Arrays and References in Resource Aware ML*

Benjamin Lichtman[1] and Jan Hoffmann[2]

1 **Carnegie Mellon University, Pittsburgh, PA, USA**
`blichtma@alumni.cmu.edu`

2 **Carnegie Mellon University, Pittsburgh, PA, USA**
`jhoffmann@cmu.edu`

Abstract

This article introduces a technique to accurately perform static prediction of resource usage for ML-like functional programs with references and arrays. Previous research successfully integrated the potential method of amortized analysis with a standard type system to automatically derive parametric resource bounds. The analysis is naturally compositional and the resource consumption of functions can be abstracted using potential-annotated types. The soundness theorem of the analysis guarantees that the derived bounds are correct with respect to the resource usage defined by a cost semantics. Type inference can be efficiently automated using off-the-shelf LP solvers, even if the derived bounds are polynomials. However, side effects and aliasing of heap references make it notoriously difficult to derive bounds that depend on mutable structures, such as arrays and references. As a result, existing automatic amortized analysis systems for ML-like programs cannot derive bounds for programs whose resource consumption depends on data in such structures. This article extends the potential method to handle mutable structures with minimal changes to the type rules while preserving the stated advantages of amortized analysis. To do so, we introduce a swap operation for references and arrays that users can use to make programs suitable for automatic analysis. We prove the soundness of the analysis introducing a potential-annotated memory typing, which gathers all unique locations reachable from a reference. Apart from the design of the system, the main contribution is the proof of soundness for the extended analysis system.

1998 ACM Subject Classification D.2.4 Software/Program Verification

Keywords and phrases Resource Analysis, Functional Programming, Static Analysis, OCaml, Amortized Analysis

Digital Object Identifier 10.4230/LIPIcs.FSCD.2017.26

1 Introduction

Several tools currently exist that can automatically derive loop and recursion bounds for imperative programs including COSTA and SACO [1, 3], KoAT [8], CoFloCo [14], SPEED [17], and LOOPUS [35]. These analyses produce impressive results for integer programs, but mutation, cycles, and the absence of type information make it difficult to automatically derive bounds that depend on the sizes of pointer-based data structures. One approach to deal with the problem is to represent the size of data structures with ghost variables [2].

In purely functional programs, reasoning about heap-based data structures is more feasible since there are strong type guarantees on the shape of the data. For example, we know

[*] This article is based on research that has been supported, in part, by AFRL under DARPA STAC award FA8750-15-C-0082, by NSF under grant 1319671 (VeriQ), and by the Eric and Wendy Schmidt Fund for Strategic Innovation. Any opinions, findings, and conclusions contained in this document are those of the authors and do not necessarily reflect the views of the sponsoring organizations.

2nd International Conference on Formal Structures for Computation and Deduction (FSCD 2017).
Editor: Dale Miller; Article No. 26; pp. 26:1–26:20
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

that a functional list is immutable and does not have cycles. As a result, there are several techniques that can automatically or semi-automatically derive bounds that depend on the sizes of functional data structures. Automation is often achieved by relying on type systems [11, 12, 29], recurrence relations [6, 13], and automatic amortized resource analysis (AARA) [9, 18, 19, 22, 27]. These techniques work well as long as the programs are purely functional. The only technique that can derive bounds for ML-like programs with references and arrays is the AARA [21] that is implemented in Resource Aware ML (RAML) [19]. While this analysis works well for higher-order programs that use mutable heap structures to store functions, it cannot derive bounds for programs if their execution depends on the size of data structures that are stored in mutable structures.

In this article, we propose an extension of RAML for automatically deriving symbolic resource bounds that depend on the size of data that are stored in references and arrays. We build off of previous work on type-based amortized resource analysis, which has been shown to be able to infer polynomial bounds for functional programs with nested data structures [22, 18, 19]. Like in previous work, we achieve compositionality by integrating the analysis with a standard type system. Type inference is reduced to efficient linear constraint solving and type derivations can be used as certificates that prove the correctness of the bound. Our technique is also parametric in the resource of interest (e.g., analyzing heap usage, clock cycles, and so on) and works for non-monotone resources that can become available during the execution.

The proposed type system is simpler than existing techniques that incorporate AARA in more imperative settings using separation logic [4] or object-oriented types [23]. One advantage is that our technique can be smoothly integrated with the efficient LP-based type inference of RAML. As a result, bound inference is fully automatic. Another advantage is that the design of our resource-annotated types mirror the design decisions of the ML type system, which is the basis of RAML. A disadvantage is that our system is less expressive than other approaches and requires a certain style of programming. However, the advantages that we get from the automation seem to outweigh the disadvantages, and programming with our system is relatively straightforward. For example, as we show in Section 5, we can automatically derive a bound for a graph search algorithm that traverses and mutates a potentially cyclic data structure.

To illustrate the main ideas of our type system, we describe it for a minimal subset of RAML and restrict the technical development to linear bounds. As we argue in Section 6, the proposed techniques carry over to the polynomial and higher-order setting. Apart from the design of the system, our main contribution is the soundness proof of the analysis with respect to an operational cost semantics. To properly calculate the potential of data stored in mutable heap cells at a given program state, we leverage the idea of *memory typing* [36]. This addition is essential to show the soundness of operations like `swap` that involve mutable heap cells.

2 Language Definition and Semantics

We now define a minimal first-order functional language that only contains the features that are relevant to our contributions. The syntax of our language is defined as follows, where $x \in \mathrm{VID}$ (the set of variable identifiers), $f \in \mathrm{FID}$ (the set of function identifiers), $c \in \mathrm{CID}$ (the set of constructor identifiers), and $n \in \mathbb{Z}$.

$$\begin{aligned} e ::= {} & x \mid n \mid f(x_1,\dots,x_n) \mid \text{let } x = e_1 \text{ in } e_2 \mid \text{true} \mid \text{false} \mid () \mid c\,\langle x_1,\dots,x_k\rangle \\ & \mid \text{match } x \text{ with } c\,\langle x_1,\dots,x_k\rangle \Rightarrow e_1 \mid e_2 \\ & \mid \text{ref } x \mid !x \mid x_1 := x_2 \mid \text{swap}(x_1,x_2) \mid \text{share } x \text{ as } (x_1,x_2) \text{ in } e \\ & \mid \text{create}(x,e) \mid \text{get}(x_1,x_2) \mid \text{set}(x_1,x_2,x_3) \mid \text{aswap}(x_1,x_2,x_3) \mid \text{length}(x) \end{aligned}$$

$$F ::= \cdot \mid (f,\vec{x},e_f) :: F$$

Function definitions are mutually recursive and given by triples that consist of a function name f, a vector of formal arguments $\vec{x}$, and a function body e_f. Expressions include function calls, constructors $c\,\langle x_1,\dots,x_k\rangle$, pattern matches, and the usual operations on references and arrays. We use `ref x` to instantiate a reference, `!x` to retrieve the value of a reference, and `x1 := x2` to set the value of a reference. However, to enable data that has been stored in mutable heap cells to be considered in resource analysis, we leverage `swap` operations [34] for references, which combines retrieval and update in one operation. Similarly for arrays, we use `create(n,e)` to initialize an array of length `n` with each cell set to the value expression `e`, `get(A,i)` to retrieve the `i`th value of `A`, `set(A,i,x)` to set the `i`th value of `A` to be `x`, and `aswap(A,i,x)` to combine retrieval and update like with `swap`.

To simplify type rules and proofs, we assume programs are in *share-let normal form*, where term formers are only applied to variables, as much as it does not restrict expressivity. Furthermore, we use an affine type system, so that every bound variable can occur at most once. If some bound value x must be used more than once, the term `share` x `as` (x_1, x_2) `in` e must be used to bind the value of x to x_1 and x_2 in the expression e. We note that in the implementation of RAML, the programmer is not required to either write in share-let normal form or use the `share` expression, as the compiler translates the program into this form before analysis occurs.

Big-Step Operational Cost Semantics

Let *Loc* be an infinite set of memory locations and define the set of values $v \in \mathit{Val}$ to be

$$v ::= \ \ell \mid \text{Null} \mid \texttt{tt} \mid \texttt{ff} \mid (\mathsf{constr}_c, v_1,\dots,v_k) \mid (\sigma, n)$$

where $\ell \in \mathit{Loc}$ and an array (σ, n) consists of a size $n \in \mathbb{N}$ and a map $\sigma\colon \{0,\dots,n{-}1\} \to \mathit{Loc}$.

We further define a *heap* H to map locations to values, an *environment* V to map variable identifiers to values, and a *resource metric* $M\colon K \times \mathbb{N} \to \mathbb{Q}$ to describe the resource consumption in each evaluation step of the big-step semantics, where K is a set of constants describing each possible operation in the language. We write M^k_n for $M(k,n)$ and M^k for $M(k,0)$ in the style of previous work [19]. A metric defines the constant cost of different atomic steps in the cost semantics.

To formalize the notion of tracking resources that can become available during evaluation, we define the *high-water mark* of the resource usage to be the maximal number of resource units under a given metric that are simultaneously used during an evaluation. We use this information in the soundness proof to relate the bound of resource usage generated by the type system to the actual cost of program evaluation.

The operational evaluation rules in Appendix A define the evaluation judgement

$$F, V, H \,{}_M\!\vdash e \Downarrow (v, H') \mid (q, q')$$

where, given a family of functions F, environment $V \colon \text{VID} \rightharpoonup \mathit{Val}$, initial heap $H \colon \mathit{Loc} \rightharpoonup \mathit{Val}$, and resource metric M, the expression e evaluates to the value v and new heap H', requiring $q \in \mathbb{Q}_0^+$ resource units to evaluate and leaving $q' \in \mathbb{Q}_0^+$ resource units available after evaluation. The net resource consumption is $\delta = q - q'$. Note that δ is negative if more resources become available than are consumed during the execution of e.

