Automatic Resource Bound Analysis for Functional Programs with Garbage Collection

Andrew Carnegie
(andrew)

July 22, 2018

Abstract

In todays complex software systems, resource consumption such as time, memory, and energy is increasingly more relevant. Manually analyzing and predicting resource consumption is cumbersome and error prone. As a result, programming language research studies techniques that support developers by automating and guiding such a resource analysis. Most current techniques for resource bound inference work in an environment with manual memory management, or simply assume there is no automatic memory management. As a result, memory bounds derived by the aforementioned techniques are not accurate when the underlying system employs garbage collection. We develop new techniques for automatic bound inference in the presence of garbage collection, and implement them as an extension to Resource Aware ML (RaML). We show that our system is able to derive tight symbolic bounds for heap usage of common algorithms for which current analysis give asymptotically worse bounds.

1 Introduction

We present an algorithm for statically determining the symbolic cost bound of the space complexity for first-order, garbage-collected functional programs. The algorithm is based on Automatic Amortized Resource Analysis (AARA) as implemented in Resource Aware ML (RaML) [1].

1.1 Background

We define the execution of the given language by giving it an operational cost semantics. The semantics is a big-step (or natural) semantics in the style of Spoonhower and Minamide ([4] and [3]), which assigns any program a space cost. This cost is the highwater mark on the heap, or the maximum number of cells used in the mutable store during evaluation. While [4] and [3] compute this cost at the leaves of a evaluation judgment, we compute the cost "as it happens" by leveraging the *free-list*, which represents the cells available for evaluation. The concept of the free-list can be found in Hofmann et. al. ([2]). However, unlike [2], in order to account for garbage collection, we define the free-list as a set of locations instead of single number denoting its size.

The abstract and concrete syntax of the language is show below. Note that we only allow first order functions of type $\tau_1 \to \tau_2$, where τ_1 and τ_2 are base types: unit, bool, product, or lists. In this paper, we restrict the language to terms that are in let-normal form to simplify the presentation. We allow an extended syntax in the implementation.

BTypes	τ ::=		
	$\texttt{nat}$	$\texttt{nat}$	naturals
	$\texttt{unit}$	$\texttt{unit}$	unit
	$\texttt{bool}$	$\texttt{bool}$	boolean
	$\texttt{prod}(\tau_1; \tau_2)$	$\tau_1 \times \tau_2$	product
	$\texttt{list}(\tau)$	$L(\tau)$	list
FTypes	ρ ::=		
	$\texttt{arr}(\tau_1; \tau_2)$	$\tau_1 \to \tau_2$	first order function
Exp	e ::=		
	x	x	variable
	$\texttt{nat}[n]$	$\overline{n}$	number
	$\texttt{unit}$	$()$	unit
	$\texttt{T}$	$\texttt{T}$	true
	$\texttt{F}$	$\texttt{F}$	false
	$\texttt{plus}(e_1; e_2)$	$e_1 + e_2$	plus
	$\texttt{minus}(e_1; e_2)$	$e_1 - e_2$	minus
	$\texttt{eq}(e_1; e_2)$	$e_1 = e_2$	equality
	$\texttt{lt}(e_1; e_2)$	$e_1 < e_2$	less-than
	$\texttt{and}(e_1; e_2)$	$e_1 \wedge e_2$	conjunction
	$\texttt{or}(e_1; e_2)$	$e_1 \vee e_2$	disjunction
	$\texttt{not}(e)$	$\neg e$	negation
	$\texttt{if}(x; e_1; e_2)$	$\texttt{if}\, x \,\texttt{then}\, e_1 \,\texttt{else}\, e_2$	if
	$\texttt{lam}(x : \tau.e)$	$\lambda\, x : \tau.e$	abstraction
	$\texttt{ap}(f; x)$	$f(x)$	application
	$\texttt{tpl}(x_1; x_2)$	$\langle x_1, x_2 \rangle$	pair
	$\texttt{case}(x_1, x_2.e_1)$	$\texttt{case}\ p\ \{(x_1; x_2) \hookrightarrow e_1\}$	match pair
	$\texttt{nil}$	$[]$	nil
	$\texttt{cons}(x_1; x_2)$	$x_1 :: x_2$	cons
	$\texttt{case}\{l\}(e_1; x, xs.e_2)$	$\texttt{case}\, l\, \{\texttt{nil} \hookrightarrow e_1 \mid \texttt{cons}(x; xs) \hookrightarrow e_2\}$	match list
	$\texttt{let}(e_1; x : \tau.e_2)$	$\texttt{let}\ x = e_1\ \texttt{in}\ e_2$	let
	$\texttt{share}(x; x_1, x_2.e)$	$\texttt{share}\ x\ \texttt{as}\ x_1, x_2\ \texttt{in}\ e$	share
Val	v ::=		
	$\texttt{val}(n)$	n	numeric value
	$\texttt{val}(\texttt{T})$	$\texttt{T}$	true value
	$\texttt{val}(\texttt{F})$	$\texttt{F}$	false value
	$\texttt{val}(\texttt{Null})$	$\texttt{Null}$	null value
	$\texttt{val}(\texttt{cl}(V; x.e))$	$(V, x.e)$	function value
	$\texttt{val}(l)$	l	loc value
	$\texttt{val}(\texttt{pair}(v_1; v_2))$	$\langle v_1, v_2 \rangle$	pair value
Loc	l ::=		
	$\texttt{loc}(l)$	l	location
Var	l ::=		
	$\texttt{var}(x)$	x	variable

2 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.

3 Garbage collection semantics

The garbage collection operation semantics is defined by a collection of judgement of the form:

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

Where $\mathcal{C}$ is a *configuration*, consisting of a 4-tuple in $\mathsf{Stack} \times \mathsf{Heap} \times \mathcal{P}(\mathsf{Loc}) \times \mathcal{P}(\mathsf{Loc})$, usually written as V, H, R, F. P is a program with signature $\Sigma : \mathsf{Var} \to \mathsf{FTypes}$. This can be read as: under stack V, heap H, continuation set R, free-list F, and program P with signature Σ, the expression e evaluates to v, and engenders a new heap H' and freelist F'. Here, Stack is defined as the set of finite mappings $\mathsf{Var} \to \mathsf{Val}$, and Heap is defined as the set of finite mappings $\mathsf{Loc} \to \mathsf{Val}$.

A *program* is then a Σ indexed map P from Var to pairs $(y_f, e_f)_{f \in \Sigma}$, where $\Sigma(y_f) = A \to B$, and $\Sigma; y_f : A \vdash e_f : B$ (typing rules are discussed in 7). We write $P : \Sigma$ to mean P is a program with signature Σ. Because the signature Σ for the mapping of function names to first order functions does not change during evaluation, we drop the subscript Σ from $\vdash_\Sigma$ when the context of evaluation is clear. It is convenient to think of the evaluation judgement $\vdash$ as being indexed by a family of signatures Σ's, each of which is a set of "top-level" first-order declarations to be used during evaluation.

