Arrays and References in Resource Aware ML

Benjamin Lichtman and Jan Hoffmann

Carnegie Mellon University

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 resources bounds. Due to its type-based nature, 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 type derivations prove 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 cannot derive bounds for programs whose resource consumption is dependent 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 the notion of a reference collection, which gathers all unique locations pointed to by a reference in a given program. Apart from the design of the system, the main contribution is the proof of the soundness of the extended analysis system.

1 Introduction

Quantitative properties such as memory usage and execution time are often key to the success or failure of a given piece of software. Traditionally, when precise non-asymptotic bounds are required, this information must be derived manually or through testing. This process is error-prone and tedious, especially in programs that evolve over time. As a result, many approaches to automatic resource analysis have been studied.

There exist several tools that can automatically derive loop and recursion bounds for imperative programs including SPEED [20], KoAT [9], PUBS [1, 3], Rank [4], and LOOPUS [38, 41]. 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 existing approach to deal with the problem is to represent 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 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 [12, 14, 34, 39], recurrence relations [7, 15, 16, 18], and automatic amortized resource analysis (AARA) [10, 22, 24, 27, 31]. 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 [26] that is implement in Resource Aware ML (RAML) [24]. While this analysis works well for higher-order programs that store functions in mutable heap structures, it cannot derive bounds for programs with arrays or references if the execution depends on the size of data structures that are stored in mutable structure.

In this article, we propose an extension of RAML for automatically deriving symbolic resource bounds that depend on the sizes of data that is 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, 24, 27]. To achieve *compositionality*, the analysis is integrated with a standard type system. To achieve *scalability*, type inference is reduced to efficient linear constraint solving. Type derivations can be used as certificates that proof the correctnes of the bound. The technique is also parametric in the resource of interest 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 [5] or object-oriented types [28]. The 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-annoted types mirror the design desigions of the ML type system, which is the basis of RAML. The disadvantage is that it is less expressive and requires a certain style of programming. However, the advantages that we get from the automation seems 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 small language with a subset of the features RAML and for linear bounds. As we argue in Section 7, the main ideas carry over to the polynomial and higher-order setting. To enable data that has been stored in mutable heap cells to be considered in resource analysis, we introduce the `swap` operation, which restricts such data to be explicitly overwritten on retrieval. 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 introduce the notion of a *reference collection*. This reference collection is essential to show the soundness of operations like `swap` that involve mutable heap cells.

2 Informal Account

In this section, we briefly introduce type-based amortized resource analysis. We then motivate and describe our contributions. For the examples in this section we use the number of uses of *cons* as a metric; however, the contributions presented here are fully agnostic with respect to resource metrics.

Automatic Amortized 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.

For the purpose of the examples in this section, we use the type $L^q(T)$, which represents a list containing elements of type T, where each element carries q potential. The potential annotation of a type directly determines the potential represented by values of that type. When we consider a list of length n and of type $L^q(T)$, then the potential carried by that list is qn, where that each element carries q resource units. With the resource metric considered here, since each element carries q resource units, we are able to pay for up to q *cons* operations for each element in the list.

On this list type, we can now consider the *append* function with the standard efficient implementation. As this implementation calls *cons* on every element of its first argument once, 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 from every element of the first input list, 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 explicitly 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 l_1 and l_2 have type $L^1(T)$.

Arbitrary Potential With Side Effects We now introduce references by way of the following example, where the function g' uses 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 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 ensuring 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 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 potential could vary within these references, then the system would have to track all possible combinations of distinct and aliased references in these situations. As this clearly cannot scale, we resort to demanding invariant potential in 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 potential of the underlying type (as we do when sharing a list directly) due to our new requirement of invariant potential. 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, 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 the call to `share`, but due to our invariant potential in references, it does not in this case. Therefore, our program uses double the amount of potential that the type annotations provide.

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 [35, 37]. 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 carries the same potential. As a result, for data in a single reference to be used twice, it must be swapped out, explicitly shared, and swapped back in.

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. Consider

the following program representing this situation, in which we append two lists that have been stored in references that are possibly aliased to each other.

```
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 consequently 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 ML operator. However, we ensure in our type system that the retrieved data has no potential. 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 the type of a function contain information about aliasing. One possible typing for the function f is $(L^1(T)\ \textit{ref}, L^1(T)\ \textit{ref}) \to L^0(T)$. Here, $r1$ and $r2$ are potentially aliased. Another possible typing for f is $(L^1(T)\ \textit{ref}, L^0(T)\ \textit{ref}) \to L^0(T)$. This typing implies that the arguments are not aliased; if they were, they would break an invariant of our type system.

The concerns and techniques presented above extend cleanly from references to arrays. As before, for each cell in an array, we cannot tell statically how much potential has been used up. Therefore, we maintain the same requirement of invariant potential for every cell in the array. We now consider an example of this phenomenon in which we suppose an integer type *int*, a function $\textit{sum} : L^1(\textit{int}) \to \textit{int}$ which sums the elements in a list, and a function $\textit{iterlist} : (L^q(T) \to 1) \to L^q(T)\ \textit{array} \to 1$ for any type T that applies a function with no return value to the list contained in each cell of an array.

```
h A = let r = ref 0 in
  let _ = iterlist (fun l -> r := !r + (sum l)) A in !r
```

When considering the implementation of *iterlist*, it could either destructively or non-destructively manipulate the input array. In the former case, it could pull out the data in every cell and replace it with an empty list with sufficient potential. However, any subsequent use of the array would not be able to operate

on the same data, as it will have been consumed. Below, we implement the latter case using Resource Aware ML syntax.