We define the operation $(q,q')\cdot(p,p')$ to account for an evaluation made up of an evaluation with resource consumption (q,q') followed by an evaluation with resource consumption (p,p') as follows:

$$(q,q')\cdot(p,p') = \begin{cases} (q+p-q',\ p') & \text{if } q' \leq p \\ (q,\ p'+q'-p) & \text{if } q' > p \end{cases}$$

We present selected rules from the operational semantics below. We note that the (E:Fun) and (E:Let) rules are minimally changed from previous work, and the (E:Constr) rule is a generalized form of the rule for specific datatypes in earlier forms of Resource Aware ML.

$$\frac{(f,\vec{y},e_f) \in F \qquad F,[y_1 \mapsto V(x_1),\cdots,y_n \mapsto V(x_n)], H\,{}_M\!\vdash e_f \Downarrow (v,H') \mid (q,q')}{F,V,H\,{}_M\!\vdash f(x_1,\ldots,x_n) \Downarrow (v,H') \mid M_1^{\mathrm{fun}_n}\cdot(q,q')\cdot M_2^{\mathrm{fun}_n}}\,(\text{E:Fun})$$

$$\frac{F,V,H\,{}_M\!\vdash e_1 \Downarrow (v_1,H_1) \mid (q,q') \qquad F,V[x \mapsto v_1],H_1\,{}_M\!\vdash e_2 \Downarrow (v_2,H_2) \mid (p,p')}{F,V,H\,{}_M\!\vdash \text{let } x = e_1 \text{ in } e_2 \Downarrow (v_2,H_2) \mid M_1^{\mathrm{let}}\cdot(q,q')\cdot M_2^{\mathrm{let}}\cdot(p,p')\cdot M_3^{\mathrm{let}}}\,(\text{E:Let})$$

$$\frac{c_i \in \text{CID} \qquad v = (\mathsf{constr}_{c_i}, V(x_1),\ldots,V(x_k)) \qquad H' = H, \ell \mapsto v \qquad \ell \notin \text{dom}(H)}{F,V,H\,{}_M\!\vdash c_i\,\langle x_1,\ldots,x_k\rangle \Downarrow (\ell,H') \mid M^{\mathrm{cons}}}\,(\text{E:Constr})$$

The rule (E:Fun) can be read as follows. We start evaluating $f(x_1,\ldots,x_n)$ by incurring the constant resource cost $M_1^{\mathrm{fun}_n}$, which depends on the number of function arguments n. If the function f is available in the current context of functions, associated with parameters $y_1,\ldots,y_n$ and body e_f, then e_f is evaluated in an environment where each parameter y_i maps to the value associated with the variable x_i in the calling environment. Moreover, e_f evaluates to the value v and new heap H' with resource cost (q,q'). Then we incur the constant resource cost $M_2^{\mathrm{fun}_n}$ to complete evaluation.

The rule (E:Let) is similar. We first incur the constant resource cost M_1^{let} to begin evaluation, and then in the same environment and heap, evaluate e_1 to the value v_1 and new heap H_1 with resource cost (q,q'). Then, after incurring another constant resource cost M_2^{let}, we evaluate e_2 under the new heap and the same environment with the bound variable x mapping to the value v_1. We thus evaluate e_2 to the value v_2 and heap H_2 with resource cost (p,p'), and then conclude evaluation with the constant cost M_3^{let}.

Lastly, the rule (E:Cons) can be understood as follows. If c_i is a valid constructor identifier, we create a new value $(\mathsf{constr}_{c_i}, V(x_1),\ldots,V(x_n))$ that is a tuple of the constructor name and the values associated with each of its arguments. We then add a fresh memory location ℓ and evaluate to our new value and a heap that has been extended with the new location ℓ pointing to the new value. This all incurs the constant resource cost M^{cons}.

The remaining rules for the operational semantics are presented in Appendix A.

Well-Formed Environments

Our type judgement takes the form $\Sigma;\Gamma \vdash e : T$; under the function context Σ, which describes the types of functions currently in scope, and the variable context Γ, which maps variables to their types, the expression e has type T. The rules for the judgement are presented in

Appendix B. We note that these rules only differ from those discussed in Section 3 in their exclusion of resource annotations, which are developed in the next section.

Given a heap H, value v, and type T, the judgement $H \vDash v : T$ means that the value v under the heap H is well-formed with respect to T. We denote that an environment V and heap H are *well-formed* with respect to a context Γ with $H \vDash V : \Gamma$ if $H \vDash V(x) : \Gamma(x)$ holds for every $x \in \mathrm{dom}(\Gamma)$.

We show that under any evaluation with a well-formed environment, every location in the environment remains well-formed and the return value is also well-formed. We include the proof for Theorem 1 in the technical report [30].

▶ **Theorem 1.** *If* $\Sigma; \Gamma \vdash e : T$, $H \vDash V : \Gamma$, *and* $F, V, H \vdash_M e \Downarrow (v, H') \mid (q, q')$ *then* $H' \vDash V : \Gamma$ *and* $H' \vDash v : T$.

3 Annotated Types for Linear Resource Analysis

The core idea of AARA is to annotate each program point with a *potential function* which maps sizes of reachable data structures to non-negative numbers. The potential functions must ensure that, for every input and every possible evaluation, the potential at a program point is sufficient to pay for the resource cost of the following transition and the potential at the next point. It then follows that the initial potential function describes an upper bound on the resource consumption of the program.

In the examples in this section, we use a resource metric in which we count the number of constructor applications.

Resource Annotations

To perform AARA for our language, we annotate our types for inductive data structures with non-negative rational numbers $q \in \mathbb{Q}_0^+$, defining the linear resource-annotated data types. To annotate types in a compact manner, we use a function Q for each inductive type that maps constructor names to their associated potential annotations. The data types of our language are defined as follows.

$$A ::= 1 \mid B \mid X \mid \mu X^Q.\{c_i : (A_1, \ldots, A_{k_i})\}_i \mid A \text{ ref} \mid A \text{ array}^q \,.$$

Let $\mathscr{A}_{\text{lin}}$ be the set of linear resource-annotated data types and $\mathscr{T}$ be the set of unannotated types. We use 1 as the unit type, B for booleans, X for type variables, A ref for references with data of type A, and A array for arrays with data of type A. We also use recursive datatypes of the form $\mu X^Q.\{c_i : (A_1, \ldots, A_{k_i})\}_i$, where the type variable X is bound in the types A_1 through A_{k_i} for each constructor c_i.

We define the type $N^q = \mu X^{Q_N}.\{\text{zero} : 1 \mid \text{succ} : X\}$ to be used with arrays, where $Q_N[\text{zero}] = 0$ and $Q_N[\text{succ}] = q$. Additionally, for the purpose of the examples in this section, we use the type $L^q(A) = \mu X^{Q_L}.\{\text{nil} : 1 \mid \text{cons} : (A, X)\}$, where $Q_L[\text{nil}] = 0$ and $Q_L[\text{cons}] = q$. This represents a list containing elements of type A, where each element carries q potential. We also have linear resource-annotated first-order types, which are defined by the following grammar.

$$R ::= (A_1, \ldots, A_n) \xrightarrow{q/q'} A \,.$$

In this notation, $q, q' \in \mathbb{Q}_0^+$ and $A, A_i \in \mathscr{A}_{\text{lin}}$, meaning that q is the constant potential before a call to the function and q' is the constant potential after the call to the function. Let $\mathscr{R}_{\text{lin}}$ denote the set of linear resource-annotated first-order types.

The potential annotation of a type directly determines the potential represented by values of that type. For example, when we consider a list of length n and of type $L^q(T)$, the potential carried by that list is qn, where each element carries q resource units. With the resource metric considered here, we are able to pay for up to q `cons` operations for each element in the list (since each element carries q resource units).

We can more formally define the potential of a value. Let $A \in \mathscr{A}_{\text{lin}}$, let H be a heap, and let $v \in \mathit{Val}$ be a value with $H \vDash v : A$. Then, if $H(\ell) = (\mathsf{constr}_{c_i}, v_1, \dots, v_{k_i})$ and $A = \mu X^Q.\{\cdots \mid c_i : (A_1, \dots, A_{k_i}) \mid \dots\}$, we have that

$$\Phi_H(\ell : A) = Q[c_i] + \sum_{i=1}^{k_i} \Phi_H(v_i : [A/X]A_i)$$

where $[A/X]A_i$ represents the substitution of the type A for every instance of the type variable X in A_i. If $H(\ell) = (\sigma, n)$ and $A = A' \text{ array}^q$, then $\Phi_H(\ell : A) = qn$. Otherwise, $\Phi_H(v : A) = 0$.

We establish the following definition for the amount of potential provided by a typing context under a particular environment V and heap H.

$$\Phi_{V,H}(\Gamma) = \sum_{x \in \text{dom}(\Gamma)} \Phi_H(V(x) : \Gamma(x))\,.$$

In particular, we sum over the potential contributed by each variable in scope by the definition presented earlier. However, note that this causes data stored within references and arrays to not be included by the above definition of Φ_H.

In order to relate resource-annotated types with different resource annotations, we define the function $|\cdot| : \mathscr{A}_{\text{lin}} \to \mathscr{T}$ that simply removes the potential annotations from a given type. If we have annotated types A and A' such that $|A| = |A'|$, we say that they have the same *underlying types*.

