Automatic Resource Bound Analysis for Functional Programs with Garbage Collection

Serguei A. Mokhov[1]*, Geoff Sutcliffe[2]†, and Andrei Voronkov[345]‡

[1] Concordia University, Montreal, Quebec, Canada
mokhov@cse.concordia.ca
[2] University of Miami, Miami, Florida, U.S.A.
geoff@cs.miami.edu
[3] University of Manchester, Manchester, U.K.
andrei@voronkov.com
[4] Chalmers University of Technology, Gothenburg, Sweden
[5] EasyChair

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.

Contents

*Designed and implemented the class style
†Did numerous tests and provided a lot of suggestions
‡Masterminded EasyChair and created versions 3.0–3.5 of the class style

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].

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 high watermark 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.

2 Notation

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 $rep(a)$.

3 First Order Programs

In this paper, in order to make clear the ideas and proofs involving our new garbage-collection cost semantics, we focus our attention to a first-order, strictly evaluated functional language. The reader can think of our target language to be a very simple subset of OCaml or SML. Furthermore, although we only demonstrate data types in the language with lists, our techniques extend to the expected algebraic data types definable in ML; in fact, our OCaml implementation leverages the OCaml compiler and supports type declarations in OCaml. Being first order, our language does not allow arbitrary local functional definitions. Instead, all functions are defined in a "global header" consisting of n mutually recursive blocks. 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 types of the language can be expressed as an arrow between zero-order (base) types:

BTypes	τ ::=		
	`nat`	`nat`	naturals
	`unit`	`unit`	unit
	`bool`	`bool`	boolean
	`prod`$(\tau_1;\tau_2)$	$\tau_1 \times \tau_2$	product
	`list`(τ)	$L(\tau)$	list
FTypes	ρ ::=		
	`arr`$(\tau_1;\tau_2)$	$\tau_1 \to \tau_2$	first order function

Next, we have the (abridged) expressions:

Exp	e ::=		
	x	x	variable
	$\texttt{nat}[n]$	$\overline{n}$	number
	$\texttt{unit}$	$()$	unit
	$\texttt{T}$	$\texttt{T}$	true
	$\texttt{F}$	$\texttt{F}$	false
	$\texttt{if}(x; e_1; e_2)$	$\texttt{if}\, x\, \texttt{then}\, e_1\, \texttt{else}\, e_2$	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

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. The one nonstandard construct is the $\texttt{share}\, x\, \texttt{as}\, x_1, x_2\, \texttt{in}\, e$, 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, which we introduce next. Although throughout the sections, semantic linearity will depend on the semantics at hand, everything we work with respects syntactic linearity. The sharing construct allows us to maintain syntactic linearity while working with multiple semantics that might or might not respect semantic linearity.

In line the previous works on space cost semantics, we employ a heap (also known as environment) which persistently binds locations to values (normalized terms). As usual, we derive the cost of a program from the number of heap locations number of heap locations used during execution. Locations is an infinite set of names for addressing the heap; we used natural numbers in the implementation:

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
	$\texttt{val}(l)$	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

For the rest of the paper, we assume the following shorthands:

$$\mathsf{Stack} \triangleq \{V \mid V : \mathsf{Var} \to \mathsf{Val}\}$$
$$\mathsf{Heap} \triangleq \{H \mid H : \mathsf{Loc} \to \mathsf{Val}\}$$

4 Reachability

Before we define the rules for the cost semantics, we relate the heap locations to expressions and value with the 3-place reachability relation $reach(H, v, L)$ on $\mathsf{Heap} \times \mathsf{Val} \times \wp(\mathsf{Loc})$. 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 $(+, +, -)$.