```
type (‘a,‘b) swapper = Left of ‘a | Right of ‘b | TEMP

apply (f,ref_l) =
  match ref_l with
  | Left l  -> share l as (l1,l2) in
               let _ = f l1 in Right l2
  | Right l -> ref_l
  | TEMP    -> ref_l

iterlist_help f A i =
  share A as (A1,A2) in
  share i as (i1,i2) in
  share i2 as (i3,i4) in
  match i1 with
  | zero   ->  let ref_l = aswap(A1,i3,TEMP) in
               let ref_l' = apply (f,ref_l) in
               let _ = set(A2,i4,ref_l') in ()
  | succ i' -> share A1 as (A',A1') in
               let _ = iterlist_help f A' i' in
               let ref_l = aswap(A1',i3,TEMP) in
               let ref_l' = apply (f,ref_l) in
               let _ = set(A2,i4,ref_l') in ()

iterlist f A =
  share A as (A1,A2) in
  let n = len A1 in
  match n with
  | zero -> ()
  | succ i -> iterlist_help f A2
```

In order for each list to be placed back into an array cell after being swapped out, it must be wrapped in an extra algebraic datatype, called *swapper* here. If some potential-carrying data is removed from a cell of an array, it must be explicitly shared to be both used and placed back for later use as is done in the *apply* function. However, any such sharing operation will necessarily decrease the amount of potential it carries, and therefore the data being placed back into the cell will not have the same potential annotations, thereby breaking our invariant. If we use an algebraic datatype, we can assign different potential annotations to the different constructors it defines, and consequently the data can be safely placed back into the mutable cell. For example, in the case where the iterating function is the sum function above, and we track the metric of calls to *plus*, the list in each cell will need a potential annotation of 1 in order to pay for the sum operation. If we initialize the array so that every cell contains a list wrapped in the *Left* constructor, the function *apply* is assigned the following type.

$$\mathit{apply} : ((L^1(T) \to 1), ((L^1(T), L^0(T))\ \mathit{swapper})) \to (L^1(T), L^0(T))\ \mathit{swapper}$$

Clearly, this extra layer adds an extra unwrapping step for the client attempting to use the data, but is necessary in order to safely swap the same data in and out of a mutable cell.

3 A Simple Language with References and Arrays

We now present our analysis for a minimal first-order functional language that only contains the features that are relevant to our contributions. However, we designed these features to be implemented in Resource Aware ML (RAML) [23], which also includes (signed) integers, conditionals, and higher-order functions.

Syntax

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

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

Let $x \in \text{VID}$ as usual, where VID is the set of all variable identifiers, let $c \in \text{CID}$, where CID is the set of all constructor identifiers, and let $f \in \text{FID}$ where FID is the set of all function identifiers. Note that these expressions are in share-let normal form; that is, variable identifiers are used as opposed to arbitrary expressions wherever possible in order to simplify the semantics of the language. Due to this restriction, we use the sharing expression `share` x `as (`x_1`,` x_2`) in` e to mean that the value of the free variable x is bound to the variables x_1 and x_2 for use in e.

Simple Types

$$\begin{aligned} T ::= {} & 1 \mid B \mid X \mid \mu X.\{c_i : (T_1, \ldots, T_{k_i})\}_i \mid T \text{ ref} \mid T \text{ array} \\ R ::= {} & (T, \ldots, T) \to T \end{aligned}$$

Let $\mathscr{T}$ be the set of simple data types and let $\mathscr{R}$ be the set of simple first-order types as defined above. We also define the type $N = \mu X.\{\text{zero} : 1 \mid \text{succ} : X\}$ to

represent natural numbers. To each $T \in \mathscr{T}$ we assign a set of semantic values $[\![T]\!]$; for the derived type N, we assign integers as semantic values, but allow them to be matched against like standard recursive types.

Let the *typing context* $\Gamma\colon \mathrm{VID} \rightharpoonup \mathscr{T}$ be partial and finite, and define the union of typing contexts to be Γ_1, Γ_2 such that $\Gamma_1 \cap \Gamma_2 = \emptyset$. Furthermore, let the *signature* $\Sigma\colon \mathrm{FID} \rightharpoonup \mathscr{R}$ be partial and finite.

We now define the typing judgement $\Sigma; \Gamma \vdash e : T$ to mean that the expression e has type T under the signature Σ in context Γ. An expression e is well-typed in Γ under Σ iff $\Sigma; \Gamma \vdash e : T$ holds for some type T. The rules for the judgement are standard and are given in Appendix A.

We therefore define a well-typed program to consist of a signature Σ and a family $F = (f, \boldsymbol{x}, e_f)_{f \in \mathrm{dom}(\Sigma)}$ of function identifiers $f \in \mathrm{FID}$ with variable identifiers $\boldsymbol{x} = x_1, \ldots, x_n \in \mathrm{VID}$ and expression e_f such that if $\Sigma(f) = (T_1, \ldots, T_n) \to T$ then $\Sigma; x_1{:}T_1, \ldots, x_n{:}T_n \vdash e_f : T$.

Big-Step Operational Semantics Let *Loc* be an infinite set of locations modeling memory locations in a heap and define the set of values $v \in \mathit{Val}$ to be

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

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

We further define a *heap* to be a finite partial mapping $H\colon \mathit{Loc} \rightharpoonup \mathit{Val}$ that maps locations to values, an *environment* to be a finite partial mapping $V\colon \mathrm{VID} \rightharpoonup \mathit{Val}$ from variable identifiers to values, and a *resource metric* to be a mapping $M\colon K \times \mathbb{N} \to \mathbb{Q}$ that defines the resource consumption in each evaluation step of the big-step semantics where K is a set of constants. We write M_n^k for $M(k, n)$ and M^k for $M(k, 0)$. Note that M is not restricted to be finite, as M_n^{app} is defined for any $n \in \mathbb{N}$ to handle arbitrary numbers of function arguments.