The sharing relation $\curlyvee$ defines how the potential of a variable can be shared by multiple occurrences of that variable. We have $A \curlyvee (A_1, A_2)$ if and only if $|A| = |A_1| = |A_2|$ and for every heap H and value v such that $H \vDash v : A$, $\Phi_H(v : A) = \Phi_H(v : A_1) + \Phi_H(v : A_2)$ holds. The sharing relation $\curlyvee$ is the smallest relation such that the following hold.

$$\frac{A \in \{1, B, N, A' \text{ ref}\}}{A \curlyvee (A, A)}\,(S_1) \qquad\qquad \frac{p = q + r}{A \text{ array}^p \curlyvee (A \text{ array}^q, A \text{ array}^r)}\,(S_2)$$

$$\frac{A_i \curlyvee (A_i', A_i'') \qquad \forall 1 \le i \le k.\, P[c_i] = Q[c_i] + R[c_i]}{\mu X^P.\{c_i : (A_1, \dots, A_k)\}_i \curlyvee (\mu X^Q.\{c_i : (A_1', \dots, A_k')\}_i, \mu X^R.\{c_i : (A_1'', \dots, A_k'')\}_i)}\,(S_3)$$

▶ **Example 2.** Consider the standard `append` function for our list type. With our resource metric, the cost of append is n where n is the length of the first argument. To derive a bound in our type system, we assign the function `append` the type $(L^1(T), L^0(T)) \to L^0(T)$. This function can also be assigned the type $(L^2(T), L^1(T)) \to L^1(T)$ and so forth; as long as the annotations represent that one resource unit has been consumed for every element of the first input list and there is enough potential remaining to pay for the potential of the result, the potential annotations are valid. The second annotation can be used for the inner call of append in an expression like `append(append(x,y),z)`.

We now consider the following function that calls `append`.

```
f l = share l as (l1,l2) in let _ = append(l1,[]) in append(l2,[])
```

In this example program, we see the results of reusing a value that carries potential. Since we call `append` on the same list twice, it must have sufficient potential to be iterated over twice. In order to do so, when we share the list between two variables, we split the potential over the two, and thus assign the type $L^2(T) \to L^0(T)$ to `f`. In the type derivation, the variable `l` has type $L^2(T)$, and `l1` and `l2` have type $L^1(T)$.

A *resource-annotated typing judgement* has the form $\Sigma; \Gamma \vdash^{q}_{q'} e : A$. This means that, under environment V and heap H such that $H \vDash V : \Gamma$ holds, as well as a resource-annotated signature Σ and resource-annotated context Γ, the expression e has the resource-annotated data type A. If there are at least $q + \Phi_{V,H}(\Gamma)$ resource units available, then e may be evaluated. Furthermore, if e evaluates to a value v, resulting in an updated heap H', there are more than $q' + \Phi_{H'}(v : A)$ resource units left.

We define a well-typed program to consist of a resource-annotated signature Σ and a family $F = (f, \vec{x}, e_f)_{f \in \mathrm{dom}(\Sigma)}$ of function identifiers $f \in \mathrm{FID}$ with variable identifiers $\vec{x} = x_1, \ldots, x_n \in \mathrm{VID}$ and expression e_f such that $\Sigma; x_1{:}A_1, \ldots, x_n{:}A_n \vdash^{q}_{q'} e_f : A$ for each $(A_1, \ldots, A_n) \xrightarrow{q/q'} A \in \Sigma(f)$.

We present selected type rules related to the example above. The rules (L:Fun) and (L:Share) are unchanged from previous work, and the (L:Constr) and (L:Mat) rules are expanded to handle the more general form of recursive datatypes.

$$\frac{\begin{array}{c} c_i \in \mathrm{CID} \\ A = \mu X^P.\{\cdots \mid c_i{:}(A'_1, \ldots, A'_k) \mid \cdots\} \qquad \forall 1 \le j \le k.\, A_j = A'_j \vee (A_j = A \wedge A'_j = X) \end{array}}{\Sigma; x_1{:}A_1, \ldots, x_k{:}A_k \vdash^{q+P[c_i]+M^{\mathrm{cons}}}_{q} c_i\ (x_1, \ldots, x_k) : A} \;\text{(L:Constr)}$$

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

$$\frac{(A_1, \ldots, A_n) \xrightarrow{q/q'} A \in \Sigma(f)}{\Sigma; x_1{:}A_1, \ldots, x_n{:}A_n \vdash^{q+M_1^{\mathrm{fun}_n}}_{q'-M_2^{\mathrm{fun}_n}} f(x_1, \ldots, x_n) : A} \;\text{(L:Fun)}$$

The rule (L:Constr) can be understood as follows. Since the construction of a new list element costs M^{cons} resource units, we have to pay M^{cons}, as well as the potential that is available after the evaluation. The potential of each of the components of the constructor is paid for by the context, and the missing potential $P[c_i]$ of the new list element, the constant cost M^{cons}, and the resulting potential q are paid by the intial constant potential $q + P[c_i] + M^{\mathrm{cons}}$.

The rule (L:Share) incurs no resource consumption. However, the type A' of the variable x is split into two new types A_1 and A_2 for the new variables x_1 and x_2, which share the potential associated with x.

Lastly, in the rule (L:Fun), the evaluation of $f(x_1, \ldots, x_n)$ incurs the constant cost $M_1^{\mathrm{fun}_n}$ before the evaluation of the body and $M_2^{\mathrm{fun}_n}$ after the evaluation of the body. Since $(A_1, \ldots, A_n) \xrightarrow{q/q'} A \in \Sigma(f)$, we know that the body evaluates with initial constant potential q. Thus in order to pay for the cost before evaluation, we need to have initial potential $q + M_1^{\mathrm{fun}_n}$ for the whole expression. Similarly, we have remaining constant potential q' after evaluation of the body, so after evaluating the whole expression, we have remaining constant potential $q' - M_2^{\mathrm{fun}_n}$.

Potential in References

We now introduce references by way of the following example, where the function `g'` might use the data in the supplied reference in some unknown way.

```
g l = let r = ref l in
      share r as (r1,r2) in
      let _ = g' r1 in
      append(!r2,[])
```

If we assign a type of the form $L^q(T)\ \mathit{ref} \to 1$ to `g'`, we have a weak contract between `g` and `g'` concerning the usage of the data referenced by `r`. Looking only at the type of `g'`, we cannot know how `g'` consumes the potential of the data referenced by `r`. Therefore, we also cannot know how much potential we have left over on the data in `r` after `g'` executes.

The most straightforward way to address the ambiguity of mutable data usage is to insist that the potential annotations of mutable data must remain invariant. By requiring this, we are guaranteed that we can freely share, pass around, and alias references without being concerned with tracking every possible use of the underlying data. This restriction of invariant potential annotations is precisely in line with the goals of the ML type system. As opposed to carrying around extra information about effectful computation statically, the only information that ML tracks is the unchanging type of data that is stored in the reference. Furthermore, as the number of mutable cells generated in a program scales, so too does the opportunity for aliased references to be passed to some subroutine. If the potential annotations of these references could vary, then the system would have to track all possible combinations of distinct and aliased references in these situations in order to successfully analyze resource usage. As this approach clearly cannot scale, we resort to demanding invariant potential annotations for references.

However, when considering the above example, the action of sharing references that contain lists cannot operate in the same way it does when we directly share lists. We note that when sharing the reference `r`, we cannot split the potential of the underlying data (as we do when sharing a list directly) due to our new requirement of invariant potential annotations. In other words, since `r`, `r1`, and `r2` all point to the same heap location, they all must contain the same potential annotation for the data stored there.

If we suppose now that `g'` consumes the potential stored in `r` in the function `g` from before, we find that we have violated the soundness of our system. In this scenario, we see that the same potential is being used to pay for the call to `append` in `g` as well as for whatever operation occurs in `g'`. As discussed above, in a situation not involving references, this duplication would be explicitly handled by using `share`, but due to our invariant potential annotations for references, it does not in this case. Therefore, our program uses double the amount of potential that the type annotations provide.

Introducing Swap

In order to prevent uncontrolled duplication of potential, we use the `swap` operation to better track usage of data in mutable structures, as used in previous treatments of substructural type systems [34, 32], where it is operationally defined as follows:

```
swap (r,l) ≡ let x = !r in let _ = (r := l) in x.
```

With `swap`, we ensure that in order to use data that has been stored in a reference in a way that consumes potential, it must be "swapped" out for data of the same type that satisfies the same potential annotation. As a result, for data in a single reference to be used twice, it must be swapped out, explicitly shared, and swapped back in.

▶ **Example 3.** If we use `swap` to access data in mutable structures, then we must take care in situations where two references may be, but are not necessarily, aliased so that we do not mistakenly expect to use data that has already been destructively modified. Consider the following program representing this situation, in which we append two lists after retrieving them from references that are possibly aliased to each other. With this, we demonstrate a programming style that ensures that Resource Aware ML can successfully compute a bound.

```
f (r1, r2) =
  let l2_1 = !r2 in
  let l1 = swap(r1,[]) in
  let l2_2 = swap(r2,[]) in
  match (l2_1,l2_2) with
  | ([],_) -> l1
  | (_,[]) -> share l1 as (l1_1,l1_2) in
              append(l1_1,l1_2)
  | (_,_) ->  append(l1,l2_2)
```

Here, we only perform the normal `append` operation if the data stored in `r2` is a non-empty list before *and* after `l1` is retrieved. Otherwise, if it is non-empty before and empty after, we know that the two references must be aliased, and therefore, to achieve the proper result, we duplicate the data retrieved from the first (and therefore explicitly share its potential) and then append the resulting lists. Therefore, if we assign the type $L^p(T)$ *ref* to `r1` and $L^q(T)$ *ref* to `r2`, we thus assign the type $L^p(T)$ to `l1`, $L^0(T)$ to `l2_1`, and $L^q(T)$ to `l2_2`.