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

Notice that primitives and types with statically-known sizes are stack-allocated ($\mathtt{bool}, \mathtt{nat}, \tau_1 \times \tau_2$) and use no heap cells. The notion of reachability naturally lifts to expressions:

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

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

5 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 earlier version we tried (this particular one is adapted from [3]):

$$V, H, R \vdash e \Downarrow^s v, H$$

Which can be read as under stack $V \in \mathsf{Stack}$, heap $H \in msHeap$, 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:

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

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:

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

Which 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, 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. This mismatch between the dynamics and statics of language prevent us from proving the soundness of the analysis. Now, we present our novel cost semantics which solves this issue by combining "freelist" semantics from [2] with the above approach.

6 Garbage Collection Cost 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'. Compared to the previous section, the key ingredient we added is the freelist, which will serve as the set of available locations (for allocation of data constructors).

A *program* is 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 9). 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. Thus the evaluation judgement $\vdash$ is indexed by a signatures Σ, which is a mutually recursive block of global 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 tracing 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. As before, the continuation set R represents the set of locations required to compute the continuation *excluding* the current expression. Define the *root set* as the union of the locations in the continuation set R and the locations in the current expression e.

7 The Garbage Collection Cost Semantics copy

In order to establish the soundness of the our intended semantics free, we need an intermediary semantics which is *semantically linear*: copy. As mentioned in 3, this means that locations are linear, i.e. no location can be used twice in a program. Thus, 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 $copy(H, L, v, H', v')$ 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: $(+, +, +, -, -)$.

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

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

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. We now give the representative rules for copy. First, we have E:Var:

$$\frac{V(x) = v}{V, H, R, F \vdash x \Downarrow v, H, F} (\mathrm{S}_1)$$

Note that in contrast to the semantics in the previous section, looking up a variable incurs no cost. This ensures that we will be able prove the soundness of the type system. Next, we have E:Let:

$$\frac{\begin{array}{c} V_1 = V \restriction_{FV(e_1)} \\ R' = R \cup locs_{V_2,H}(\texttt{lam}(x : \tau.e_2)) \qquad V_1, H, R', F \vdash e_1 \Downarrow v_1, H_1, F_1 \qquad V_2' = (V[x \mapsto v_1]) \restriction_{FV(e_2)} \\ g = \{l \in H_1 \mid l \notin F_1 \cup R \cup locs_{V_2',H_1}(e_2)\} \qquad V_2', H_1, R, F_1 \cup g \vdash e_2 \Downarrow v_2, H_2, F_2 \end{array}}{V, H, R, F \vdash \texttt{let}(e_1; x : \tau.e_2) \Downarrow v_2, H_2, F_2} (\mathrm{S}_2)$$

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. We list the rest of the (abridged) rules below:

$$\frac{\begin{array}{c} V(x) = \texttt{T} \\ g = \{l \in H \mid l \notin F \cup R \cup locs_{V,H}(e_1)\} \qquad V' = V \restriction_{FV(e_1)} \qquad V', H, R, F \cup g \vdash e_1 \Downarrow v, H', F' \end{array}}{V, H, R, F \vdash \texttt{if}(x; e_1; e_2) \Downarrow v, H', F'} (\mathrm{S}_3)$$

$$\frac{\begin{array}{c} V(x) = \texttt{F} \\ g = \{l \in H \mid l \notin F \cup R \cup locs_{V,H}(e_2)\} \qquad V' = V \restriction_{FV(e_2)} \qquad V', H, R, F \cup g \vdash e_2 \Downarrow v, H', F' \end{array}}{V, H, R, F \vdash \texttt{if}(x; e_1; e_2) \Downarrow v, H', F'} (\mathrm{S}_4)$$

$$\frac{\begin{array}{c} V(x) = v' \\ P(f) = (y_f, e_f) \qquad g = \{l \in H \mid l \notin F \cup R \cup locs_{V,H}(e_f)\} \qquad [y_f \mapsto v'], H, R, F \cup g \vdash e_f \Downarrow v, H', F' \end{array}}{V, H, R, F \vdash f(x) \Downarrow v, H', F'} (\mathrm{S}_5)$$