To formalize the notion of tracking resources becoming available during evaluation, we define the *watermark* of the resource usage to be the maximal number of resource units under a given metric that are simultaneously used during an evaluation.

The operational evaluation rules in Figure 1 define the evaluation judgement

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

where, given a family of functions F, environment $V\colon \mathrm{VID} \rightharpoonup \mathit{Val}$, 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, leaving $q' \in \mathbb{Q}_0^+$ resource units available after evaluation. Moreover, the actual resource consumption is $\delta = q - q'$; note that δ is negative if more resources become available than are consumed during the execution of e.

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

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

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

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

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

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

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

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

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

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

$$\frac{H(V(x_1)) = n \qquad \forall i : \sigma(i) = V(x_2) \qquad H' = H, \ell \mapsto (\sigma, n) \qquad \ell \notin \mathrm{dom}(H)}{F, V, H_M \vdash \mathrm{create}(x_1, x_2) \Downarrow (\ell, H') \mid M_n^{\mathrm{create}}}\,(\text{E:Create})$$

$$\frac{n \in \mathbb{N}}{F, V, H_M \vdash n \Downarrow (n, H) \mid M^{\mathrm{nat}}}\,(\text{E:Nat}) \qquad \frac{H(V(x_1)) = (\sigma, n) \qquad H(V(x_2)) = i \qquad 0 \leq i < n}{F, V, H_M \vdash \mathrm{get}(x_1, x_2) \Downarrow (\sigma(i), H) \mid M^{\mathrm{get}}}\,(\text{E:Get})$$

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

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

$$\frac{H(V(x)) = (\sigma, n)}{F, V, H_M \vdash \mathrm{length}(x) \Downarrow (n, H) \mid M^{\mathrm{len}}}\,(\text{E:Len})$$

Fig. 1. Rules defining the big-step operational semantics.

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') by

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

For $q \in \mathbb{Q}$, we use $q \geq 0$ to denote $(q, 0)$ and $q < 0$ to denote $(0, -q)$ to support the codomain of M without case distinctions.

The following facts derived from the definition of $(q, q') \cdot (p, p')$ will be used in future proofs.

Proposition 3.01 *Let* $(q, q') = (r, r') \cdot (s, s')$.

1. $q \geq r$ *and* $q - q' = r - r' + s - s'$
2. *If* $(p, p') = (\overline{r}, r') \cdot (s, s')$ *and* $\overline{r} \geq r$ *then* $p \geq q$ *and* $p' = q'$
3. *If* $(p, p') = (r, r') \cdot (\overline{s}, s')$ *and* $\overline{s} \geq s$ *then* $p \geq q$ *and* $p' \leq q'$.
4. $(r, r') \cdot ((s, s') \cdot (t, t')) = ((r, r') \cdot (s, s')) \cdot (t, t')$

Note that Proposition 1.3.1.4 shows that the pairs (q, q') form a monoid $\mathscr{Q}$ with identity element $(0, 0)$. If resources are never restituted (as in the case of time), we can restrict ourselves to elements of the form $(q, 0)$ where $(q, 0) \cdot (p, 0) = (q+p, 0)$. The values of the constants $M_i^x \in \mathbb{Q}$ depend on the resource being measured, the implementation, and the system architecture.

We note, however, that in contrast to earlier works, heap cells may be changed during an evaluation after they are allocated. While invariance of heap cells is used in previous work, this property is not necessarily needed, and in fact, this difference drives the primary source of novelty for this work.

Well-Formed Environments 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. The judgement is defined by the rules in Figure 2.

We now denote that a 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 $x \in \mathrm{dom}(\Gamma)$.

To make certain later proofs clearer, we now show that under any evaluation with a well-formed environment, every location in the environment remains well-typed. We include the proof for Theorem 3.01 in Appendix B.

Theorem 3.01 *If* $\Sigma; \Gamma \vdash e : T$, $H \vDash V : \Gamma$, *and* $F, V, H_M \vdash e \Downarrow (v, H') \mid (q, q')$ *then* $H' \vDash V : \Gamma$.

We now show that the evaluation of a well-typed expression in a well-formed environment results in a well-formed environment. We include the proof for Theorem 3.02 in Appendix C.

Theorem 3.02 *If* $\Sigma; \Gamma \vdash e : T$, $H \vDash V : \Gamma$, *and* $F, V, H_M \vdash e \Downarrow (v, H') \mid (q, q')$ *then* $H' \vDash V : \Gamma$ *and* $H' \vDash v : T$.

$$\frac{}{H \vDash \texttt{tt} : B}\text{(V:True)} \qquad \frac{}{H \vDash \texttt{ff} : B}\text{(V:False)} \qquad \frac{}{H \vDash \text{Null} : 1}\text{(V:Unit)}$$

$$\frac{\begin{array}{c} v \in \mathit{Loc} \qquad H(v) = (\mathsf{constr}_{c_i}, v_1, \ldots, v_{k_i}) \qquad H' = H \setminus v \\ T = \mu X.\{\cdots \mid c_i{:}(T_1, \ldots, T_{k_i}) \mid \cdots\} \qquad H' \vDash v_1 : [T/X]T_1 \ \ldots \ H' \vDash v_{k_i} : [T/X]T_{k_i} \end{array}}{H \vDash v : \mu X.\{\cdots \mid c_i{:}(T_1, \ldots, T_{k_i}) \mid \cdots\}}\text{(V:Constr)}$$

$$\frac{n \in \mathbb{N}}{H \vDash n : N}\text{(V:Nat)} \qquad \frac{v \in \mathit{Loc} \qquad H(v) = v' \qquad H \vDash v' : T}{H \vDash v : \text{T ref}}\text{(V:Ref)}$$