The garbage collection semantics is to designed to model the heap usage of a program running with a "perfect" reference counting garbage collector: whenever a heap cell becomes unreachable from the root set, it becomes collected and added to the free-list as available for reallocation. The continuation set R represents the set of locations required to compute the continuation *excluding* the current expression. We can think of R heap cells needed to compute the (surrounding) context with a hole which is filled by the current expression. The root set is then the union of the locations in the continuation set R and the locations in the current expression e. Primitives and types with statically-known sizes are stack-allocated ($\texttt{bool}, \texttt{nat}, \tau_1 \times \tau_2$) and use no heap cells; this is reflected in the reachability predicate defined below.

3.1 Reachability

In order to formalize the intuitive operations of the garbage collector, we define some auxiliary relations.

We define the 3-place reachability relation $reach(H, v, L)$ on $\mathsf{Heap} \times \mathsf{Val} \times \wp(\mathsf{Loc})$. We write $L = reach_H(v)$ to indicate this is a functional relation justified by the valid mode $(+, +, -)$.

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

The notion of reachability naturally lifts to expressions via $locs_{V,H}$:

$$locs_{V,H}(e) = \bigcup_{x \in FV(e)} reach_H(V(x))$$

Where $FV : \mathsf{Exp} \to \mathcal{P}(\mathsf{Var})$ denotes the set of free-variables of expressions as usual. We 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; the only unusual thing is that multiple occurences of a free variable x in e will be reflected in the multiplicity of $FV^\star(e)$.

3.2 The Copying Semantics copy

As mentioned in the introduction, copy is an intermediary semantics in which all variable sharing is accomplished by physically allocating a fresh set of locations from the free-list and copying the cells of the original value one by one. This is also sometimes referred to as deep copying. Let $copy(H, L, v, H', v')$ 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 $H', v = copy(H, L, v)$ to signify the intended mode for this predicate: $(+, +, +, -, -)$.

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

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

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.

3.3 Rules for copy

$$\frac{V(x) = v}{V, H, R, F \vdash x \Downarrow v, H, F}(\mathrm{S}_1) \qquad \frac{}{V, H, R, F \vdash \overline{n} \Downarrow \mathtt{val}(n), H, F}(\mathrm{S}_2) \qquad \frac{}{V, H, R, F \vdash \mathtt{T} \Downarrow \mathtt{val}(\mathtt{T}), H, F}(\mathrm{S}_3)$$

$$\frac{}{V, H, R, F \vdash \mathtt{F} \Downarrow \mathtt{val}(\mathtt{F}), H, F}(\mathrm{S}_4) \qquad \frac{}{V, H, R, F \vdash () \Downarrow \mathtt{val}(\mathtt{Null}), H, F}(\mathrm{S}_5)$$

$$\frac{V(x) = \mathtt{T} \qquad g = \{l \in H \mid l \notin F \cup R \cup \mathit{locs}_{V,H}(e_1)\} \qquad V' = V \restriction_{FV(e_1)} \qquad V', H, R, F \cup g \vdash e_1 \Downarrow v, H', F'}{V, H, R, F \vdash \mathtt{if}(x; e_1; e_2) \Downarrow v, H', F'}(\mathrm{S}_6)$$

$$\frac{V(x) = \mathtt{F} \qquad g = \{l \in H \mid l \notin F \cup R \cup \mathit{locs}_{V,H}(e_2)\} \qquad V' = V \restriction_{FV(e_2)} \qquad V', H, R, F \cup g \vdash e_2 \Downarrow v, H', F'}{V, H, R, F \vdash \mathtt{if}(x; e_1; e_2) \Downarrow v, H', F'}(\mathrm{S}_7)$$

$$\frac{\begin{array}{c} V(x) = v' \\ P(f) = (y_f, e_f) \qquad g = \{l \in H \mid l \notin F \cup R \cup \mathit{locs}_{V,H}(e_f)\} \qquad [y_f \mapsto v'], H, R, F \cup g \vdash e_f \Downarrow v, H', F' \end{array}}{V, H, R, F \vdash f(x) \Downarrow v, H', F'}(\mathrm{S}_8)$$

$$\frac{V(x_1) = v_1 \qquad V(x_2) = v_2}{V, H, R, F \vdash \langle x_1, x_2 \rangle \Downarrow \langle v_1, v_2 \rangle, H, F}(\mathrm{S}_9)$$

$$\frac{\begin{array}{c} V(x) = \langle v_1, v_2 \rangle \\ g = \{l \in H \mid l \notin F \cup R \cup \mathit{locs}_{V,H}(e)\} \qquad V'' = (V[x_1 \mapsto v_1, x_2 \mapsto v_2]) \restriction_{FV(e)} \qquad V'', H, R, F \cup g \vdash e \Downarrow v, H', F' \end{array}}{V, H, R, F \vdash \mathtt{case}\ x\ \{(x_1; x_2) \hookrightarrow e\} \Downarrow v, H', F'}(\mathrm{S}_{10})$$

$$\frac{}{V, H, R, F \vdash \mathtt{nil} \Downarrow \mathtt{val}(\mathtt{Null}), H, F}(\mathrm{S}_{11}) \qquad \frac{v = \langle V(x_1), V(x_2) \rangle \qquad l \in F \qquad H' = H\{l \mapsto v\}}{V, H, R, F \vdash \mathtt{cons}(x_1; x_2) \Downarrow l, H', F \setminus \{l\}}(\mathrm{S}_{12})$$

$$\frac{\begin{array}{c} V(x) = v \\ L \subseteq F \qquad |L| = |\mathit{dom}(\mathit{reach}_H(v'))| \qquad H', v'' = \mathit{copy}(H, L, v') \qquad V_2 = (V[x_1 \mapsto v', x_2 \mapsto v'']) \restriction_{FV(e)} \\ F' = F \setminus L \qquad g = \{l \in H \mid l \notin F' \cup R \cup \mathit{locs}_{V_2,H}(e)\} \qquad V_2, H', R, F' \sqcup g \vdash e \Downarrow v, H'', F' \end{array}}{V, H, R, F \vdash \mathtt{shareCopy}\ x\ \mathtt{as}\ x_1, x_2\ \mathtt{in}\ e \Downarrow v, H'', F'}(\mathrm{S}_{13})$$

$$\frac{\begin{array}{c} V(x) = \mathtt{Null} \\ V' = V \restriction_{FV(e_1)} \qquad g = \{l \in H \mid l \notin F \cup R \cup \mathit{locs}_{V,H}(e_1)\} \qquad V', H, R, F \cup g \vdash e_1 \Downarrow v, H', F' \end{array}}{V, H, R, F \vdash \mathtt{case}\ x\ \{\mathtt{nil} \hookrightarrow e_1 \mid \mathtt{cons}(x_h; x_t) \hookrightarrow e_2\} \Downarrow v, H', F'}(\mathrm{S}_{14})$$