It is allowed in our system to dereference data using the usual `!` operator. However, we ensure in our type system that data retrieved in this way has no potential. As is shown by the above example, while the `swap` operation places a new burden on the programmer, it is not unreasonable to work around the unintended effects it may cause.

An interesting observation is that the potential annotations in function types contain information about aliasing. One possible typing for the function `f` is $(L^1(T)\ \mathit{ref}, L^1(T)\ \mathit{ref}) \to L^0(T)$. Here, `r1` and `r2` are potentially aliased. Another possible typing for the function `f` is $(L^1(T)\ \mathit{ref}, L^0(T)\ \mathit{ref}) \to L^0(T)$. This typing implies that the arguments are not aliased; if they were, they would break the invariant of our type system discussed earlier. We now present the type rules that are related to references.

$$\frac{}{\Sigma; x{:}A \vdash^{q+M^{\mathrm{ref}}}_{q} \mathrm{ref}\ x : A\ \mathrm{ref}}\ (\text{L:Ref}) \qquad \frac{}{\Sigma; \Gamma, x_1{:}A\ \mathrm{ref}, x_2{:}A \vdash^{q+M^{\mathrm{assign}}}_{q} x_1 := x_2 : 1}\ (\text{L:Assign})$$

$$\frac{|A'| = |A| \qquad A \curlyvee (A, A)}{\Sigma; x{:}A'\ \mathrm{ref} \vdash^{q+M^{\mathrm{dref}}}_{q} !x : A}\ (\text{L:DRef}) \qquad \frac{}{\Sigma; x_1{:}A\ \mathrm{ref}, x_2{:}A \vdash^{q+M^{\mathrm{swap}}}_{q} \mathrm{swap}(x_1, x_2) : A}\ (\text{L:Swap})$$

The rule (L:Ref) can be read as follows. Our context contains just the variable $x{:}A$. We thus create a reference of type A ref. Since the operational semantics state that the cost of evaluating such a reference is M^{ref}, we need initial constant potential $q + M^{\mathrm{ref}}$ to pay for this cost and the remaining constant potential q. The rules (L:Assign) and (L:Swap) are similar.

(L:DRef) is also similar, but additionally restricts the type of the dereferenced value. The premise $A \curlyvee (A, A)$ requires that the type have a potential annotation of zero, thus ensuring that the return value has no potential. The first premise simply ensures that the type of the reference and the type of the dereferenced value only differ in their potential annotations.

Potential for Arrays

The concerns and techniques presented above extend cleanly from references to arrays. As before, for each cell in an array, we cannot always tell statically how much potential has been used up by effectful subroutines. Therefore, we maintain the same requirement of invariant potential for every cell in the array, and use the operation `aswap(A,i,x)`, which is defined as follows:

```
aswap(A,i,x) ≡ share A as (A1,A2) in share i as (i1,i2) in
                  let x’ = get(A1,i1) in let _ = set(A2,i2,x) in x’
```

The relevant rules follow.

$$\frac{\Sigma;\emptyset \vdash^{p-r}_{0} e : A}{\Sigma; x{:}N^{p} \vdash^{q+M^{\mathrm{create}}}_{q} \mathrm{create}(x,e) : A\ \mathrm{array}^{r}}\ \text{(L:Create)}$$

$$\frac{|A'| = |A| \qquad A \curlyvee (A,A)}{\Sigma; x_1{:}A'\ \mathrm{array}^{p}, x_2{:}N^{r} \vdash^{q+M^{\mathrm{get}}}_{q} \mathrm{get}(x_1,x_2) : A}\ \text{(L:Get)}$$

$$\frac{}{\Sigma; x_1{:}A\ \mathrm{array}^{p}, x_2{:}N^{r}, x_3{:}A \vdash^{q+M^{\mathrm{set}}}_{q} \mathrm{set}(x_1,x_2,x_3) : 1}\ \text{(L:Set)}$$

$$\frac{}{\Sigma; x_1{:}A\ \mathrm{array}^{p}, x_2{:}N^{r}, x_3{:}A \vdash^{q+M^{\mathrm{aswap}}}_{q} \mathrm{aswap}(x_1,x_2,x_3) : A}\ \text{(L:Aswap)}$$

The rule (L:Create) requires the user to specify a default expression to initialize each cell of the new array. This expression is then evaluated using the potential from the number specifying the length of the array. Since this pays for the evaluation of each cell, the system simply requires enough initial constant potential to pay for the constant resource cost, M^{create}, as well as the remaining constant potential, q. Otherwise, (L:Get), (L:Set), and (L:Aswap) are analogous to their counterparts for references. The remaining typing rules are provided in Appendix C.

4 Soundness

We now show that the operational semantics and linear resource-annotated typing rules cohere, proving that type derivations establish correct bounds. We claim that if an expression e evaluates to a value v in a well-formed environment then the initial potential of the context is an upper bound on the watermark of the resource usage. Furthermore, the difference between the initial and final potential is an upper bound on the consumed resources.

To ensure that aliased references are not double-counted when determining the potential that they contribute, we use a *memory typing* $\Delta\colon \mathit{Loc} \to \mathscr{A}_{\mathrm{lin}}$. Under a given context Γ, environment V, and heap H, Δ maps all locations in the heap pointed to by references contained in Γ to the types of their values. This is similar to the usage of store typing in previous work on mutable structures in substructural settings [36]. We represent this relationship with the judgement $H, \Delta \vDash V : \Gamma$ and the operator $\otimes$ defined by the rules in Figure 1, using $\emptyset$ to represent the empty reference collection.

The most interesting rules in Figure 1 are (Δ:Ref), (Δ:Array), (Δ:Constr), and (Δ:Full). The first of these can be read as follows. When checking that the location ℓ is well-formed at

$$\frac{}{\emptyset \otimes \Delta = \Delta}\;(\otimes : \mathrm{U1}) \qquad\qquad \frac{}{\Delta \otimes \emptyset = \Delta}\;(\otimes : \mathrm{U2})$$

$$\frac{\forall i \in \{1,2\}.\forall \ell \in \mathrm{dom}(\Delta_i).\Delta(\ell) = \Delta_i(\ell) \qquad \forall \ell \in \mathrm{dom}(\Delta).(\ell \in \mathrm{dom}(\Delta_1)) \vee (\ell \in \mathrm{dom}(\Delta_2))}{\Delta_1 \otimes \Delta_2 = \Delta}\;(\otimes : \mathrm{C})$$

$$\frac{}{H, \Delta \vDash \mathtt{tt} : B}\;(\Delta : \mathrm{True}) \qquad \frac{}{H, \Delta \vDash \mathtt{ff} : B}\;(\Delta : \mathrm{False}) \qquad \frac{}{H, \Delta \vDash \mathrm{Null} : 1}\;(\Delta : \mathrm{Unit})$$

$$\frac{\begin{array}{c} H(\ell) = (\mathsf{constr}_{c_i}, v_1, \ldots, v_k) \qquad H' = H \setminus \ell \\ A = \mu X^Q.\{\cdots \mid c_i{:}(A_1, \ldots, A_k) \mid \cdots\} \\ H', \Delta_1 \vDash v_1 : [A/X]A_1 \quad \ldots \quad H', \Delta_k \vDash v_k : [A/X]A_k \qquad \Delta = \bigotimes_{1 \le j \le k} \Delta_i \end{array}}{H, \Delta \vDash \ell : \mu X^Q.\{\cdots \mid c_i{:}(A_1, \ldots, A_k) \mid \cdots\}}\;(\Delta : \mathrm{Constr})$$

$$\frac{H(\ell) = \ell' \qquad H, \Delta' \vDash \ell' : A \qquad \Delta = \Delta' \otimes \{\ell' \mapsto A\}}{H, \Delta \vDash \ell : A \text{ ref}}\;(\Delta : \mathrm{Ref})$$

$$\frac{H(\ell) = (\sigma, n) \qquad \forall i < n.H, \Delta_i' \vDash H(\sigma(i)) : A \qquad \Delta_i = \Delta_i' \otimes \{\sigma(i) \mapsto A\} \qquad \Delta = \bigotimes_{0 \le i < n} \Delta_i}{H, \Delta \vDash \ell : A \text{ array}^q}\;(\Delta{:}\mathrm{Array})$$

$$\frac{\mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(V) \qquad \forall x \in \mathrm{dom}(\Gamma).H, \Delta_x \vDash V(x) : \Gamma(x) \qquad \Delta = \bigotimes_{x \in \mathrm{dom}(\Gamma)} \Delta_x}{H, \Delta \vDash V : \Gamma}\;(\Delta{:}\mathrm{Full})$$

Figure 1 Rules Defining the $\otimes$ operator and the well-formed reference collection.

type A ref under heap H, we first ensure that the location points to some other location ℓ' and then check that ℓ' is well-formed at type A under the same heap. If this check gives us back the memory typing Δ', we add the mapping $\ell' \mapsto A$ to Δ' to make Δ. As discussed above, we do this because there exists a reference that points to ℓ', and so we track it as intended. We thus return Δ as the memory typing for the location ℓ. The (Δ:Array) rule is similar, but we check each location pointed to by the array and combine the resulting memory typings. Likewise, the (Δ:Constr) rule checks that each value included in the constructor is well-formed and combines the the resulting memory typings. Lastly, the (Δ:Full) rule checks each value included in the environment V is well-formed and returns the combination of the constructed memory typings.