$$\frac{v \in \mathit{Loc} \qquad H(v) = (\sigma, n) \qquad \forall i < n. H \vDash \sigma(i) : T}{H \vDash v : T \text{ array}}\text{(V:Array)}$$

Fig. 2. Rules relating heap cells to the types of their underlying semantic values.

4 Annotated Types for Linear Potential

We now describe type annotations, which give rise to the potential functions that serve as the foundation for our analysis. Furthermore, we introduce the reference collection, which allows for analysis of data in mutable heap cells in the presence of aliasing.

Resource Annotations To facilitate the analysis system of 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.

$$A ::= 1 \mid B \mid X \mid \mu X^{Q}.\{c_i : (T_1, \ldots, T_{k_i})\}_i \mid A \text{ ref} \mid A \text{ array}$$

Let $\mathscr{A}_{\text{lin}}$ be the set of linear resource-annotated data types. To each $A \in \mathscr{A}_{\text{lin}}$ we assign a set of semantic values $[\![A]\!]$. We again define the type $N = \mu X.\{\text{zero} : 1 \mid \text{succ} : X\}$ and assign it integers as semantic values. We extend all other definitions for simple data types to resource-annotated data types.

Let $A \in \mathscr{A}_{\text{lin}}$, let H be a heap, and let $v \in \mathit{Val}$ be a value with $H \vDash v : A$. We now define the potential $\Phi_H(v : A)$ as follows:

1. $\Phi_H(v : A) = 0$ if $v = \text{Null}$ or if $A \in \{1, B, C \text{ ref}, C \text{ array}\}$ for any annotated type C.
2. $\Phi_H(\ell : A) = Q[i] + \sum_{i=1}^{k_i} \Phi_H(v_i : [A/X]A_i)$ if $H(\ell) = (\mathsf{constr}_{c_i}, v_1, \ldots, v_{k_i})$ and $A = \mu X^{Q}.\{\cdots \mid c_i : (A_1, \ldots, A_{k_i}) \mid \ldots\}$.

Underlying Types In order to relate resource-annotated types with different resource annotations, we define the function $|\cdot| : \mathscr{A}_{\text{lin}} \to \mathscr{T}$, defined as follows:

$$
\begin{aligned}
|1| &= 1 \\
|B| &= B \\
|X| &= X \\
|\mu X^Q.\{c_i : (A_1, \ldots, A_{k_i})\}_i| &= \mu X.\{c_i : (|A_1|, \ldots, |A_{k_i}|)\}_i \\
|A \text{ ref}| &= |A| \text{ ref} \\
|A \text{ array}| &= |A| \text{ array}
\end{aligned}
$$

If we have types T and T' such that $|T| = |T'|$, 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 holds.

$$
\frac{C \in \{1, B, N, C' \text{ ref}, C' \text{ array}\}}{C \curlyvee (C, C)} \, (S_1)
$$

$$
\frac{A_i \curlyvee (A'_i, A''_i) \qquad \forall 1 \leq i \leq k \; P[i] = Q[i] + R[i]}{\mu X^P.\{c_i : (A_1, \ldots, A_k)\}_i \curlyvee (\mu X^Q.\{c_i : (A'_1, \ldots, A'_k)\}_i, \mu X^R.\{c_i : (A''_1, \ldots, A''_k)\}_i)} \, (S_2)
$$

Lemma 1. *Let A, A', and A'' be resource-annotated data types with $A \curlyvee (A', A'')$. Then $|A| = |A'| = |A''|$ and for every heap H and value v such that $H \vDash v : A$, $\Phi_H(v : A) = \Phi_H(v : A') + \Phi_H(v : A'')$ holds.*

Proof. By induction on the definition of the sharing relation. If $A = A' = A'' \in \{1, B, C \text{ ref}, C \text{ array}\}$ then $|A| = |A'| = |A''|$ and $\Phi_H(v : A) = 0 = 0 + 0 = \Phi_H(v : A') + \Phi_H(v : A'')$.

If $A = \mu X^P.\{c_i : (A_1, \ldots, A_{k_i})\}_i$ then $A' = \mu X^Q.\{c_i : (A'_1, \ldots, A'_{k_i})\}_i$ and $A'' = \mu X^Q.\{c_i : (A''_1, \ldots, A''_{k_i})\}_i$ for some $P, Q, R \in \mathbb{Q}_0^{+\,n}$ with $A_j \curlyvee (A'_j, A''_j)$ and $P[j] = Q[j] + R[j]$ for all $1 \leq j \leq k_i$. By induction we have $|A_j| = |A'_j| = |A''_j|$ for all $1 \leq j \leq k$ and thus $|A| = |A'| = |A''|$. Let v and H be fixed such that $H \vDash v : \mu X^P.\{c_i : (A_1, \ldots, A_{k_i})\}_i$. Then

$$
\begin{aligned}
&\Phi_H(v : \mu X^P.\{c_i : (A_1, \ldots, A_{k_i})\}_i) \\
&= P[i] + \sum_{j=1}^{k_i} \Phi_H(v_j : [A/X]A_j) \\
&= Q[i] + R[i] + \sum_{j=1}^{k_i} \Phi_H(v_j : [A/X]A_j)
\end{aligned}
$$

$$= Q[i] + R[i] + \sum_{j=1}^{k_i} \Phi_H(v_j : [A/X]A'_j) + \Phi_H(v_j : [A/X]A''_j)$$
$$= \Phi_H(v : \mu X^Q.\{c_i : (A'_1, \ldots, A'_{k_i})\}_i) + \Phi_H(v : \mu X^R.\{c_i : (A''_1, \ldots, A''_{k_i})\}_i)$$

The Reference Collection We note that the sharing relation allows for what may appear to be arbitrary duplication of potential for the underlying data in reference types. However, when instances of such types are shared, this data is *aliased*, meaning that multiple references exist to the same underlying data. When one reference points to the same data as another reference, we say that one reference *aliases* the other.

To ensure that aliased references are not double-counted when determining the potential that is contributed by references, we propose a new mapping Δ: $Loc \to A$, where A is the set of linear resource-annotated data types. 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. We represent this relationship with the judgement $H, \Delta \vDash V : \Gamma$ and the operator $\otimes$ defined by the rules in Figure 3, using $\emptyset$ to represent the empty reference collection. Note that $\otimes$ is left-associative and that $H, \Delta \vDash V : \Gamma$ implies $H \vDash V : \Gamma$. 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.

Lemma 2. *Suppose $H, \Delta \vDash \ell : A$ and $A \curlyvee (A', A'')$. Then $H, \Delta \vDash \ell : A'$ and $H, \Delta \vDash \ell : A''$.*

Proof. By induction on the structure of A. If $A = A' = A'' \in \{1, B, C \text{ ref}, C \text{ array}\}$ for some $C \in \mathscr{A}_{\text{lin}}$, then the claim holds trivially.