$$\frac{\begin{array}{c} V(x) = l \qquad H(l) = \langle v_h, v_t \rangle \qquad V'' = (V[x_h \mapsto v_h, x_t \mapsto v_t]) \restriction_{FV(e_2)} \\ g = \{l \in H \mid l \notin F \cup R \cup \mathit{locs}_{V'',H}(e_2)\} \qquad V'', H, R, F \cup g \vdash e_2 \Downarrow v, H', F' \end{array}}{V, H, R, F \vdash \mathtt{case}\ x\ \{\mathtt{nil} \hookrightarrow e_1 \mid \mathtt{cons}(x_h; x_t) \hookrightarrow e_2\} \Downarrow v, H', F'}(\mathrm{S}_{15})$$

$$\frac{\begin{array}{c} V_1 = V \restriction_{FV(e_1)} \\ R' = R \cup \mathit{locs}_{V_2,H}(\mathtt{lam}(x : \tau.e_2)) \qquad V_1, H, R', F \vdash e_1 \Downarrow v_1, H_1, F_1 \qquad V_2' = (V[x \mapsto v_1]) \restriction_{FV(e_2)} \\ g = \{l \in H_1 \mid l \notin F_1 \cup R \cup \mathit{locs}_{V_2',H_1}(e_2)\} \qquad V_2', H_1, R, F_1 \cup g \vdash e_2 \Downarrow v_2, H_2, F_2 \end{array}}{V, H, R, F \vdash \mathtt{let}(e_1; x : \tau.e_2) \Downarrow v_2, H_2, F_2}(\mathrm{S}_{16})$$

As a simple example, take the rule (E:CondT):

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

This 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. In the process, we update the stack so it only binds variables reachable from the first branch. This update is only necessary to prove properties about the evaluation, and is not needed in an implementation. Furthermore, we collect cells in the heap that are not reachable from the root set $(R \cup locs_{V,H}(e_1))$ or already in the current free-list F, and add them (g) to the available cells for evaluating e_1. We could have removed the restriction that the cells be not in the current free-list F, but this simplifies the presentation of the proofs and the actual check can be removed in an implementation as the free-list is a set.

Next, we look at a more complicated rule, (E:MatchCons):

$$\frac{\begin{array}{c} V(x) = l \qquad H(l) = \langle v_h, v_t \rangle \qquad V' \subseteq V \qquad dom(V') = FV(e_2) \setminus \{x_h, x_t\} \\ V'' = V'[x_h \mapsto v_h, x_t \mapsto v_t] \qquad g = \{l \in H \mid l \notin F \cup R \cup locs_{V'',H}(e_2)\} \qquad V'', H, R, F \cup g \vdash e_2 \Downarrow v, H', F' \end{array}}{V, H, R, F \vdash \mathtt{case}\, x\, \{\mathtt{nil} \hookrightarrow e_1 \mid \mathtt{cons}(x_h; x_t) \hookrightarrow e_2\} \Downarrow v, H', F'}\text{(E:MatchCons}$$

This states that, to map out of list, look in the stack for the value of the list. In the case it is a heap location l, we check that it is mapped to a pair representing the head and tail. Similar to the conditional, we restrict the stack to only variables in the cons branch, and collect all heap locations not reachable from the root set, adding it to the current free-list F. Finally, we evaluate e_2 with the restricted stack and new free-list.

As the last (and most involved) example, we look at the rule (E:Let):

$$\frac{\begin{array}{c} V = V_1 \sqcup V_2 \qquad dom(V_1) = FV(e_1) \\ dom(V_2) = FV(\mathtt{lam}(x : \tau.e_2)) \qquad R' = R \cup locs_{V_2,H}(\mathtt{lam}(x : \tau.e_2)) \qquad V_1, H, R', F \vdash e_1 \Downarrow v_1, H_1, F_1 \\ V_2' = V_2[x \mapsto v_1] \qquad g = \{l \in H_1 \mid l \notin F_1 \cup R \cup locs_{V_2',H_1}(e_2)\} \qquad V_2', H_1, R, F_1 \cup g \vdash e_2 \Downarrow v_2, H_2, F_2 \end{array}}{V, H, R, F \vdash \mathtt{let}(e_1; x : \tau.e_2) \Downarrow v_2, H_2, F_2}(\mathrm{S}_{17})$$

This states that, to evaluate a sequence of expressions (e_1, e_2), we evaluate the first expression with the corresponding restricted stack V_1 and a 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 extend the restricted stack V_2 with the result v_1, and evaluate e_2 with this stack and the original continuation set R.

Of course, in many real-world languages, variable aliasing does not actually incur overhead proportional to the size of the value; instead, the aliased value simply gains one more reference. free (rules introduced in **??**) is the version of the garbage collection semantics designed to capture "shallow" copying. It is identical to copy except for the variable sharing rule. We use copy to mediate between the type system and free. The chart below shows the plan for proving the soundess of the type system w.r.t free:

$$\begin{tikzcd}
\boxed{\text{Type Rules}} \arrow[r, "\text{Is sound w.r.t (\textbf{??})}"] \arrow[dr, dashed] & \boxed{\mathsf{copy}} \arrow[d, "\text{Over-approximates (\textbf{??})}"] \\
 & \boxed{\mathsf{free}}
\end{tikzcd}$$

Where the dashed arrow represents the intended result.

4 Operational semantics

In order to state and prove the soundess of the type system, we also define a simplified operational semantics that does not account for garbage collection. This technique is similar to the one employed in [2].

$$\boxed{\mathcal{S} \vdash e \Downarrow v, H'}$$

Where $\mathcal{S}$ is a *context*, consisting of a tuple in $\mathsf{Stack} \times \mathsf{Heap}$, and usually written as (V, H). This can be read as: under stack V, heap H the expression e evaluates to v, and engenders a new heap H'. We write the representative rules, since the rest are derived in the obvious way from the garbage collection semantics.