Note that $H, \Delta \vDash V : \Gamma$ implies $H \vDash V : \Gamma$. Furthermore, we intentionally allow extra locations to be present in Δ, as this simplifies the soundness proof without contributing extra potential. We now define the potential of Δ as follows:

$$\Phi_H^D(\Delta) = \sum_{\ell \in \mathrm{dom}(\Delta)} \Phi_H(\ell : \Delta(\ell))\,.$$

By defining the potential of mutable cells in this way, we ensure that every reachable heap cell is counted exactly once, thereby disallowing the possibility of aliasing leading to extra potential.

Given this additional method of contributing potential, the typing judgement $\Sigma; \Gamma \mathrel{\Big|\!\frac{q}{q'}} e : A$ now means that there must be $q + \Phi_{V,H}(\Gamma) + \Phi_H^D(\Delta)$ resource units available to evaluate e, and after evaluation to a value v, there will be $q' + \Phi_{H'}(v : A) + \Phi_{H'}^D(\Delta_{\mathrm{ret}})$ units available, where $H, \Delta \vDash V : \Gamma$ and $H', \Delta_{\mathrm{ret}} \vDash v : A$. We can now formally state our soundness claim.

▶ **Theorem 4.** *Let $H, \Delta \vDash V : \Gamma$ and $\Sigma; \Gamma \mathrel{\Big|\!\frac{q}{q'}} e : A$ hold. If $F, V, H\,{}_{M}\!\vdash e \Downarrow (v, H') \mid (p, p')$ then there exists some Δ_{ret} such that the following hold:*

- $p \leq \Phi_{V,H}(\Gamma) + \Phi^{D}_{H}(\Delta) + q$
- $p - p' \leq \Phi_{V,H}(\Gamma) + \Phi^{D}_{H}(\Delta) + q - (\Phi_{H'}(v{:}A) + \Phi^{D}_{H'}(\Delta \otimes \Delta_{ret}) + q')$
- $H', \Delta \vDash V : \Gamma$ *and* $H', \Delta_{ret} \vDash v : A$

Theorem 4 is proved by a nested induction on the derivation of the evaluation judgement and the type judgement, with the latter being needed due to the structural rules. We present the proof of Theorem 4 in the technical report [30]. Moreover, we note that the proof is similar to the soundness proof for resource analysis for the language without mutable heap cells [18].

The soundness proof uses Lemma 5 to show the soundness of the rule L:Let, stating that the potential of a context is invariant during the evaluation.

▶ **Lemma 5.** *Let* $H, \Delta \vDash V : \Gamma$, $\Sigma; \Gamma \left|\frac{q}{q'}\right. e : A$, *and* $F, V, H \,{}_{M}\!\vdash e \Downarrow (v, H') \mid (p, p')$. *It follows that* $\Phi_{V,H}(\Gamma) = \Phi_{V,H'}(\Gamma)$.

Proof. We get that $H' \vDash V : \Gamma$ by Theorem 1. Thus, the lemma follows directly by this fact and the definition of the potential Φ. ◀

5 Graphs and Graph Search

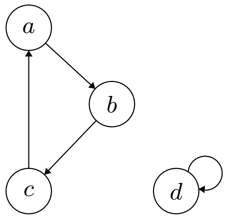
We now discuss graph search as an example of reference usage in RAML. Rather than explicitly showing occurrences of sharing, we use standard OCaml syntax.

We use the following set of user defined types as our graph representation.

```
type ‘a node = Not_Visited of ‘a * ‘a node ref list
             | Visited of ‘a * ‘a node ref list
             | TEMP

type ‘a graph = ‘a node ref list
```

Here, we represent a graph as an adjacency list, where each node contains some data, as well as a list of references to the nodes to which it has edges. We wrap each node by either `Not_Visited` or `Visited` so that we can consume the potential of the list of children and then place the node back in its reference. We now build the following graph using the function `make_graph`.



```
let make_graph () : int graph =
  let c_ref = ref (Node (Not_Visited (3, []))) in
  let b_ref = ref (Node (Not_Visited(2, [c_ref]))) in
  let a_ref = ref (Node (Not_Visited(1, [b_ref]))) in
  let _ = swap(c_ref, Node (Not_Visited(3, [a_ref]))) in
  let d_ref = ref (Node (Not_Visited (4, []))) in
  let _ = swap(d_ref, Node (Not_Visited (4, [d_ref]))) in
  [a_ref; b_ref; c_ref; d_ref]
```

We now show how to write DFS over this graph representation. In this implementation, we use the function `iter : ('a -> 1) -> 'a list -> ()` which sequentially applies a function to every element of a list. This implementation could easily be extended to apply a function to the data at each node, but here we simply traverse the graph.

At a not-yet-visited node in the graph, this function will iterate over its list of out-edges. For each, it will swap the node out of its particular reference and replace it with `TEMP`, recursively traverse that node and its respective out-edges, and then place the `Visited` version of that node back into the reference when it has completed its search on that subtree.

As a result, if the search finds TEMP within a reference, it must have been swapped out further up the current call stack and therefore must have already been visited.

```
let rec DFS(n : int node) : int node =
  match n with
  | TEMP -> n (* node was visited further up current call stack *)
  | Visited _ -> n (* node was visited in diff. branch of search *)
  | Not_Visited (d,cs) ->
    let _ = iter (fun n_ref ->
              let n' = swap(n_ref, TEMP) in
              let n'' = DFS n' in
              n_ref := n'') cs in
    Visited (d,cs)
```

We now consider tracking the number of recursive calls as a resource metric. Due to the structure of the node datatype, the list in a not-visited node is able to carry a different amount of potential than the list in a visited node. Therefore, we can define the type *node* as follows in order to pay for one full run of *DFS*.

$$A\ \mathit{node} = \mu X^{[0,0,0]}.\{\mathrm{Not\_Visited} : (A, L^1(X\ \mathit{ref})) \mid \mathrm{Visited} : (A, L^0(X\ \mathit{ref})) \mid \mathrm{TEMP} : 1\}\,.$$

Since the list contained within the Not_Visited constructor has a potential annotation of 1, each element can pay for one recursive call to DFS. When looking at the code, we see that this is exactly what occurs.

After running our DFS operation on every member of the list of nodes produced by make_graph, we have flipped each node to be marked with Visited and therefore have exhausted all the potential in each of the nodes. If we wish to refresh the graph and set each node back to Not_Visited with the intent to run our graph search again, we could once again iterate over the top-level list of nodes as returned by make_graph. However, if our program did this, then the potential annotation of the return type of make_graph would have to be increased, where our metric now tracks the number of times a node is changed from Visited to Not_Visited and vice versa. In summary, the type of make_graph is as follows (assuming we have some type *int*).

$$\mathit{make\_graph}\ (() : 1) : L^1(\mathit{int\ node}) \qquad\qquad \text{[With no refresh]}$$

$$\mathit{make\_graph}\ (() : 1) : L^3(\mathit{int\ node}) \qquad\qquad \text{[With one refresh]}$$

6 Type Inference for Linear and Polynomial Bounds and Higher-Order Functions

The main advantage of the resource-annotated type system we introduce here is that type inference can be reduced to efficient linear programming in the same way as for classic AARA for purely functional programs. This is also true for more complex polynomial type annotations [18].

The type inference works as follows. We first perform a standard, unification-based type inference for simple types. We then annotate the derivation tree with (yet unknown) potential annotations that will be determined by an LP solver. To generate the linear program, we apply the annotated type rules from Section 3 and collect the local constraints that need to hold for the annotations. We then minimize the initial potential using the LP solver.

To make the material more accessible, we have presented the potential annotations for references in the first-order setting. However, the proposed technique scales to higher-order programs and polynomial bounds [19].

Assume we would add references of ML type $T \equiv (T_1 \to T_2)$ *ref* to our language, which we can use to store first-order functions. In the annotated type system we would simply replace T with the annotated version $A \xrightarrow{q/q'} A$ *ref*. Like previous work [19], we can basically leave the rules L:Swap, L:Assign, L:DRef, and L:Ref unchanged; the same is true for the rules concerning array operations. This means that we fix a function annotation for each heap location that is referenced by a higher-order reference. Intuitively, every function that is stored in such a location would have to adhere to the resource behavior that is specified by the type. On the other hand, we can assume that a function has the specified worst-case behavior if we dereference a function value. Since we often need to use a function with different valid potential annotations, we implemented a generalization in RAML; functions are typed with a set of annotated function types $A \xrightarrow{q/q'} A$. However, the type rules remain conceptually identical.

Similarly, we do not have to alter the type rules for references and arrays when switching to univariate polynomial potential annotations [20]. In this scenario, we have potential annotations that are coefficients for more complex terms like $\binom{n}{k}$ rather than just the number of nodes n. Fortunately, `swap` does not alter the size of data structures, and therefore the rules do not have to be changed.

The situation is more complex when we consider multivariate polynomial potential annotations [18]. The difficulty is to decide how to handle mixed potential of the form $n{\cdot}m$ where m corresponds to the size of a list stored in a reference and n corresponds to the size of another data structure. Our current approach to this is to simply require that such a potential annotation is zero. This amounts to deriving multivariate bounds for data that is entirely stored within a single reference but not for data that is stored across multiple different references.