Otherwise, $A = \mu X^P.\{\cdots \mid c_i{:}(A_1, \ldots, A_{k_i}) \mid \cdots\}$ and thus $A' = \mu X^Q.\{\cdots \mid c_i{:}(A'_1, \ldots, A'_{k_i}) \mid \cdots\}$ and $A'' = \mu X^R.\{\cdots \mid c_i{:}(A''_1, \ldots, A''_{k_i}) \mid \cdots\}$ for $P, Q, R \in \mathbb{Q}_0^{+n}$ with $A_j \curlyvee (A'_j, A''_j)$ and $P[j] = Q[j] + R[j]$ for all $1 \le j \le k_i$. By the premise we get that $H(\ell) = (\mathsf{constr}_{c_i}, v_1, \ldots, v_{k_i})$, $H' = H \setminus \ell$, $H', \Delta_j \vDash v_j : A_j$ for all $1 \le j \le k_i$, and $\Delta = \bigotimes_{1 \le j \le k_i} \Delta_j$. We then apply the induction hypothesis and see that $H', \Delta_j \vDash v_j : A'_j$ and $H', \Delta_j \vDash v_j : A''_j$ for all $1 \le j \le k_i$. By applying Δ:Cons we get that $H, \Delta \vDash \ell : \mu X^Q.\{\cdots \mid c_i{:}(A'_1, \ldots, A'_{k_i}) \mid \cdots\}$ and $H, \Delta \vDash \ell : \mu X^R.\{\cdots \mid c_i{:}(A''_1, \ldots, A''_{k_i}) \mid \cdots\}$ and the claim holds.

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

$$\frac{\begin{array}{c}\forall \ell \in \text{dom}(\Delta_1).\Delta(\ell) = \Delta_1(\ell) \\ \forall \ell \in \text{dom}(\Delta_1) \cap \text{dom}(\Delta_2).\Delta_1(\ell) = \Delta_2(\ell) \qquad \forall \ell \in (\text{dom}(\Delta_2) \setminus \text{dom}(\Delta_1)).\Delta(\ell) = \Delta_2(\ell)\end{array}}{\Delta_1 \otimes \Delta_2 = \Delta}\ (\otimes : \text{C})$$

$$\frac{}{H, \Delta \vDash \texttt{tt} : B}\ (\Delta : \text{True}) \qquad \frac{}{H, \Delta \vDash \texttt{ff} : B}\ (\Delta : \text{False}) \qquad \frac{}{H, \Delta \vDash \text{Null} : 1}\ (\Delta : \mathit{Unit})$$

$$\frac{\begin{array}{c}v \in \mathit{Loc} \qquad H(v) = (\mathsf{constr}_{c_i}, v_1, \ldots, v_k) \qquad H' = H \setminus v \\ T = \mu X^Q.\{\cdots \mid c_i{:}(T_1, \ldots, T_k) \mid \cdots\} \\ H', \Delta_1 \vDash v_1 : [T/X]T_1 \quad \ldots \quad H', \Delta_k \vDash v_k : [T/X]T_k \qquad \Delta = \bigotimes_{1 \le j \le k} \Delta_i\end{array}}{H, \Delta \vDash v : \mu X^Q.\{\cdots \mid c_i{:}(T_1, \ldots, T_k) \mid \cdots\}}\ (\Delta : \text{Constr})$$

$$\frac{n \in \mathbb{N}}{H, \Delta \vDash n : N}\ (\Delta : \text{Nat}) \qquad \frac{H(\ell) = \ell' \qquad H, \Delta' \vDash \ell' : A \qquad \Delta = \Delta' \otimes \{\ell' \mapsto A\}}{H, \Delta \vDash \ell : A \text{ ref}}\ (\Delta : \text{Ref})$$

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

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

Fig. 3. Rules defining the $\otimes$ operator and the well-formed reference collection.

Type Rules As in the case of simple types, a *typing context* is a finite mapping $\Gamma\colon \text{VID} \rightharpoonup \mathscr{A}_{\text{lin}}$. It follows that the potential of a typing context Γ under environment V and heap H is

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

The linear resource-annotated first-order types are defined by the following grammar.

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

In this notation, $q, q' \in \mathbb{Q}_0^+$ and $A \in \mathscr{A}_{\text{lin}}$, 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 $\mathscr{F}_{\text{lin}}$ denote the set of linear resource-annotated first-order types.

A *resource-annotated signature* is a finite mapping $\Sigma\colon \text{FID} \rightharpoonup (\mathscr{P}(\mathscr{F}_{\text{lin}}) \setminus \emptyset)$. As a result, every function can have different resource annotations depending on the context.

Under environment V, heap H, and reference collection Δ such that $H, \Delta \vDash V : \Gamma$ holds, a *resource-annotated typing judgement* has the form

$$\Sigma; \Gamma \vdash^{q}_{q'} e : A$$

which means that, with RAML expression e, resource-annotated signature Σ, resource-annotated context Γ, resource-annotated data type A, and $q, q' \in \mathbb{Q}_0^+$, if there are more than $q + \Phi_{V,H}(\Gamma) + \Phi_H^D(\Delta)$ resource units available, then e may be evaluated, and there are more than $q' + \Phi_{H'}(v : A) + \Phi_{H'}^D(\Delta)$ resource units left if e evaluates to a value v.

As with simple types, we define a well-typed program to consist of a resource-annotated signature Σ and a family $F = (f, \boldsymbol{x}, e_f)_{f \in \mathrm{dom}(\Sigma)}$ of function identifiers $f \in \mathrm{FID}$ with variable identifiers $\boldsymbol{x} = x_1, \ldots, x_n \in \mathrm{VID}$ and expression e_f such that $\Sigma; x_1{:}T_1, \ldots, x_n{:}T_n \vdash^q_{q'} e_f : T$ for each $(T_1, \ldots, T_n) \xrightarrow{q/q'} T \in \Sigma(f)$.

Figure 4 contains the rules that define the resource-annotated typing judgement, including both syntax-directed and structural rules. Most rules are discussed in prior work, and the rules that handle mutable structures and algebraic datatypes are discussed below.

L:Cons assigns potential to a newly constructed instance of a datatype. We ensure that each of the arguments being supplied to the constructor are either the type that it is expecting, or are of the same type as the constructor itself in the places where it expects a recursive type. To do the work, the potential of the context has to pay for both the potential $P[i]$ of the resulting datatype as well as the resource cost M^{cons} of constructing the datatype, where i is determined by the particular constructor being used.

L:Mat shows how to handle pattern matching of constructors. Here, if the constructor is in fact the one we expect, we substitute the full recursive type of our constructor for the type of its elements where necessary. Furthermore, when evaluating the case where the constructors do match, the system provides the $P[i]$ resource units that have been saved up, less the resource cost M_1^{matT} required to initiate the evaluation of $e1$. When $e1$ has evaluated, we must have enough resource units left to satisfy both the constraints on the top-level match expression, as well as to pay the resource cost M_2^{matT} to return from the match expression. In the case where the constructors do not match, we have the same number of resource units as the top-level evaluation, minus the resource cost of initiating the evaluation of $e2$, and upon evaluation, we must have the required number of resource units left in addition to sufficiently many to pay for the resource cost of returning from the evaluation of $e2$.