$$\frac{}{V, H, R, F \vdash \texttt{nil} \Downarrow \texttt{val}(\texttt{Null}), H, F} (\mathrm{S}_6) \qquad \frac{v = \langle V(x_1), V(x_2) \rangle \qquad l \in F \qquad H' = H\{l \mapsto v\}}{V, H, R, F \vdash \texttt{cons}(x_1; x_2) \Downarrow l, H', F \setminus \{l\}} (\mathrm{S}_7)$$

$$\frac{\begin{array}{c} V(x) = v \\ L \subseteq F \qquad |L| = |dom(reach_H(v'))| \qquad H', v'' = copy(H, L, v') \qquad V_2 = (V[x_1 \mapsto v', x_2 \mapsto v'']) \restriction_{FV(e)} \\ F' = F \setminus L \qquad g = \{l \in H \mid l \notin F' \cup R \cup locs_{V_2,H}(e)\} \qquad V_2, H', R, F' \sqcup g \vdash e \Downarrow v, H'', F' \end{array}}{V, H, R, F \vdash \texttt{share}\ x\ \texttt{as}\ x_1, x_2\ \texttt{in}\ e \Downarrow v, H'', F'} (\mathrm{S}_8)$$

$$\frac{\begin{array}{c} V(x) = \texttt{Null} \\ V' = V \restriction_{FV(e_1)} \qquad g = \{l \in H \mid l \notin F \cup R \cup locs_{V,H}(e_1)\} \qquad V', H, R, F \cup g \vdash e_1 \Downarrow v, H', F' \end{array}}{V, H, R, F \vdash \texttt{case}\, x\, \{\texttt{nil} \hookrightarrow e_1 \mid \texttt{cons}(x_h; x_t) \hookrightarrow e_2\} \Downarrow v, H', F'} (\mathrm{S}_9)$$

$$\frac{\begin{array}{c} V(x) = l \qquad H(l) = \langle v_h, v_t \rangle \qquad V'' = (V[x_h \mapsto v_h, x_t \mapsto v_t]) \restriction_{FV(e_2)} \\ g = \{l \in H \mid l \notin F \cup R \cup locs_{V'',H}(e_2)\} \qquad V'', H, R, F \cup g \vdash e_2 \Downarrow v, H', F' \end{array}}{V, H, R, F \vdash \texttt{case}\, x\, \{\texttt{nil} \hookrightarrow e_1 \mid \texttt{cons}(x_h; x_t) \hookrightarrow e_2\} \Downarrow v, H', F'} (\mathrm{S}_{10})$$

In particular, note that since we don't allow local function definitions, no closures are created during evaluation. However, our implementation uses closures to implement the global block of function definitions. This is discussed in 13.

8 Preparing for Soundness

In this section, we define some auxiliary predicates and judgments for the main lemma in the following section.

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:

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

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 $a \in [\![A]\!]$.

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

$$\frac{}{H \vDash \texttt{T} \mapsto \top : \texttt{bool}}(\text{V:True}) \qquad \frac{}{H \vDash \texttt{F} \mapsto \bot : \texttt{bool}}(\text{V:False})$$

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

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

Given a stack V, we write $H \vDash V : \Gamma$ if $dom(V) = dom(\Gamma)$ and for every $x : A \in \Gamma$, $H \vDash V(x) \mapsto a : A$ for some $a \in [\![A]\!]$.

9 Linear Garbage Collection Type Rules

The type system FO^{gc} consists of rules of the form $\boxed{\Sigma; \Gamma \left|^{q}_{q'}\right. e : A}$, 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 1.6) 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.