7 Related Work

Automatic amortized resource analysis (AARA) has been introduced to derive linear bounds on the number of heap allocations in a simple, strict, and first-order functional language [22]. Linear AARA has been extended to work with higher-order functions [27], object-oriented programs [23], lazy evaluation [37], imperative integer programs [9], pointer-based data structures [4], polynomial bounds [18], and term rewriting [25]. Many of the features for strict functional programs have been combined in Resource Aware ML [19]. In contrast to the presented technique, none of the aforementioned works allow the derivation of bounds that depend on the sizes of data structures that are stored in references and arrays.

Most closely related to our work is the treatment of references in recent work on RAML [21, 19]. Previous techniques allow references but statically ensure that the cost of computation does not depend on the sizes of data structures that have been retrieved from references and arrays. However, the higher-order system [19] derives bounds for programs whose cost depend on applications of functions that have been stored in and dereferenced from references and arrays. The innovation in this work is to allow the cost of computation to depend on data that is stored in references and arrays. There also exist AARAs for mutable heap data structures that are integrated in separation logic [4] and object-oriented type systems (in RAJA) [23, 26]. These type systems are somewhat incomparable but in many ways more expressive compared to the introduced system. The price they pay is that automatic bound inference is challenging and often only possible when user annotations are provided. An advantage of our method is that it can naturally and automatically derive bounds that depend on the size of cyclic data structures, as demonstrated in Section 5.

There are also other type-based approaches to bound analysis. They are based on linear dependent types [28, 29] and type annotations [12, 31]. Cicek et al. [11] study a type system for incremental complexity. Moreover, there are analyses for functional programs that are based on solving and deriving (potentially higher-order) recurrence relations [6, 13]. None of these analysis systems can deal with side-effects and most have only basic support for automation.

Another research direction is to apply techniques from term rewriting to complexity analysis [33, 8, 5]; sometimes in combination with amortized analysis [24]. However, existing techniques seem to be restricted to purely functional programs and time complexity.

Approaches to resource analysis based on abstract interpretation [1, 3, 7, 10, 15, 16, 35] focus on bounds that depend on integers. There are techniques that take into account mutable heap structures by abstracting their sizes with ghost variables [2]. However, it is unclear how well these techniques scale to data with cycles, as well as with nested data structures that are potentially stored in arrays.

8 Conclusion

We have extended automatic amortized resource analysis to determine bounds for programs whose resource consumption depends on data stored in mutable heap cells. Moreover, we have followed the design philosophy of the ML type system in the sense that we refrain from tracking effects within the type system itself. In order to track resource usage that depends on the size of data stored in mutable cells, we require that the potential annotations for the types of these cells are invariant during the execution. We used a primitive `swap` operation that allows us to make use of the potential in references and arrays. The type rules ensure that we "swap in" data with the same potential annotation when we extract potential from a memory cell. To prove the non-trivial soundness theorem, we have used memory typing, which allows us to track relevant heap cells while not falling victim to aliasing.

Our analysis preserves the benefits of AARA such as compositionality and reduction of type inference to linear constraint solving. However, our work is also a departure from previous work in the sense that we extend the programming language with a new construct that guides the resource bound analysis instead of purely focusing on making the analysis work well for existing code. The additional burden that we put on the programmer is balanced by an elegant, clear, and transparent type system. A user of our system only needs to remember one simple rule: if the resource usage of the program depends on the size of dereferenced data, then the data has to be dereferenced using `swap`.

References

1 Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. Cost Analysis of Java Bytecode. In *16th Euro. Symp. on Prog. (ESOP'07)*, pages 157–172, 2007.

2 Elvira Albert, Richard Bubel, Samir Genaim, Reiner Hähnle, and Guillermo Román-Díez. Verified Resource Guarantees for Heap Manipulating Programs. In *15th Int. Conf. on Fundamental Approaches to Software Engineering (FASE'12)*, pages 130–145, 2012.

3 Elvira Albert, Jesús Correas Fernández, and Guillermo Román-Díez. Non-cumulative Resource Analysis. In *Tools and Algorithms for the Construction and Analysis of Systems – 21st Int. Conf., (TACAS'15)*, pages 85–100, 2015.

4 Robert Atkey. Amortised Resource Analysis with Separation Logic. In *19th Euro. Symp. on Prog. (ESOP'10)*, pages 85–103, 2010.

5 Martin Avanzini, Ugo Dal Lago, and Georg Moser. Analysing the Complexity of Functional Programs: Higher-Order Meets First-Order. In *29th Int. Conf. on Functional Programming (ICFP'15)*, 2012.

6 Ralph Benzinger. Automated Higher-Order Complexity Analysis. *Theor. Comput. Sci.*, 318(1-2):79–103, 2004.

7 Régis Blanc, Thomas A. Henzinger, Thibaud Hottelier, and Laura Kovács. ABC: Algebraic Bound Computation for Loops. In *Logic for Prog., AI., and Reasoning – 16th Int. Conf. (LPAR'10)*, pages 103–118, 2010.

8 Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. Alternating Runtime and Size Complexity Analysis of Integer Programs. In *Tools and Alg. for the Constr. and Anal. of Systems – 20th Int. Conf. (TACAS'14)*, pages 140–155, 2014.

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

10 Pavol Cerný, Thomas A. Henzinger, Laura Kovács, Arjun Radhakrishna, and Jakob Zwirchmayr. Segment Abstraction for Worst-Case Execution Time Analysis. In *24th European Symposium on Programming (ESOP'15)*, pages 105–131, 2015.

11 Ezgi Çiçek, Deepak Garg, and Umut A. Acar. Refinement Types for Incremental Computational Complexity. In *24th European Symposium on Programming (ESOP'15)*, pages 406–431, 2015.

12 Nils Anders Danielsson. Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures. In *35th ACM Symp. on Principles Prog. Langs. (POPL'08)*, pages 133–144, 2008.

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

14 Antonio Flores-Montoya. Upper and Lower Amortized Cost Bounds of Programs Expressed as Cost Relations. In *Formal Methods – 21st International Symposium (FM'16)*, pages 254–273, 2016.

15 Antonio Flores-Montoya and Reiner Hähnle. Resource Analysis of Complex Programs with Cost Equations. In *Programming Languages and Systems – 12th Asian Symposiu (APLAS'14)*, pages 275–295, 2014.

16 Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *36th ACM Symp. on Principles of Prog. Langs. (POPL'09)*, pages 127–139, 2009.

17 Sumit Gulwani and Florian Zuleger. The Reachability-Bound Problem. In *Conf. on Prog. Lang. Design and Impl. (PLDI'10)*, pages 292–304, 2010.

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

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

20 Jan Hoffmann and Martin Hofmann. Amortized Resource Analysis with Polynomial Potential. In *19th European Symposium on Programming (ESOP'10)*, 2010.

21 Jan Hoffmann and Zhong Shao. Type-Based Amortized Resource Analysis with Integers and Arrays. In *12th International Symposium on Functional and Logic Programming (FLOPS'14)*, 2014.

22 Martin Hofmann and Steffen Jost. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *30th ACM Symp. on Principles of Prog. Langs. (POPL'03)*, pages 185–197, 2003.

23 Martin Hofmann and Steffen Jost. Type-Based Amortised Heap-Space Analysis. In *15th Euro. Symp. on Prog. (ESOP'06)*, pages 22–37, 2006. doi:10.1007/11693024_3.

24 Martin Hofmann and Georg Moser. Amortised Resource Analysis and Typed Polynomial Interpretations. In *Rewriting and Typed Lambda Calculi (RTA-TLCA;14)*, pages 272–286, 2014.

25 Martin Hofmann and Georg Moser. Multivariate Amortised Resource Analysis for Term Rewrite Systems. In *13th International Conference on Typed Lambda Calculi and Applications (TLCA'15)*, pages 241–256, 2015.

26 Martin Hofmann and Dulma Rodriguez. Efficient Type-Checking for Amortised Heap-Space Analysis. In *18th Conf. on Comp. Science Logic (CSL'09)*. LNCS, 2009.

27 Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. Static Determination of Quantitative Resource Usage for Higher-Order Programs. In *37th ACM Symp. on Principles of Prog. Langs. (POPL'10)*, pages 223–236, 2010.

28 Ugo Dal Lago and Marco Gaboardi. Linear Dependent Types and Relative Completeness. In *26th IEEE Symp. on Logic in Computer Science (LICS'11)*, pages 133–142, 2011.

29 Ugo Dal Lago and Barbara Petit. The Geometry of Types. In *40th ACM Symp. on Principles Prog. Langs. (POPL'13)*, pages 167–178, 2013.

30 Benjamin Lichtman and Jan Hoffmann. Arrays and references in resource aware ml. Technical report, Carnegie Mellon University, 2017.

31 Ravichandhran Madhavan, Sumith Kulal, and Viktor Kuncak. Contract-based resource verification for higher-order functions with memoization. In *44th Symposium on Principles of Programming Languages POPL'17*, 2017.

32 Greg Morrisett, Amal Ahmed, and Matthew Fluet. L3: A linear language with locations. In *Typed Lambda Calculi and Applications: 7th International Conference, TLCA 2005, Nara, Japan, April 21-23, 2005. Proceedings*, pages 293–307, 2005.

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

34 Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2004.

35 Moritz Sinn, Florian Zuleger, and Helmut Veith. A Simple and Scalable Approach to Bound Analysis and Amortized Complexity Analysis. In *Computer Aided Verification – 26th Int. Conf. (CAV'14)*, pages 743–759, 2014.

36 Frederick Smith, David Walker, and J. Gregory Morrisett. Alias types. In *Proceedings of the 9th European Symposium on Programming Languages and Systems*, ESOP'00, pages 366–381, London, UK, UK, 2000. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=645394.651903.