$$\frac{l \notin dom(H) \qquad v = \langle V(x_1), V(x_2)\rangle \qquad H' = H\{l \mapsto v\}}{V, H \vdash \texttt{cons}(x_1; x_2) \Downarrow l, H'} (\mathrm{S}_{18})$$

$$\frac{\begin{array}{c} V(x) = l \qquad H(l) = \langle v_h, v_t\rangle \\ V' \subseteq V \qquad dom(V') = FV(e_2) \setminus \{x_h, x_t\} \qquad V'' = V'[x_h \mapsto v_h, x_t \mapsto v_t] \qquad V'', H \vdash e_2 \Downarrow v, H' \end{array}}{V, H \vdash \texttt{case}\, x\, \{\texttt{nil} \hookrightarrow e_1 \mid \texttt{cons}(x_h; x_t) \hookrightarrow e_2\} \Downarrow v, H'} (\mathrm{S}_{19})$$

$$\frac{\begin{array}{c} V = V_1 \sqcup V_2 \qquad dom(V_1) = FV(e_1) \\ dom(V_2) = FV(\texttt{lam}(x : \tau.e_2)) \qquad V_1, H \vdash e_1 \Downarrow v_1, H_1 \qquad V_2' = V_2[x \mapsto v_1] \qquad V_2', H_1 \vdash e_2 \Downarrow v_2, H_2 \end{array}}{V, H \vdash \texttt{let}(e_1; x : \tau.e_2) \Downarrow v_2, H_2} (\mathrm{S}_{20})$$

$$\frac{\begin{array}{c} V = V'[x \mapsto v'] \qquad L \cap dom(H) = \emptyset \\ |L| = dom(reach_H(v')) \qquad H', v'' = copy(H, L, v') \qquad V'[x_1 \mapsto v', x_2 \mapsto v''], H' \vdash e \Downarrow v, H'' \end{array}}{V, H \vdash \texttt{shareCopy}\; x\; \texttt{as}\; x_1, x_2\; \texttt{in}\; e \Downarrow v, H''} (\mathrm{S}_{21})$$

5 Well Defined Environments

In order to define the potential for first-order types, we need a notion of well-define environments, one that relates heap values to semantic values of a type. We first give a denotational semantics for the first-order types:

$$\begin{gathered}
() \in [\![\texttt{unit}]\!] \\
\bot \in [\![\texttt{bool}]\!] \\
\top \in [\![\texttt{bool}]\!] \\
0 \in [\![\texttt{nat}]\!]
\end{gathered}$$

$$\begin{aligned}
n+1 &\in [\![\texttt{nat}]\!] \text{ if } n \in [\![\texttt{nat}]\!] \\
\langle a_1, a_2\rangle &\in [\![A_1 \times A_2]\!] \text{ if } a_1 \in [\![A_1]\!] \text{ and } a_2 \in [\![A_2]\!] \\
[] &\in [\![L(A)]\!] \\
\pi(a, l) &\in [\![L(A)]\!] \text{ if } a \in [\![A]\!] \text{ and } l \in [\![L(A)]\!]
\end{aligned}$$

Where semantic set for each type is the least set such that the above holds. Note $\pi(x, y)$ is the usual set-theoretic pairing function, and write $[a_1, ..., a_n]$ for $\pi(a_1, ..., \pi(a_n, []))$. We also refer to the elements of a semantic set as structures.

Now 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: under heap H, heap value v defines the semantic value $a \in [\![A]\!]$.

$$\frac{n \in \mathbb{Z}}{H \vDash n \mapsto n : \texttt{nat}}\text{(V:ConstI)} \qquad \frac{}{H \vDash \texttt{Null} \mapsto n : \texttt{unit}}\text{(V:ConstI)} \qquad \frac{A \in \mathsf{BType}}{H \vDash \texttt{Null} \mapsto n : L(A)}\text{(V:Nil)}$$

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

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

6 Linear Potential

We introduce linear potential for structures corresponding to the base types. 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:

$$\begin{aligned}
&\Phi_H(a : A) = 0 && A \in \{\texttt{unit}, \texttt{bool}, \texttt{nat}\} \\
&\Phi_H(\langle a_1, a_2\rangle : A_1 \times A_2) = \Phi_H(a_1 : A_1) + \Phi_H(a_2 : A_2) \\
&\Phi_H([a_1, ...a_n] : L^p(A)) = p \cdot n + \sum_{1 \le i \le n} \Phi_H(a_i : A)
\end{aligned}$$

7 Linear Garbage Collection Type Rules

The type system consists of rules of the form $\boxed{\Sigma; \Gamma \,\Big|\!\frac{q}{q'}\; e : A}$, read as under signature Σ, context Γ, e has type A starting with q units of constant potential and ending with q' units.

The linear version of the type system takes into account of garbaged collected cells by returning potential locally in a match construct. Since we are interested in the number of heap cells, there is an implicit side condition which ensures all constants are assumed to be nonnegative.

The type system is based on the affine type system in **??**. The only major difference is the rule for L:MatL, in which an additional unit of potential is returned. This reflects the fact (Lemma 1.7) 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.

$$\frac{n \in \mathbb{Z}}{\Sigma; \emptyset \Big|^{q}_{q}\, n : \mathtt{nat}}\,(\text{L:ConstI}) \qquad \frac{}{\Sigma; \emptyset \Big|^{q}_{q}\, () : \mathtt{unit}}\,(\text{L:ConstU}) \qquad \frac{}{\Sigma; \emptyset \Big|^{q}_{q}\, \mathtt{T} : \mathtt{bool}}\,(\text{L:ConstT})$$

$$\frac{}{\Sigma; \emptyset \Big|^{q}_{q}\, \mathtt{F} : \mathtt{bool}}\,(\text{L:ConstF}) \qquad \frac{}{\Sigma; x : B \Big|^{q}_{q}\, x : B}\,(\text{L:Var}) \qquad \frac{\Sigma(f) = A \xmapsto{q/q'} B}{\Sigma; x : A \Big|^{q}_{q'}\, f(x) : B}$$

$$\frac{\Sigma; \Gamma \Big|^{q}_{q'}\, e_t : B \qquad \Sigma; \Gamma \Big|^{q}_{q'}\, e_f : B}{\Sigma; \Gamma, x : \mathtt{bool} \Big|^{q}_{q'}\, \mathtt{if}\, x\, \mathtt{then}\, e_t\, \mathtt{else}\, e_f : B}\,(\text{L:Cond}) \qquad \frac{}{\Sigma; x_1 : A_1, x_2 : A_2 \Big|^{q}_{q}\, \langle x_1, x_2 \rangle : A_1 \times A_2}\,(\text{L:Pair})$$

$$\frac{\Sigma; \Gamma, x_1 : A_1, x_2 : A_2 \Big|^{q}_{q'}\, e : B}{\Sigma; \Gamma, x : (A_1, A_2) \Big|^{q}_{q'}\, \mathtt{case}\ x\ \{(x_1; x_2) \hookrightarrow e\} : B}\,(\text{L:MatP}) \qquad \frac{}{\Sigma; \emptyset \Big|^{q}_{q}\, \mathtt{nil} : L^{p}(A)}\,(\text{L:Nil})$$

$$\frac{}{\Sigma; x_h : A, x_t : L^{p}(A) \Big|^{q+p+1}_{q}\, \mathtt{cons}(x_h; x_t) : L^{p}(A)}\,(\text{L:Cons})$$

