) \setminus (M \cup j) \to dom(H') \setminus (F' \cup g)$ by defined by $\gamma''(l) = \gamma'(l) \setminus g$

γ'' is a proper partition (Similar to E:CondT)

Let $l \in dom(\gamma'')$

Now, NTS $|\gamma''(l)| = reach_Y(W')(l) + S(l')$

$\gamma''(l) = \gamma'(l) \setminus g$ (definition)

$|\gamma''(l)| = |\gamma'(l)| - |\gamma'(l) \cap g|$

$= |\gamma(l)| + \mathbb{1}_{l \in reach_Y(w')} - |\gamma'(l) \cap g|$

$= reach_Y(W)(l) + S(l) + reach_Y(w')(l) - |\gamma'(l) \cap g|$

case $\{x_1, x_2\} \subseteq FV(e)$:

$g = collect(R, reach_H(V), H, F) = \emptyset$ (well-formed configuration)

$|\gamma''(l)| = reach_Y(W)(l) + S(l) + reach_Y(w')(l)$ (By 26)

$= reach_Y(W')(l) + S(l)$

case $x_1 \in FV(e), x_2 \notin FV(e)$:

$g = collect(R, reach_H(V), H, F) \cup reach'_H(v') = reach_{H'}(v')$ (well-formed configuration)

$|\gamma''(l)| = reach_Y(W)(l) + S(l) + reach_Y(w')(l) - |\gamma'(l) \cap reach_{H'}(v')|$

$= reach_Y(W)(l) + S(l) + reach_Y(w')(l) - |\gamma(l) \cap reach_H(v')|$ (By 26)

$= reach_Y(W)(l) + S(l) + reach_Y(w')(l) - reach_Y(w')(l)$ (condition 3. of $\preceq$)

$= reach_Y(W)(l) + S(l)$

$= reach_Y(W')(l) + S(l)$

case $x_2 \in FV(e), x_1 \notin FV(e)$ and $\{x_1, x_2\} \cap FV(e) = \emptyset$: symmetric to above

(3) - *(5)* Similar to E:MatCons

Applying the IH then F:Share, and we're done.

□