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

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

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

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

$$\frac{\Sigma; \Gamma_1 \vert^{q}_{p} e_1 : A \qquad \Sigma; \Gamma_2, x : A \vert^{p}_{q'} e_2 : B}{\Sigma; \Gamma_1, \Gamma_2 \vert^{q}_{q'} \texttt{let}(e_1; x : \tau . e_2) : B}(\text{L:Let})$$

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

Define $\dagger : L^p(A) \mapsto L(A)$ as the map that erases resource annotations. We use this simplified typing judgment in proofs where the resource annotation is unnecessary. $\boxed{\Sigma^\dagger; \Gamma^\dagger \vdash e : B^\dagger}$.

In previous versions of RAML, the typing judgment is parametrized by a *cost metric* $m : \mathsf{res_c onst} \to \mathbb{Q}$, which assigns a rational constant to the set of control-flow points. Since the cost for all program construct is zero save for the cons data constructor, we elide the cost metric altogether. Although we defined the constructor to cost 1 heap location (as shown in L:Cons and L:MatL), it can be any constant as long as the introduction and elimination rules agree on that constant. Thus we can extend the type system to accurately track closure sizes and constructor which vary in size depending on the argument (more in 15).

Futhermore, note that given a judgment $\Sigma; \Gamma \vert^{q}_{q'} e : A$, there are infinitely many admissible judgments $\Sigma; \Gamma \vert^{q+k}_{q'+k} e : A$ for all $k \in \mathbb{N}$. The analysis module will export the constraints generated by the type system into an LP-solver which minimizes all coefficients, so that the smallest admissible q, q' will be used in the resulting symbolic bound.

10 Linearity of Copy Semantics

In this section, we establish a key lemma for the copy semantics: given a *linear* computation, the evaluation result is also linear. First, we characterize semantically linear contexts:

Definition 10.1. (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 non-aliasing given that:

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

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

Whenever $\mathsf{linear}(V, H)$ 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.

Definition 10.2 (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. $\mathsf{linear}(V, H)$
3. $\mathsf{disjoint}(\{R, F, locs_{V,H}(e)\})$

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

Definition 10.3 (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 1.1. *If* $\Sigma; \Gamma \left|\frac{q}{q'}\right. e : B$*, 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 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 $H \vDash v \mapsto a : A$. □

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

The following lemma one of main results of this paper: 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):

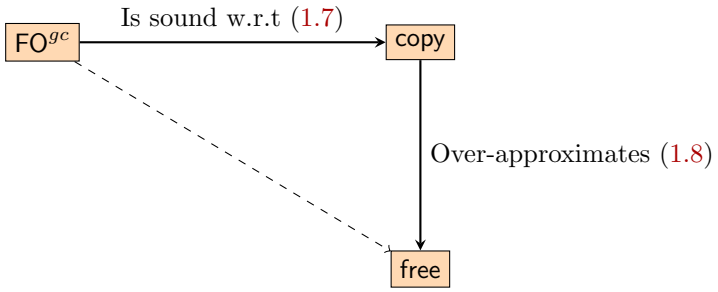
Lemma 1.6 (Linearity of copy)**.** *For all stacks* V *and heaps* H*, let* $V, H, R, F \vdash e \Downarrow v, H', F'$ *and* $\Sigma; \Gamma \vdash e : B$*. Then given that* $\mathsf{linear}(V, H, R, F, e)$*, we have the follwoing:*

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

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 makes the presentation clearer; furthermore, the linearity of copy is an interesting in itself, regardless of the type system.

11 Soundness of FO^{gc}

Recall that the semantics copy makes a deep copy of every variable that is used more than once in a program. 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. The semantics free (rules introduced in 12) is the version of the garbage collection semantics designed to capture this "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 figure below outlines the plan for proving the soundess of the type system w.r.t free:



Where the dashed arrow represents the intended result.