$$\frac{\Sigma; \Gamma \Big|^{q}_{q'}\, e_1 : B \qquad \Sigma; \Gamma, x_h : A, x_t : L^{p}(A) \Big|^{q+p+1}_{q'}\, e_2 : B}{\Sigma; \Gamma, x : L^{p}(A) \Big|^{q}_{q'}\, \mathtt{case}\, x\, \{\mathtt{nil} \hookrightarrow e_1 \mid \mathtt{cons}(x_h; x_t) \hookrightarrow e_2\} : B}\,(\text{L:MatL})$$

$$\frac{\Sigma; \Gamma_1 \Big|^{q}_{p}\, e_1 : A \qquad \Sigma; \Gamma_2, x : A \Big|^{p}_{q'}\, e_2 : B}{\Sigma; \Gamma_1, \Gamma_2 \Big|^{q}_{q'}\, \mathtt{let}(e_1; x : \tau.e_2) : B}\,(\text{L:Let}) \qquad \frac{\Sigma; \Gamma \Big|^{q}_{q'}\, e : B}{\Sigma; \Gamma, x : A \Big|^{q}_{q'}\, \mathtt{drop}(x; e) : B}\,(\text{L:Drop})$$

$$\frac{A \curlyvee A_1, A_2, 1 \qquad \Sigma; \Gamma, x_1 : A_1, x_2 : A_2 \Big|^{q}_{q'}\, e : B}{\Sigma; \Gamma, x : A \Big|^{q}_{q'}\, \mathtt{shareCopy}\ x\ \mathtt{as}\ x_1, x_2\ \mathtt{in}\ e : B}\,(\text{L:ShareCopy})$$

Where $A \curlyvee A_1, A_2, n$ is the sharing relation defined as:

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

Now if we take $\dagger : L^{p}(A) \mapsto L(A)$ as the map that erases resource annotations, we obtain a simpler typing judgement $\boxed{\Sigma^{\dagger}; \Gamma^{\dagger} \vdash e : B^{\dagger}}$.

8 Soundness for Linear GC

We state and prove the soundness of FO^{gc} for *well-formed* computations. We need the following definitions to state well-formedness:

Definition 8.1. (Non-aliasing 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 non-aliasing given that:

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

Denote this by $\mathsf{no\_alias}(V, H)$.

For a stack V and a heap H, whenever $\mathsf{no\_alias}(V, H)$ holds, visually, one can think of the situation as the following: the induced graph of heap H with variables on the stack as additional leaf nodes is a forest: a disjoint union of directed trees; consequently, there is at most one path from a live variable on the stack V to a location in H by following the pointers.

Definition 8.2 (Stability)**.** Given heaps H, H', a set of locations is *stable* if $\forall l \in R.\ H(l) = H'(l)$. Denote this by $\mathsf{stable}(R, H, H')$.

Definition 8.3 (Well-formed 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 *well-formed computation* given the following:

1. $dom(V) = FV(e)$
2. $\mathsf{no\_alias}(V, H)$
3. $\mathsf{disjoint}(\{R, F, locs_{V,H}(e)\})$

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

Lemma 1.1. *If* $\Sigma; \Gamma \left|\frac{q}{q'}\right. e : B$*, then* $\Sigma^\dagger; \Gamma^\dagger \vdash e : B^\dagger$.

Lemma 1.2. *If* $\Sigma; \Gamma \left|\frac{q}{q'}\right. e : B$*, then* $\mathsf{set}(FV^\star(e))$ *and* $dom(\Gamma) = FV(e)$.

Proof. Induction on the typing judgement. ☐

Lemma 1.3. *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 *cst*. ☐

Corollary 1.3.1. *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 1.3. ☐

Lemma 1.4. *Let* $H \vDash v \mapsto a : A$*. If* $\mathtt{stack}(A)$*, then* $\Phi_H(v : A) = 0$.

Proof. Induction on $H \vDash v \mapsto a : A$. ☐

Lemma 1.5 (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 1.6 (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 1.7 (main lemma)**.** *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* $\mathsf{wfc}(V, H, R, F, e)$*, we have the follwoing:*

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

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

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

To formally state the soundness theorem (and later the equivalence of free and copy semantics), we need the notion of context equivalence. Here we define it for contexts, which consisting of only the stack and heap. Later, we extend it the the full configuration. First, define *value* equivalence:

Definition 8.4 (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 8.5 (Context Equivalence)**.** Two simple contexts $(V_1, H_1), (V_2, H_2)$ are equivalent (with the presupposition that both are well-formed contexts) iff $dom(V_1) = dom(V_2)$ and for all $x \in dom(V_1)$, $V_1(x) \sim^{H_1}_{H_2} V_2(x)$. 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.

Task 1.8 (Soundness)**.** *let* $H \vDash V : \Gamma$*,* $\Sigma; \Gamma \left|\frac{q}{q'}\right. e : B$*,* $V, H \ \vdash e \Downarrow v, H'$*, and* $H' \vDash v \mapsto a : A$*. Then* $\forall C \in \mathbb{Q}^+$ *and* $\forall F, R \subseteq \mathsf{Loc}$*, given the following (existence lemma):*

1. $\mathsf{wfc}(V, H, R, F, e)$

2. $|F| \geq \Phi_{V,H}(\Gamma) + q + C$

then there exists a context (W, Y)*, a value* w*, and a freelist* F' *s.t.*

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

2. $W, Y, R, F \vdash e \Downarrow w, Y', F'$

3. $v \sim^{H'}_{Y'} w$

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

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

Case 1: E:Var

$V, H \ \vdash x \Downarrow V(x), H$ (admissibility)

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

Let $C \in \mathbb{Q}^+, F, R \subseteq \mathsf{Loc}$ be arb.

Suppose this eval-config is well-formed, and further, $|F| \geq \Phi_{V,H}(x : B) + q + C$

Let $F' = F$. Then

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

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

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

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

$|F| - |F'| = |F| - |F|$ (ad.)

$= 0$

$\Phi_{V,H}(\Gamma) + q - (\Phi_{H'}(v : B) + q') = \Phi_{V,H}(\emptyset) + q - (\Phi_H(v : int) + q)$ (ad.)

$= 0$ (def of $\Phi_{V,H}$)

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

Case 4: E:App

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

$V, H \vdash e \Downarrow v_2, H_2$ (case)

$V, H \vdash e_1 \Downarrow v_1, H_1$ (ad.)

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

$H \vDash V_1 : \Gamma_1$ (def of W.D.E)

Let $C \in \mathbb{Q}^+, F, R \subseteq \mathsf{Loc}$ be arb.

Suppose $dom(V) = FV(e), \mathsf{no\_alias}(V, H), \mathsf{disjoint}(\{R, F, locs_{V,H}(e)\}),$ and $|F| \geq \Phi_{V,H}(\Gamma) + q + C$

NTF F' s.t.