While L:Ref seems straightforward, it in fact provides strong guarantees about the preservation of potential, in addition to requiring constant resource cost to create the reference. Since the type of the context and of the the data contained by the resulting reference must be the same, the potential within the reference must be exactly the same as the context provides. As a result, we do not lose any potential when data is placed in a mutable heap cell.

L:Dref also requires that the resource cost M^{dref} is available to perform the operation. In addition, it demands that the type of the data returned from the reference has the same underlying type as specified in the reference's type, but where sharing has no effect. Since sharing will always have an effect on the potential annotations unless all annotations are zero, this requires that the output type of a dereference operation carries no potential, as required for soundness.

Furthermore, the L:Assign case provides similar guarantees to the L:Ref case. Beyond the constant cost of performing the operation, this rule requires again that the potential put into the cell is exactly the same as that which the reference already contains. This enforces our requirement that the potential within references remains invariant.

As a natural extension, the L:Swap case combines the operational behavior of the dereference and assign operations. However, since the potential of the data we are putting in is exactly the same as the potential we get out of the cell, we are able to remove data with potential without invalidating the soundness of our system.

L:Create, L:Get, L:Set, and L:Aswap mimic the corresponding rules for references, but we allow the resource cost for creating an array to be dependent on the length of the array.

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.

Theorem 4.01 *Let* $H, \Delta \vDash V : \Gamma$ *and* $\Sigma; \Gamma \vdash^{q}_{q'} e : T$ *hold. If* $F, V, H_M \vdash e \Downarrow (v, H') \mid (p, p')$ *then there exists some* Δ' *such that the following hold:*

- $p \leq \Phi_{V,H}(\Gamma) + \Phi^D_H(\Delta) + q$
- $p - p' \leq \Phi_{V,H}(\Gamma) + \Phi^D_H(\Delta) + q - (\Phi_{H'}(v{:}T) + \Phi^D_{H'}(\Delta') + q')$
- $H', \Delta \vDash V : \Gamma$ *and* $H', \Delta_{ret} \vDash v : T$
- $\Delta' = \Delta \otimes \Delta_{ret}$

Theorem 4.01 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 4.01 in Appendix D. Moreover, we note that the proof is similar to the soundness proof for resource analysis for the language without mutable heap cells [21].

The soundness proof uses Lemma 3 to show the soundness of the rule L:Let, stating that the potential of a context is invariant during the evaluation.

Lemma 3. *Let* $H \vDash V : \Gamma$, $H, \Delta \vDash V : \Gamma$, $\Sigma; \Gamma \vdash^{q}_{q'} e : T$, *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 3.01. Thus, the lemma follows directly by this fact and the definition of the potential Φ.

Many of the cases of the soundness proof follow the same technique, so we show an abbreviated version of the L:Let case as an example. This is particularly demonstrative, as it directly relies on the construction of the reference collection in order to be sound.

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

$$\frac{}{\Sigma; x{:}T \vdash^{\frac{q+M^{\mathrm{var}}}{q}} x : T}\ (\text{L:Var}) \qquad \frac{(T_1,\ldots,T_n) \xrightarrow{q/q'} T \in \Sigma(f)}{\Sigma; x_1{:}T_1,\ldots,x_n{:}T_n \vdash^{\frac{q+M_1^{\mathrm{fun}_n}}{q'-M_2^{\mathrm{fun}_n}}} f(x_1,\ldots,x_n) : T}\ (\text{L:Fun})$$

$$\frac{}{\Sigma;\emptyset \vdash^{\frac{q+M^{\mathrm{nat}}}{q}} n : N}\ (\text{L:Nat}) \qquad \frac{\Sigma;\Gamma_1 \vdash^{\frac{q-M_1^{\mathrm{let}}}{p}} e_1 : T' \qquad \Sigma;\Gamma_2, x{:}T' \vdash^{\frac{p-M_2^{\mathrm{let}}}{q'+M_3^{\mathrm{let}}}} e_2 : T}{\Sigma;\Gamma_1,\Gamma_2 \vdash^{\frac{q}{q'}} \mathrm{let}\ x = e_1\ \mathrm{in}\ e_2 : T}\ \text{L:Let}$$

$$\frac{T = \mu X^P.\{\cdots \mid c_i{:}(B_1,\ldots,B_k) \mid \cdots\} \qquad \begin{array}{c} c \in \mathrm{CID} \\ \forall 1 \le j \le k.\, T_j = B_j \vee (T_j = T \wedge B_j = X) \end{array}}{\Sigma; x_1{:}T_1,\ldots,x_k{:}T_k \vdash^{\frac{q+P[i]+M^{\mathrm{cons}}}{q}} c_i\ (x_1,\ldots,x_k) : T}\ (\text{L:Constr})$$

$$\frac{\begin{array}{c} c \in \mathrm{CID} \qquad T' = \mu X^P.\{\cdots \mid c_i{:}(B_1,\ldots,B_k) \mid \cdots\} \\ \Sigma;\Gamma, x_1{:}[T'/X]B_1,\ldots,x_k{:}[T'/X]B_k \vdash^{\frac{q+P[i]-M_1^{\mathrm{matT}}}{q'+M_2^{\mathrm{MatT}}}} e_1 : T \qquad \Sigma;\Gamma, x{:}T' \vdash^{\frac{q-M_1^{\mathrm{matF}}}{q'+M_2^{\mathrm{matF}}}} e_2 : T \end{array}}{\Sigma;\Gamma, x : T' \vdash^{\frac{q}{q'}} \mathrm{match}\ x\ \mathrm{with}\ c_i\ \langle x_1,\ldots,x_k\rangle \Rightarrow e_1 \mid e_2 : T}\ (\text{L:Mat})$$

$$\frac{\Sigma;\Gamma \vdash^{\frac{q}{q'}} e : T}{\Sigma;\Gamma, x{:}T' \vdash^{\frac{q}{q'}} e : T}\ (\text{L:Weaken}) \qquad \frac{\Sigma;\Gamma, x_1{:}T_1, x_2{:}T_2 \vdash^{\frac{q}{q'}} e : T \qquad T' \curlyvee (T_1,T_2)}{\Sigma;\Gamma, x{:}T' \vdash^{\frac{q}{q'}} \mathrm{share}\ x\ \mathrm{as}\ (x_1,x_2)\ \mathrm{in}\ e : T}\ (\text{L:Share})$$