First, we show the soundness of FO^{gc} w.r.t copy. To formally state this theorem, we need to introduce an auxiliary semantics (call this the operational semantics) that does not use freelists or accounts for garbage collection. We use it to characterize expressions that normalizes to a value if given enough heap locations. This technique was also employed in [2] to establish the soundness of their type system. The operational semantics consists of judgments of the following form:

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

Where $\mathcal{S}$ is a *context*, consisting of a tuple in Stack $\times$ 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'. Below are the representative rules (the rest are derived in the obvious way from copy:

$$\frac{l \notin dom(H) \qquad v = \langle V(x_1), V(x_2)\rangle \qquad H' = H\{l \mapsto v\}}{V, H \vdash \mathtt{cons}(x_1; x_2) \Downarrow l, H'} (\mathrm{S}_{11})$$

$$\frac{V' \subseteq V \qquad dom(V') = FV(e_2) \setminus \{x_h, x_t\} \qquad \begin{array}{c} V(x) = l \qquad H(l) = \langle v_h, v_t\rangle \\ V'' = V'[x_h \mapsto v_h, x_t \mapsto v_t] \end{array} \qquad V'', H \vdash e_2 \Downarrow v, H'}{V, H \vdash \mathtt{case}\, x\, \{\mathtt{nil} \hookrightarrow e_1 \mid \mathtt{cons}(x_h; x_t) \hookrightarrow e_2\} \Downarrow v, H'} (\mathrm{S}_{12})$$

$$\frac{\begin{array}{c} V = V_1 \sqcup V_2 \qquad dom(V_1) = FV(e_1) \\ dom(V_2) = FV(\mathtt{lam}(x : \tau.e_2)) \qquad V_1, H \vdash e_1 \Downarrow v_1, H_1 \qquad V_2' = V_2[x \mapsto v_1] \end{array} \qquad V_2', H_1 \vdash e_2 \Downarrow v_2, H_2}{V, H \vdash \mathtt{let}(e_1; x : \tau.e_2) \Downarrow v_2, H_2} (\mathrm{S}_{13})$$

$$\frac{\begin{array}{c} V = V'[x \mapsto v'] \qquad L \cap dom(H) = \emptyset \\ |L| = dom(reach_H(v')) \qquad H', v'' = copy(H, L, v') \end{array} \qquad V'[x_1 \mapsto v', x_2 \mapsto v''], H' \vdash e \Downarrow v, H''}{V, H \vdash \mathtt{share}\ x\ \mathtt{as}\ x_1, x_2\ \mathtt{in}\ e \Downarrow v, H''} (\mathrm{S}_{14})$$

Next, 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 11.1 (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 $v_1 \sim_{H_2}^{H_1} v_2$.

Definition 11.2 (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 $x \in dom(V_1)$, $V_1(x) \sim_{H_2}^{H_1} V_2(x)$. 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. We are ready to state the soundness theorem now:

Task 1.7 (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* $\forall F, R \subseteq \mathsf{Loc}$, *if the following holds:*

1. $\mathsf{linear}(V, H, R, F, e)$
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. $v \sim_{Y'}^{H'} w$
4. $|F'| \geq \Phi_{H'}(v : B) + q' + C$

12 Garbage Collection Cost Semantics free

Consider copy 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 equally-sized or smaller freelist. Before formalizing this idea, we extend context equivalence to a preorder on configurations:

Definition 12.1. A configuration (V, H, R, F) is well-formed if $dom(H) \subseteq reach_H(V) \cup R \cup F$.

Definition 12.2. 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 $\mathcal{C}_1 \preceq \mathcal{C}_2$.

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

Lemma 1.8. *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 1.9. *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.

13 Implementation

The garbage collection cost semantics is implemented as an alternative evaluation module inside Resource Aware ML (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 operates on the resulting RAML syntax tree. First, we describe the normal evaluation module `Eval` that's parametrized by a cost metric. It exports the function `evaluate` with the following signature:

```
evaluate :
    ('a, unit) Expressions.expression ->
  (Metric.res_const -> float) list ->
  ((location * 'a heap) option) * (float * float) list
```

In order to implement the 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.

14 Evaluation

15 Future Work

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.