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

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

Let $R' = R \cup locs_{V,H}(\texttt{lam}(x : \tau.e_2))$

$\mathsf{disjoint}(\{R', F, locs_{V,H}(e_1)\})$ (Similar to case in Lemma 1.7)

Instantiate IH with $C = C + \Phi_{V_2,H}(\Gamma_2), F = F, R = R',$ we get existence lemma on J_1 :

NTS (1) - (4) to instantiate existence lemma on J_1

(1) $dom(V_1) = FV(e_1)$

(2) $\mathsf{no\_alias}(V_1, H)$

(3) $\mathsf{disjoint}(\{R, F, locs_{V,H}(e)\})$ ((1) - (3) all verbatim as in Lemma 1.7)

(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 lemma on $J_1,$ we get F'' s.t.

1.$V, H, R', F \vdash e_1 \Downarrow v_1, H_1, F''$ and

2.$|F''| \geq \Phi_{H_1}(v_1 : A) + p + C + \Phi_{V_2,H_1}(\Gamma_2)$

For the second premise:

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

$H_1 \vDash v_1 : A$ and (Theorem 3.3.4)

$H_1 \vDash V : \Gamma_2$ (???)

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

$V', H_1 \vdash e_2 \Downarrow v_2, H_2$ (ad.)

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

Instantiate IH with $C = C, F = F'' \cup g, R = R,$ we get existence lemma on J_2 :

NTS (1) - (4) to instantiate existence lemma on J_1

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

(2) $\mathsf{no\_alias}(V_2', H_1)$

(3) $\mathsf{disjoint}(\{R, F'' \cup g, locs_{V_2', H_1}(e_2)\})$ ((1) - (3) all verbatim as in Lemma 1.7)

(4) $|F'' \cup g| \geq \Phi_{V_2', H_1}(\Gamma_2, x : (A-1)) + p + C$

STS $|F'' \cup g| \geq \Phi_{V_2, H_1}(\Gamma_2) + \Phi_{H_1}(v_1 : (A-1)) + p + C$

$|F'' \cup g| \geq \|V_1\|_H + |F| - \|v_1\|_{H_1}$ (conservation lemma)

$\geq \Phi_{V,H}(\Gamma) + q + C + \|V_1\|_H - \|v_1\|_{H_1}$ ($|F| \geq \Phi_H(V) + q + C$)

STS $\Phi_{V_1,H}(\Gamma_1) + q + C\ \|V_1\|_H - \|v_1\|_{H_1} \geq \Phi_{H_1}(v_1 : (A-1)) + p + C$

$\Phi_{V_1,H}(\Gamma_1) \geq \Phi_{H_1}(v_1 : (A-1))$ (lemma about cf typing)

STS $\|V_1\|_H - \|v_1\|_{H_1} + q \geq p$ (done by aux lemma)

Instantiating existence lemma on J_2, we get $F^{(3)}$ s.t.

1.$V_2', H_1, R, F'' \cup g \vdash e_2 \Downarrow v_2, H_2, F^{(3)}$

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

Take $F' = F^{(3)}$

$V, H, R, F \vdash e \Downarrow v_2, H_2, F'$ and (E:Let)

$|F'| \geq \Phi_{H_2}(v_2 : B) + q' + C$ (from IH)

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

$V, H \vdash \mathtt{cons}(x_1; x_2) \Downarrow l, H'$ (case)

Let $C \in \mathbb{Q}^+, F, R \subseteq \mathsf{Loc}$ be arb.

Suppose $dom(V) = FV(e), \mathsf{no\_alias}(V, H), \mathsf{disjoint}(\{R, F, locs_{V,H}(e)\}), |F| \geq \Phi_{V,H}(\Gamma) + q + C$

NTF F' s.t.

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

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

Let $F' = F$

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

Case 13: E:MatCons

$V(x) = (l, \mathtt{alive})$ (ad.)

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

$V'', H \ \vdash e_2 \Downarrow v, H'$ (ad.)

Let $C \in \mathbb{Q}^+, F, R \subseteq \mathsf{Loc}$ be arb.

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

$H'' \vDash v_h : A,\ H'' \vDash v_t : L^p(A)$ (ad.)

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

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

Suppose $\mathsf{no\_alias}(V,H), \mathsf{disjoint}(\{R,F,locs_{V,H}(e)\}),$ and $|F| \geq \Phi_{V,H}(\Gamma)+q+C$

NTF F' s.t.

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

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

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

We want to g nonempty, in particular, that $l \in g$

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

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

Then $l \in reach_H(\overline{V}''(x'))$ for some $x' \neq x$

$x' \in \{x_h, x_t\}$ $\quad$ (since $reach_H(\overline{V}(x')) \cap reach_H(\bar{(}Vx)) = \emptyset$ from $\mathsf{no\_alias}(V,H)$)

WLOG let $x' = x_h$

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

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

Hence $l \in g$

Next, we have $\mathsf{no\_alias}(V'',H)$ and $\mathsf{disjoint}(\{R,F\cup g,locs_{V'',H}(e_2)\})$ $\quad$ (similar to case in Lemma 1.2)

By IH with $C'=C, F''=F\cup g$ and the above conditions, we have: $F^{(3)}$ s.t.

1.$V'',H,R,F\cup g \vdash e_2 \Downarrow v,H',F^{(3)}$

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

Where we also verify the precondition that $|F''| \geq \Phi_{V'',H}(\Gamma',x_h:A,x_t:L^p(A))+q+p+1+C'$:

$$\begin{aligned}
|F''| &= |F \cup g| \\
&= |F|+|g| && (F \text{ and } g \text{ disjoint})\\
&\geq \Phi_{V,H}(\Gamma)+q+C+|g| && (\text{Sp.})\\
&= \Phi_{V,H}(\Gamma',x_h:A,x_t:L^p(A))+p+q+C+|g| && (\text{Lemma 4.1.1})\\
&= \Phi_{V,H}(\Gamma',x_h:A,x_t:L^p(A))+p+q+C+1 && (g \text{ nonempty})
\end{aligned}$$

Now take $F' = F^{(3)}$

$V,H,R,F \ \vdash e \Downarrow v,H',F'$ $\quad$ (E:MatCons)

$|F'| \geq \Phi_{H'}(v:B)+q'+C$ $\quad$ (From the IH)

Case 14: E:Share

$V,H \vdash e \Downarrow v,H''$ $\quad$ (case)

$V'[x_1 \mapsto v', x_2 \mapsto v''],H' \vdash e' \Downarrow v,H''$ $\quad$ (ad)

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

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

$\Sigma;\Gamma,x_1:A_1,x_2:A_2 \left|\frac{q}{q'}\right. e:B$ $\quad$ (ad.)

Let $C \in \mathbb{Q}^+, F,R \subseteq \mathsf{Loc}$ be arb.

Suppose $\mathsf{no\_alias}(V,H), \mathsf{disjoint}(\{R,F,locs_{V,H}(e)\}),$ and $|F| \geq \Phi_{V,H}(\Gamma,x:A)+q+C$

NTF F'' s.t.