37 Pedro B. Vasconcelos, Steffen Jost, Mário Florido, and Kevin Hammond. Type-Based Allocation Analysis for Co-recursion in Lazy Functional Languages. In *24th European Symposium on Programming (ESOP'15)*, pages 787–811, 2015.

A Rules Defining the Big-Step Operational Semantics

We use $\overline{n}$ to denote the unary representation of the number $n \in \mathbb{N}$ (as defined in Section 3) for use in the rules for array operations. For example, $\overline{0} = \text{zero}$, $\overline{1} = \text{succ}(\text{zero})$, and so on.

$$\frac{V(x) = \ell}{F, V, H\,{}_{M}{\vdash} x \Downarrow (\ell, H) \mid M^{\text{var}}}\,(\text{E:Var}) \qquad \frac{}{F, V, H\,{}_{M}{\vdash} () \Downarrow (\text{Null}, H) \mid M^{\text{triv}}}\,(\text{E:Triv})$$

$$\frac{}{F, V, H\,{}_{M}{\vdash} \text{true} \Downarrow (\texttt{tt}, H) \mid M^{\text{true}}}\,(\text{E:True}) \qquad \frac{}{F, V, H\,{}_{M}{\vdash} \text{false} \Downarrow (\texttt{ff}, H) \mid M^{\text{false}}}\,(\text{E:False})$$

$$\frac{(f, \vec{y}, e_f) \in F \qquad F, [y_1 \mapsto V(x_1), \cdots, y_n \mapsto V(x_n)], H\,{}_{M}{\vdash} e_f \Downarrow (v, H') \mid (q, q')}{F, V, H\,{}_{M}{\vdash} f(x_1, \ldots, x_n) \Downarrow (v, H') \mid M_1^{\text{fun}_n} \cdot (q, q') \cdot M_2^{\text{fun}_n}}\,(\text{E:Fun})$$

$$\frac{F, V, H\,{}_{M}{\vdash} e_1 \Downarrow (v_1, H_1) \mid (q, q') \qquad F, V[x \mapsto v_1], H_1\,{}_{M}{\vdash} e_2 \Downarrow (v_2, H_2) \mid (p, p')}{F, V, H\,{}_{M}{\vdash} \text{let } x = e_1 \text{ in } e_2 \Downarrow (v_2, H_2) \mid M_1^{\text{let}} \cdot (q, q') \cdot M_2^{\text{let}} \cdot (p, p') \cdot M_3^{\text{let}}}\,(\text{E:Let})$$

$$\frac{c_i \in \text{CID} \qquad v = (\mathsf{constr}_{c_i}, V(x_1), \ldots, V(x_k)) \qquad H' = H, \ell \mapsto v \qquad \ell \notin \text{dom}(H)}{F, V, H\,{}_{M}{\vdash} c_i \, \langle x_1, \ldots, x_k \rangle \Downarrow (\ell, H') \mid M^{\text{cons}}}\,(\text{E:Constr})$$

$$\frac{H(V(x)) = (\mathsf{constr}_{c_i}, v_1, \ldots, v_k) \qquad F, V[x_1 \mapsto v_1, \ldots, x_k \mapsto v_k], H\,{}_{M}{\vdash} e_1 \Downarrow (v, H') \mid (q, q')}{F, V, H\,{}_{M}{\vdash} \text{match } x \text{ with } c_i \, \langle x_1, \ldots, x_k \rangle \Rightarrow e_1 \mid e_2 \Downarrow (v, H') \mid M_1^{\text{matT}} \cdot (q, q') \cdot M_2^{\text{matT}}}\,(\text{E:Mat1})$$

$$\frac{H(V(x)) \neq (\mathsf{constr}_{c_i}, v_1, \ldots, v_k) \qquad F, V, H\,{}_{M}{\vdash} e_2 \Downarrow (v, H') \mid (q, q')}{F, V, H\,{}_{M}{\vdash} \text{match } x \text{ with } c_i \, \langle x_1, \ldots, x_k \rangle \Rightarrow e_1 \mid e_2 \Downarrow (v, H') \mid M_1^{\text{matF}} \cdot (q, q') \cdot M_2^{\text{matF}}}\,(\text{E:Mat2})$$

$$\frac{V(x) = v' \qquad V' = V \setminus x \qquad F, V'[x_1 \mapsto v', x_2 \mapsto v'], H\,{}_{M}{\vdash} e \Downarrow (v, H') \mid (q, q')}{F, V, H\,{}_{M}{\vdash} \text{share } x \text{ as } (x_1, x_2) \text{ in } e \Downarrow (v, H') \mid (q, q')}\,(\text{E:Share})$$

$$\frac{H' = H, \ell \mapsto V(x) \qquad \ell \notin \text{dom}(H)}{F, V, H\,{}_{M}{\vdash} \text{ ref } x \Downarrow (\ell, H') \mid M^{\text{ref}}}\,(\text{E:Ref}) \qquad \frac{\ell = H(V(x_1)) \qquad H' = H[V(x_1) \mapsto V(x_2)]}{F, V, H\,{}_{M}{\vdash} \text{swap}(x_1, x_2) \Downarrow (\ell, H') \mid M^{\text{swap}}}\,(\text{E:Swap})$$

$$\frac{\ell = H(V(x))}{F, V, H\,{}_{M}{\vdash} \, !x \Downarrow (\ell, H) \mid M^{\text{dref}}}\,(\text{E:DRef}) \qquad \frac{H' = H[V(x_1) \mapsto V(x_2)]}{F, V, H\,{}_{M}{\vdash} x_1 := x_2 \Downarrow (\text{Null}, H) \mid M^{\text{assign}}}\,(\text{E:Assign})$$

$$\frac{\begin{array}{c} H(V(x)) = \overline{n} \qquad F, V, H\,{}_{M}{\vdash} e \Downarrow (v, H') \mid (q, q') \\ H'' = H', \ell \mapsto (\sigma, n), \ell_1 \mapsto v, \ldots, \ell_n \mapsto v \qquad \forall i : \sigma(i) = \ell_{i+1} \qquad \ell, \ell_1, \ldots, \ell_n \notin \text{dom}(H) \end{array}}{F, V, H\,{}_{M}{\vdash} \text{create}(x, e) \Downarrow (\ell, H'') \mid M^{\text{create}} \cdot (q, q')^n}\,(\text{E:Create})$$

$$\frac{H(V(x_1)) = (\sigma, n) \qquad H(V(x_2)) = \overline{i} \qquad 0 \leq i < n}{F, V, H\,{}_{M}{\vdash} \text{get}(x_1, x_2) \Downarrow (H(\sigma(i)), H) \mid M^{\text{get}}}\,(\text{E:Get})$$

$$\frac{H(V(x_1)) = (\sigma, n) \qquad H(V(x_2)) = \overline{i} \qquad 0 \leq i < n \qquad H' = H[\sigma(i) \mapsto V(x_3)]}{F, V, H\,{}_{M}{\vdash} \text{set}(x_1, x_2, x_3) \Downarrow (\text{Null}, H') \mid M^{\text{set}}}\,(\text{E:Set})$$

$$\frac{\begin{array}{c} H(V(x_1)) = (\sigma, n) \qquad H(V(x_2)) = \overline{i} \qquad 0 \leq i < n \\ v = H(\sigma(i)) \qquad H' = H[\sigma(i) \mapsto V(x_3)] \end{array}}{F, V, H\,{}_{M}{\vdash} \text{aswap}(x_1, x_2, x_3) \Downarrow (v, H') \mid M^{\text{aswap}}}\,(\text{E:Aswap})$$

$$\frac{H(V(x)) = (\sigma, n) \qquad H' = H, \ell \mapsto \overline{n} \qquad \ell \notin \text{dom}(H)}{F, V, H\,{}_{M}{\vdash} \text{length}(x) \Downarrow (\ell, H') \mid M^{\text{len}}}\,(\text{E:Len})$$

B Rules Defining the Simple Typing Judgement

$$\frac{}{\Sigma;\emptyset \vdash \mathsf{true} : B}\ (\text{T:True}) \qquad \frac{}{\Sigma;\emptyset \vdash \mathsf{false} : B}\ (\text{T:False})$$

$$\frac{}{\Sigma;\emptyset \vdash () : 1}\ (\text{T:Triv}) \qquad \frac{}{\Sigma; x{:}T \vdash x : T}\ (\text{T:Var})$$

$$\frac{n \in \mathbb{N}}{\Sigma;\emptyset \vdash n : N}\ (\text{T:Nat}) \qquad \frac{\Sigma(f) = (T_1,\ldots,T_n) \to T}{\Sigma;\Gamma, x_1{:}T_1,\ldots,x_n{:}T_n \vdash f(x_1,\ldots,x_n) : T}\ (\text{T:Fun})$$

$$\frac{\Sigma;\Gamma_1 \vdash e_1 : T' \qquad \Sigma;\Gamma_2, x{:}T' \vdash e_2 : T}{\Sigma;\Gamma_1,\Gamma_2 \vdash \text{let } x = e_1 \text{ in } e_2 : T}\ (\text{T:Let})$$

$$\frac{T = \mu X.\{\cdots \mid c_i{:}(T'_1,\ldots,T'_k) \mid \cdots\} \qquad \begin{array}{c} c \in \mathsf{CID} \\ \forall 1 \le j \le k. T_j = T'_j \vee (T_j = T \wedge T'_j = X)\end{array}}{\Sigma; x_1{:}T_1,\ldots,x_k{:}T_k \vdash c_i\ (x_1,\ldots,x_k) : T}\ (\text{T:Constr})$$