$$\frac{\Sigma;\Gamma \vdash^{\frac{p}{p'}} e : T \qquad q \ge p \qquad q - p \ge q' - p'}{\Sigma;\Gamma \vdash^{\frac{q}{q'}} e : T}\ (\text{L:Relax}) \qquad \frac{}{\Sigma; x{:}T \vdash^{\frac{q+M^{\mathrm{ref}}}{q}} \mathrm{ref}\ x : T\ \mathrm{ref}}\ (\text{L:Ref})$$

$$\frac{|T'| = |T| \qquad T \curlyvee (T,T)}{\Sigma; x{:}T'\ \mathrm{ref} \vdash^{\frac{q+M^{\mathrm{dref}}}{q}} !x : T}\ (\text{L:DRef}) \qquad \frac{}{\Sigma;\Gamma, x_1{:}T\ \mathrm{ref}, x_2{:}T \vdash^{\frac{q+M^{\mathrm{assign}}}{q}} x_1 := x_2 : 1}\ (\text{L:Assign})$$

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

$$\frac{}{\Sigma; x_1{:}N, x_2{:}T \vdash^{\frac{q+M_n^{\mathrm{create}}}{q}} \mathrm{create}(x_1,x_2) : T\ \mathrm{array}}\ (\text{L:Create})$$

$$\frac{|T'| = |T| \qquad T \curlyvee (T,T)}{\Sigma; x_1{:}T'\ \mathrm{array}, x_2{:}N \vdash^{\frac{q+M^{\mathrm{get}}}{q}} \mathrm{get}(x_1,x_2) : T}\ (\text{L:Get})$$

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

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

Fig. 4. Rules defining the typing judgement for linear resource-annotated types.

(L:Let) If the type derivation ends with an application of L:Let then e is a let expression of the form let $x = e_1$ in e_2 that has eventually been evaluated with the rule E:Let. Then it follows that $F, V, H_m \vdash e_1 \Downarrow (v_1, H_1) \mid (r, r')$ and $F, V', H_{1M} \vdash e_2 \Downarrow (v_2, H_2) \mid (t, t')$ for $V' = V[x \mapsto v_1]$ and r, r', t, t' with

$$(p, p') = M_1^{\text{let}} \cdot (r, r') \cdot M_2^{\text{let}} \cdot (t, t') \cdot M_3^{\text{let}} \tag{1}$$

The derivation of the type judgement for e ends with an application of L:Let. Hence $\Gamma = \Gamma_1, \Gamma_2,\ \Sigma; \Gamma_1 \left|^{s_1}_{s_1'}\right. e_1 : T',\ \Sigma; \Gamma_2, x{:}T' \left|^{s_2}_{s_2'}\right. e_2 : T$, and

$$q = s_1 + M_1^{\text{let}} \tag{2}$$

$$s_1' = s_2 + M_2^{\text{let}} \tag{3}$$

$$q' = s_2' - M_3^{\text{let}} \tag{4}$$

It follows from the definition of Φ that

$$\Phi_{V,H}(\Gamma) = \Phi_{V,H}(\Gamma_1) + \Phi_{V,H}(\Gamma_2) \tag{5}$$

Since $H, \Delta \vDash V : \Gamma$ we also have $H, \Delta \vDash V : \Gamma_1$ and can thus apply the induction hypothesis for the evaluation judgement for e_1 to derive

$$r \leq \Phi_{V,H}(\Gamma_1) + \Phi_H^D(\Delta) + s_1 \tag{6}$$

$$r - r' \leq \Phi_{V,H}(\Gamma_1) + \Phi_H^D(\Delta) + s_1 - (\Phi_{H_1}(v_1{:}T') + \Phi_{H_1}^D(\Delta') + s_1') \tag{7}$$

and $H_1, \Delta \vDash V : \Gamma_1$, $H_1, \Delta_{\text{ret}} \vDash v : T'$, and $\Delta' = \Delta_1 \otimes \Delta_{\text{ret}}$.
We now case for each location reachable from the variables in Γ_2 on whether or not it is reachable from the variables in Γ_1.

- If ℓ is reachable from Γ_1 and has type T, we get from $H_1, \Delta \vDash V : \Gamma_1$ that $H_1, \Delta \vDash \ell : T$.
- Otherwise, if ℓ is not reachable from Γ_1 and has type T, its underlying value must not have been changed by the evaluation of e_1, and therefore $H_1, \Delta \vDash \ell : T$ holds.

Since $H_1, \Delta \vDash \ell : T$ holds for all locations ℓ reachable from Γ_2, it follows that $H_1, \Delta \vDash V : \Gamma_2$.
From the above and $H_1, \Delta_{\text{ret}} \vDash v : T'$, we get that $H_1, \Delta' \vDash V' : \Gamma_2, x{:}T'$ and thus again by induction

$$t \leq \Phi_{V,H}(\Gamma_2, x{:}T') + \Phi_{H_1}^D(\Delta') + s_2 \tag{8}$$

$$t - t' \leq \Phi_{V,H}(\Gamma_2, x{:}T') + \Phi_{H_1}^D(\Delta') + s_2 - (\Phi_{H_2}(v_2{:}T) + \Phi_{H_2}^D(\Delta'') + s_2') \tag{9}$$

Now let

$$\begin{aligned}(u, u') =& M_1^{\text{let}} \cdot (\Phi_{V,H}(\Gamma_1) + \Phi_H^D(\Delta) + s_1,\\ & \Phi_{H_1}(v_1{:}T') + \Phi_{H_1}^D(\Delta') + s_1') \cdot\\ & M_2^{\text{let}} \cdot (\Phi_{V,H}(\Gamma_2, x{:}T') + \Phi_{H_1}^D(\Delta') + s_2,\\ & \Phi_{H_2}(v_2{:}T) + \Phi_{H_2}^D(\Delta'') + s_2') \cdot M_3^{\text{let}}\end{aligned}$$

Then it follows that

$$(u, u) \overset{(3,4)}{=} M_1^{\text{let}} \cdot (v + \Phi_{V,H}(\Gamma_1) + \Phi_H^D(\Delta) + s_1, v')$$

for $v, v' \in \mathbb{Q}_0^+$ with

$$\begin{aligned} v \;\leq\;& \Phi_{V',H_1}(\Gamma_2, x{:}T') + \Phi_{H_1}^D(\Delta') + s_2\\ & - (\Phi_{H_1}(v_1{:}T') + \Phi_{H_1}^D(\Delta') + s_1' - M_2^{\text{let}})\\ =\;& \Phi_{V',H_1}(\Gamma_2) + \Phi_{H_1}^D(\Delta') + s_2\\ & - (\Phi_{H_1}^D(\Delta') + s_1' - M_2^{\text{let}})\\ \overset{(3)}{=}\;& \Phi_{V',H_1}(\Gamma_2) + \Phi_{H_1}^D(\Delta') - \Phi_{H_1}^D(\Delta')\\ =\;& \Phi_{V',H_1}(\Gamma_2)\end{aligned}$$

and thus

$$\begin{aligned} u \leq\;& \max(0,\\ & \Phi_{V,H}(\Gamma_1) + \Phi_H^D(\Delta) + \Phi_{V',H_1}(\Gamma_2) + s_1 + M_1^{\text{let}})\\ \overset{(Lem.3)}{\leq}\;& \max(0,\\ & \Phi_{V,H}(\Gamma_1) + \Phi_H^D(\Delta) + \Phi_{V,H_1}(\Gamma_2) + s_1 + M_1^{\text{let}})\\ \overset{(2)}{\leq}\;& \Phi_{V,H}(\Gamma) + \Phi_H^D(\Delta) + q\end{aligned}$$

Finally, it follows with Proposition 3.01 applied to (6), (8), and (1) that $u \geq p$.
For the second part of the statement we apply Proposition 3.01 to (1) and, by way of (3), Lemma 3, (2), and (4), we can derive the following.

$$\begin{aligned} p - p' &= r - r' + t - t' + M_1^{\text{let}} + M_2^{\text{let}} + M_3^{\text{let}}\\ &\leq \Phi_{V,H}(\Gamma) + \Phi_H^D(\Delta) + q - (\Phi_{H_2}(v_2{:}T) + \Phi_{H_2}^D(\Delta'') + q')\end{aligned}$$

This shows that the amount of resource units consumed over the evaluation of the let expression is at most the sum of the potential of the input context, reference collection, and q, less the potential of the return value, resulting reference collection, and q'. Therefore, our constraints form a valid upper bound and this case is sound.

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.

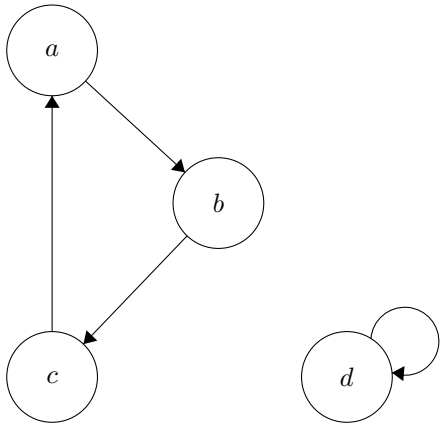
We use the following set of user defined types as our graph representation.