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

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

We need to show the freelist is sufficient for the subsequent computation to invoke the IH:

Instantiate with $C, F\setminus L, \text{and} R$

STS $|F \setminus L| \geq \Phi_{V_2,H'}(\Gamma, x_1 : A_1, x_2 : A_2) + q + C$

$\iff |F| - |L| \geq \Phi_{V_2,H'}(\Gamma) + \Phi_{V_2,H'}(x_1 : A_1) + \Phi_{V_2,H'}(x_2 : A_2) + q + C$

$\iff |F| \geq \Phi_{V_2,H'}(\Gamma) + \Phi_{V_2,H'}(x_1 : A_1) + \Phi_{V_2,H'}(x_2 : A_2) + \left\|v'\right\|_H + q + C$

$\iff |F| \geq \Phi_{V_2,H'}(\Gamma) + \Phi_{V,H}(x : A) + q + C$ (definition of sharing relation)

$\iff |F| \geq \Phi_{V,H}(\Gamma, x : A) + q + C$ (stability of copying)

done from assumption

By IH, we get F'' fulfilling the previous two points for the case.

□

9 Copy-free garbage collection semantics

Consider the GC semantics (from now on copy semantics) above, with the share rule replaced with the following:

$$\frac{V = V'[x \mapsto v'] \qquad V'[x_1 \mapsto v', x_2 \mapsto v'], H', R, F \ \vdash e \Downarrow v, H'', F'}{V, H, R, F \ \vdash \texttt{share}\ x\ \texttt{as}\ x_1, x_2\ \texttt{in}\ e \Downarrow v, H'', F'}\text{(F:Share)}$$

Call this new semantics free semantics for copy-free (all rules are renamed to F:_ for free). It is easy to see that any terminating compuation in copy has a corresponding one in free that can be instantiated with an equal or smaller freelist. Before formalizing this idea, we extend context equivalence to a preorder on configurations:

Definition 9.1 (Compute Potential)**.** Given a configuration $\mathcal{C} = V, H, R, F$, define the *compute potential* of the configuration κ as $|F| + |H \setminus F|$.

Lemma 1.9 (Compute Invariance (copy))**.** *If* $V, H, R, F \vdash^{\mathsf{copy}} e \Downarrow v, H', F'$*, then* $\kappa(H, F) = \kappa(H', F')$*.*

For each pair of structurally equivalent values $v \sim^H_{H'} v'$, there is an induced bijection between the reach locations of each respective value that serves as the evidence for their structural equivalence. We give the representative rules:

$$\frac{m_1 : v_1 \sim^H_{H'} v_1' \qquad m_2 : v_2 \sim^H_{H'} v_2'}{m_1 \uplus m_2 : \langle v_1, v_2 \rangle \sim^H_{H'} \langle v_1', v_2' \rangle}\text{(M:Pair)} \qquad \frac{m : H(l) \sim^H_{H'} H(l')}{m \uplus \{l \to l'\} : l \sim^H_{H'} l'}\text{(M:Loc)}$$

Extending to a context, we write $L : V_1, H_1 \sim V_2, H_2$ if for all $x \in \mathit{dom}(V_1)$, $L(x) : V_1(x) \sim^{H_1}_{H_2} V_2(x)$.

Definition 9.2. A configuration (V, H, R, F) is well-formed if $\mathit{dom}(H) \subseteq \mathit{reach}_H(V) \cup R \cup F$.

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

We define the induced mapping $\gamma : \mathit{dom}(H_1) \setminus F_1 \to \mathcal{P}(\mathit{dom}(H_2) \setminus F_2)$ between $\mathcal{C}_1, \mathcal{C}_2$ from L: $\gamma(l) = (\biguplus_{x \in \mathit{dom}(V_1)} L(x))(l)$. Note that by construction, for any l, $\gamma(l)$ is nonempty, and thus we can choose an arbitrary $l' \in \gamma(l)$ to be the representation for that part; call this $\mathit{rep}(l)$.

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 $x \in A$, $f(x) \neq \emptyset$.

A simple corollary is the fact that if V_2, H_2 is a linear context (e.g. $\mathsf{no\_alias}(V_2, H_2)$ holds), then $|\gamma(l)| = |(\mathit{reach}_{H_1}(V_1))(l)|$, where $\mathit{reach}_{H_1}(V_1) = \biguplus_{x \in \mathit{dom}(V)} \mathit{reach}_{H_1}(x)$. 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.

For an evaluation $\mathcal{C} = (V, H, R, F) \vdash e \Downarrow v, H', F'$, denote its garbage collection as $collect(C \vdash v, H', F') = \{l \in H' \mid l \notin F' \cup R \cup reach'_H(v)\}$.

Definition 9.3. 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. There is a mapping $\eta : (dom(H_1) \setminus F_1) \to dom(V_1) \to \mathcal{P}(dom(H_2) \setminus F_2)$ such that for all $l \notin R_1$, $\eta(l)(x) = reach_{H_2}(x) \cap \gamma(l)$. Futhermore, for all l and x, $|\eta(l)(x)| = reach_{H_1}(V_1(x))(l)$.
4. $R_1 \subseteq dom(H_1) \setminus F_1$ and $R_2 = \bigcup \gamma(R_1)$
5. $|F_1| = |F_2| + | \oslash (\gamma)|$, where $\oslash(\gamma) = \bigcup_{P \in ec(\gamma)} P \setminus (rep(P))$

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

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

Lemma 1.10. *Let $(\mathcal{C}_2, e)$ be a linear computation. Given that $\mathcal{C}_2 \vdash^{\mathsf{copy}} e \Downarrow v, H', F'$, and $H' \vDash v \mapsto a : A$, 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 $l \in dom(\gamma')$,* $|\gamma'(l)| = |reach_{Y_1}(w_1)(l)| + S(l) + |\gamma'(l) \cap collect(\mathcal{C}_2 \vdash^{\mathsf{copy}} e \Downarrow v, H', F')|$
4. $|reach_{H_1}(v) \cap \gamma'(l)| = reach_{Y_1}(w)(l)$
5. $R_2 = \bigcup \gamma'(R_1)$
6. $|M'| = |F'| + | \oslash (\gamma')|$

For a configuration $\mathcal{C} = (V, H, R, F)$, denote the current garbage w.r.t a set of root variables $X \subseteq dom(V)$ as $clean(C, X) = \{l \in H \mid l \notin F \cup R \cup reach_H(X)\}$. Some auxiliary lemmas:

Lemma 1.11. *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:*

1. *for all $l \in dom(H_1) \setminus F_1$, $X \subseteq dom(V)$, $\gamma(l) \subseteq clean(\mathcal{C}_2, X)$ implies that $l \in clean(\mathcal{C}_1, X)$.*

In particular, this means that we can (somewhat obviously), execute a computation using the free semantics if the computation suceeded with the copy semantics.

Proof. Induction on the evaluation judgement.