$$\frac{\begin{array}{c} c \in \mathsf{CID} \qquad T' = \mu X.\{\cdots \mid c_i{:}(T'_1,\ldots,T'_k) \mid \cdots\} \\ \Sigma;\Gamma, x_1{:}[T'/X]T'_1,\ldots,x_k{:}[T'/X]T'_k \vdash e_1 : T \qquad \Sigma;\Gamma, x{:}T' \vdash e_2 : T \end{array}}{\Sigma;\Gamma, x : T' \vdash \mathsf{match}\ x\ \mathsf{with}\ c_i\ \langle x_1,\ldots,x_k\rangle \Rightarrow e_1 \mid e_2 : T}\ (\text{T:Mat})$$

$$\frac{\Sigma;\Gamma \vdash e : T}{\Sigma;\Gamma, x{:}T' \vdash e : T}\ (\text{T:Weaken}) \qquad \frac{\Sigma;\Gamma, x_1{:}T', x_2{:}T' \vdash e : T}{\Sigma;\Gamma, x{:}T' \vdash \mathsf{share}\ x\ \mathsf{as}\ (x_1,x_2)\ \mathsf{in}\ e : T}\ (\text{T:Share})$$

$$\frac{}{\Sigma; x{:}T \vdash \text{ref } x : T \text{ ref}}\ (\text{T:Ref}) \qquad \frac{}{\Sigma; x{:}T \text{ ref} \vdash !x : T}\ (\text{T:DRef})$$

$$\frac{}{\Sigma; x_1{:}T \text{ ref}, x_2{:}T \vdash x_1 := x_2 : 1}\ (\text{T:Assign})$$

$$\frac{}{\Sigma; x_1{:}T \text{ ref}, x_2{:}T \vdash \mathsf{swap}(x_1,x_2) : T}\ (\text{T:Swap})$$

$$\frac{}{\Sigma; x_1{:}N, x_2{:}T \vdash \mathsf{create}(x_1,x_2) : T \text{ array}}\ (\text{T:Create})$$

$$\frac{}{\Sigma; x_1{:}T \text{ array}, x_2{:}N \vdash \mathsf{get}(x_1,x_2) : T}\ (\text{T:Get})$$

$$\frac{}{\Sigma; x_1{:}T \text{ array}, x_2{:}N, x_3{:}T \vdash \mathsf{set}(x_1,x_2,x_3) : 1}\ (\text{T:Set})$$

$$\frac{}{\Sigma; x_1{:}T \text{ array}, x_2{:}N, x_3{:}T \vdash \mathsf{aswap}(x_1,x_2,x_3) : T}\ (\text{T:Aswap})$$

$$\frac{}{\Sigma; x{:}T \text{ array} \vdash \mathsf{length}(x) : N}\ (\text{T:Len})$$

C Rules Defining the Annotated Typing Judgement

$$\dfrac{}{\Sigma;\emptyset \vdash^{q+M^{\mathrm{true}}}_{q} \mathrm{true} : B}\,(\mathrm{L{:}True}) \qquad \dfrac{}{\Sigma;\emptyset \vdash^{q+M^{\mathrm{false}}}_{q} \mathrm{false} : B}\,(\mathrm{L{:}False}) \qquad \dfrac{}{\Sigma;\emptyset \vdash^{q+M^{\mathrm{triv}}}_{q} () : 1}\,(\mathrm{L{:}Triv})$$

$$\dfrac{}{\Sigma; x{:}A \vdash^{q+M^{\mathrm{var}}}_{q} x : A}\,(\mathrm{L{:}Var}) \qquad \dfrac{(A_1,\ldots,A_n) \xrightarrow{q/q'} A \in \Sigma(f)}{\Sigma; x_1{:}A_1,\ldots,x_n{:}A_n \vdash^{q+M_1^{\mathrm{fun}_n}}_{q'-M_2^{\mathrm{fun}_n}} f(x_1,\ldots,x_n) : A}\,(\mathrm{L{:}Fun})$$

$$\dfrac{}{\Sigma;\emptyset \vdash^{q+M^{\mathrm{nat}}}_{q} n : N}\,(\mathrm{L{:}Nat}) \qquad \dfrac{\Sigma;\Gamma_1 \vdash^{q-M_1^{\mathrm{let}}}_{p} e_1 : A' \qquad \Sigma;\Gamma_2, x{:}A' \vdash^{p-M_2^{\mathrm{let}}}_{q'+M_3^{\mathrm{let}}} e_2 : A}{\Sigma;\Gamma_1,\Gamma_2 \vdash^{q}_{q'} \mathrm{let}\ x = e_1\ \mathrm{in}\ e_2 : A}\,(\mathrm{L{:}Let})$$

$$\dfrac{\begin{array}{c} c \in \mathrm{CID} \\ A = \mu X^P.\{\cdots \mid c_i{:}(A'_1,\ldots,A'_k) \mid \cdots\} \qquad \forall 1 \le j \le k.\, A_j = A'_j \vee (A_j = A \wedge A'_j = X) \end{array}}{\Sigma; x_1{:}A_1,\ldots,x_k{:}A_k \vdash^{q+P[i]+M^{\mathrm{cons}}}_{q} c_i\ (x_1,\ldots,x_k) : A}\,(\mathrm{L{:}Constr})$$

$$\dfrac{\begin{array}{c} c \in \mathrm{CID} \qquad A' = \mu X^P.\{\cdots \mid c_i{:}(A'_1,\ldots,A'_k) \mid \cdots\} \\ \Sigma;\Gamma, x_1{:}[A'/X]A'_1,\ldots,x_k{:}[A'/X]A'_k \vdash^{q+P[i]-M_1^{\mathrm{matT}}}_{q'+M_2^{\mathrm{MatT}}} e_1 : A \qquad \Sigma;\Gamma, x{:}A' \vdash^{q-M_1^{\mathrm{matF}}}_{q'+M_2^{\mathrm{matF}}} e_2 : A \end{array}}{\Sigma;\Gamma, x : A' \vdash^{q}_{q'} \mathrm{match}\ x\ \mathrm{with}\ c_i\ \langle x_1,\ldots,x_k\rangle \Rightarrow e_1 \mid e_2 : A}\,(\mathrm{L{:}Mat})$$

$$\dfrac{\Sigma;\Gamma \vdash^{q}_{q'} e : A}{\Sigma;\Gamma, x{:}A' \vdash^{q}_{q'} e : A}\,(\mathrm{L{:}Weaken}) \qquad \dfrac{\Sigma;\Gamma, x_1{:}A_1, x_2{:}A_2 \vdash^{q}_{q'} e : A \qquad A' \curlyvee (A_1,A_2)}{\Sigma;\Gamma, x{:}A' \vdash^{q}_{q'} \mathrm{share}\ x\ \mathrm{as}\ (x_1,x_2)\ \mathrm{in}\ e : A}\,(\mathrm{L{:}Share})$$

$$\dfrac{\Sigma;\Gamma \vdash^{p}_{p'} e : A \qquad q \ge p \qquad q - p \ge q' - p'}{\Sigma;\Gamma \vdash^{q}_{q'} e : A}\,(\mathrm{L{:}Relax}) \qquad \dfrac{}{\Sigma; x{:}A \vdash^{q+M^{\mathrm{ref}}}_{q} \mathrm{ref}\ x : A\ \mathrm{ref}}\,(\mathrm{L{:}Ref})$$

$$\dfrac{|A'| = |A| \qquad A \curlyvee (A,A)}{\Sigma; x{:}A'\ \mathrm{ref} \vdash^{q+M^{\mathrm{dref}}}_{q} !x : A}\,(\mathrm{L{:}DRef}) \qquad \dfrac{}{\Sigma;\Gamma, x_1{:}A\ \mathrm{ref}, x_2{:}A \vdash^{q+M^{\mathrm{assign}}}_{q} x_1 := x_2 : 1}\,(\mathrm{L{:}Assign})$$

$$\dfrac{}{\Sigma; x{:}A\ \mathrm{array}^p \vdash^{q+M^{\mathrm{len}}}_{q} \mathrm{length}(x) : N^p}\,(\mathrm{L{:}Len}) \qquad \dfrac{}{\Sigma; x_1{:}A\ \mathrm{ref}, x_2{:}A \vdash^{q+M^{\mathrm{swap}}}_{q} \mathrm{swap}(x_1,x_2) : A}\,(\mathrm{L{:}Swap})$$

$$\dfrac{\Sigma;\cdot \vdash^{p-r}_{0} e : A}{\Sigma; x{:}N^p \vdash^{q+M^{\mathrm{create}}}_{q} \mathrm{create}(x,e) : A\ \mathrm{array}^r}\,(\mathrm{L{:}Create})$$

$$\dfrac{|A'| = |A| \qquad A \curlyvee (A,A)}{\Sigma; x_1{:}A'\ \mathrm{array}^p, x_2{:}N^r \vdash^{q+M^{\mathrm{get}}}_{q} \mathrm{get}(x_1,x_2) : A}\,(\mathrm{L{:}Get})$$

$$\dfrac{}{\Sigma; x_1{:}A\ \mathrm{array}^p, x_2{:}N^r, x_3{:}A \vdash^{q+M^{\mathrm{set}}}_{q} \mathrm{set}(x_1,x_2,x_3) : 1}\,(\mathrm{L{:}Set})$$

$$\dfrac{}{\Sigma; x_1{:}A\ \mathrm{array}^p, x_2{:}N^r, x_3{:}A \vdash^{q+M^{\mathrm{aswap}}}_{q} \mathrm{aswap}(x_1,x_2,x_3) : A}\,(\mathrm{L{:}Aswap})$$