Case 1: E:Var

$$V, H, R, F \vdash^{\mathsf{copy}} e \Downarrow V(x), H, F \qquad \text{(case)}$$

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

We need to show a triple that satisfies the 3 post-conditions.

Take $(w, Y', M') = (W(x), Y, M)$

(1) $W, Y, S, M \vdash^{\mathsf{free}} e \Downarrow w, Y, M$ (F:Var)

(2) $V(x) \sim^{H}_{Y'} w$ (Definition of $\sim$)

(3) $\kappa(Y, M) = \kappa(H, F)$ (Definition of cp. ext.)

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

Case 4: E:App

Case 5: E:CondT

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

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

Let $W' = W \restriction_{V'}$

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)

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

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

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 1.11)

Contradiction since assumed $l \notin j$

Now, let $l \in dom(\gamma')$

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

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

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

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

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

$|\gamma'(l)| = |\gamma(l)| = S(l)$

(3) NTF $\eta' : (dom(Y) \setminus (M \sqcup j)) \to dom(V') \to \mathcal{P}(dom(H) \setminus (F \cup g))$

$\eta'(l)(x) = \eta(l)(x) \setminus g$

Let $l \notin$S

First, NTS $\eta'(l)$ is a partition:

Let $x, x' \in dom(V')$

$\eta'(l)(x) \cap \eta'(l)(x') = \emptyset$ $\quad$ ($\eta(l)$ is partition)

$\eta'(l)(dom(V')) = \eta(l)(dom(V'))$

$=$

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

NTS $R = \bigcup \gamma'(S)$

$\forall l \in S, \gamma'(l) = \gamma(l)$

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

$= \gamma(l) \setminus g$

$= \gamma(l)$ $\quad$ ($\gamma(l) \subseteq R$ and $R \cap g = \emptyset$)

(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 show that $g = (\bigsqcup_{l \in j} \gamma(l)) \sqcup L$ for some L

TODO: Show this

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: $f_1 : j \oplus D_1 \to \bigsqcup_{C \in \mathcal{C}_2} C, f_2 : D_2 \to L \sqcup \oslash(\gamma')$

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

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:

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

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

$= |ec(\gamma')|$

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

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

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

$l \in H \setminus (F \sqcup g)$ (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, $|j| + |\oslash(\gamma)| = |g| + |\oslash(\gamma')|$

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

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

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

(2) γ'' is a counting partition for $reach_{Y'}(w) \uplus R$

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

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

Case 6: E:CondF Similar to E:CondT

Case 7: E:Let

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

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

Case 13: E:MatCons

Case 14: E:Share

$V, H, R, F \vdash^{\mathsf{copy}} \texttt{shareCopy}\; x \;\texttt{as}\; x_1, x_2 \;\texttt{in}\; e \Downarrow v, H'', F'$ (case)

Suppose $H'' \vDash v \mapsto a : A$. We need to show a configuration and a triple that satisfies the 3 post-conditions.

By IH, we have (w, K, M) such that

(1) $V_2, H', R, F \setminus L \vdash^{\mathsf{free}} e \Downarrow w, K, M$ (F:Var)

(2) $K \vDash w \mapsto a : A$

(3) $|M| \geq |F'|$

$\square$

References

[1] Jan Hoffmann, Ankush Das, and Shu-Chun Weng. Towards automatic resource bound analysis for ocaml. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 359–373, New York, NY, USA, 2017. ACM.

[2] 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.

[3] 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.

[4] 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.

10 Examples

For brevity, we write the following examples using an extended syntax which is not in let-normal form, but which is equivalent to the restricted syntax in expressiveness and semantics. The program and signature is populated with top level let declarations.

append:

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

$P = [\mathsf{append} \mapsto e_{\mathsf{append}}]$
$\Sigma = [\mathsf{append} \xrightarrow{q/q} L^p(A) \times L^p(A) \to \text{Ł}^p(A)]$

Type derivation:

$$\dfrac{\dfrac{}{\Sigma; l_2 : L^p(A) \vdash^{q}_{q} l_2 : L^p(A)}\text{L:Var} \qquad \dfrac{\dfrac{append : \xrightarrow{q/q} L^p(A) \times L^p(A) \to \text{Ł}^p(A) \in \Sigma}{\Sigma; L_2 : L^p(A), xs : L^p(A) \vdash^{q+p+1}_{q+p+1} append(xs, l_2) : L^p(A)}\text{L:App} \qquad \dfrac{}{\Sigma; x : A, r : L^p(A) \vdash^{q+p+1}_{q} x :: r : L^p(A)}\text{L:Cons}}{\Sigma; l_2 : L^p(A), x : A, xs : L^p(A) \vdash^{q+p+1}_{q} \texttt{let } r = append(xs, l_2) \texttt{ in } x :: r : L^p(A)}\text{L:Let}}{\Sigma; l_1 : L^p(A), l_2 : L^p(A) \vdash^{q}_{q} \mathsf{case}\, l_1\, \{\texttt{nil} \hookrightarrow l_2 \mid \mathsf{cons}(x; xs) \hookrightarrow \texttt{let } r = append(xs, l_2) \texttt{ in } x :: r\} : L^p(A)}\text{L:MatL}$$

This can be read as **append** takes two lists, each with potential p per element, and a constant potential q, and returns a list with potential p per element and constant potential q. This bound is tight and reflects the fact that append is constructing the concatenation "in place" by collecting the cells in l_1. Thus, append induces no overhead heap cells in addition to its arguments.

Now we show that quicksort is also has no overhead. First, the partition:

```
let rec partition (p, l) =
  match l with
    | [] -> ([],[])
    | x::xs ->
      let (l1,l2) = partition xs in
      let r = x < p in
      if r then
        (x::l1,l2)
      else
        (l1,x::l2)
```

$P = [partition \mapsto e_{partition}]$
$\Sigma = [partition \mapsto \texttt{nat} \times L^p(\texttt{nat}) \to L^p(\texttt{nat}) \times L^p(\texttt{nat})]$
The type derivation for the Cons branch:

$$\dfrac{}{\Sigma; p : \texttt{nat}, x : \texttt{nat}, xs : L^p(\texttt{nat}) \vdash \texttt{let } r = \ \texttt{in } x :: r}$$

```
let rec quicksort l =
match l with
  | [] -> []
  | x::xs ->
      let ys, zs = partition (x,xs) in
let l1 = quicksort ys in
let l2 = quicksort zs in
let r = x :: l2 in
      append (l1, r)
```

map:

```
let rec map (f,l) =
match l with
| [] -> []
| x::xs -> let r = f x in r :: map (f, xs)
```

$P = [\mathsf{map} \mapsto e_{\mathsf{map}}]$
$\Sigma = [\mathsf{map} \mapsto A \to B \times L^0(A) \to \text{Ł}^0(B)]$

Type derivation:

11 Implementation

12 Experimentation

13 Future Work