```
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 define the following 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.

```
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)
```

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.

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*.

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

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*).

$$make\_graph\ (() : 1) : L^1(int\ node) \qquad \text{[With no refresh]}$$
$$make\_graph\ (() : 1) : L^3(int\ node) \qquad \text{[With one refresh]}$$

6 Type Inference

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 [22, 25].

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 4 and collect the local constraints that need to hold for the annotations. We then minimize the initial potential using the LP solver.

The process is best explained by example. Consider the following program.

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

To infer an annotated type we start with simple type derivation. We then fix an annotation

$$f : (L^a(1), L^b(B)\ \text{ref}) \xrightarrow{q/q'} L^c(L^d(B)\ \text{ref})$$

for the function f and annotate the root of the derivation with

$$\Sigma; l{:}L^a(1), r{:}L^b(B)\ \text{ref} \left|\frac{q}{q'}\right. e_f : L^c(L^d(B)\ \text{ref})\ .$$

We furthermore annotated the rest of the derivation with fresh resource variables, as shown below.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Sigma; \emptyset \left|\frac{q_9}{q_{10}}\right. \text{nil} : L^c(L^b(B)\ \text{ref})}}{\Sigma; l{:}L^a(1), r{:}L^b(B)\ \text{ref} \left|\frac{q_{11}}{q_{12}}\right. \text{nil} : L^c(L^d(B)\ \text{ref})}}{\Sigma; r2{:}L^b(B)\ \text{ref}, t{:}L^c(L^d(B)\ \text{ref}) \left|\frac{q_3}{q_4}\right. \text{cons}(r2, t) : L^c(L^d(B)\ \text{ref})}}{\Sigma(f) = (L^a(1), L^b(B)\ \text{ref}) \xrightarrow{q/q'} L^c(L^d(B)\ \text{ref})}}{\Sigma; xs{:}L^a(1), r1{:}L^b(B)\ \text{ref} \left|\frac{q_1}{q_2}\right. f(xs, r1) : L^c(L^d(B)\ \text{ref})}}{\Sigma; r1{:}L^b(B)\ \text{ref}, r2{:}L^b(B)\ \text{ref}, xs{:}L^a(1) \left|\frac{q_5}{q_6}\right. e'_T : L^c(L^d(B)\ \text{ref})}}{\Sigma; r{:}L^b(B)\ \text{ref}, x{:}1, xs{:}L^a(1) \left|\frac{q_7}{q_8}\right. e_T : L^c(L^d(B)\ \text{ref})}}{\Sigma; l{:}L^a(1), r{:}L^b(B)\ \text{ref} \left|\frac{q}{q'}\right. e_f : L^c(L^d(B)\ \text{ref})}$$

In the typing derivation we use the following abbreviations.

$$\begin{aligned} e'_T &= \text{let } t = f(xs, r1) \text{ in } \text{cons}(r2, t) \\ e_T &= \text{share } r \text{ as } (r1, r2) \text{ in } e'_T \\ e_f &= \text{match } l \text{ with } \text{cons}(x, xs) \to e_T \mid \text{nil} \end{aligned}$$

By inspecting the earlier type rules we produce the following local constraints.

$$
\begin{array}{lll}
q_1 = q + M_1^{\mathrm{fun}_n} & q_3 = p - M_2^{\mathrm{let}} & q_{12} = q_{10} \\
q_2 = q' - M_2^{\mathrm{fun}_n} & q_4 = q_6 + M_3^{\mathrm{let}} & q_7 = q_{13} + a - M_1^{\mathrm{matT}} \\
q_3 = q_4 + c + M^{\mathrm{cons}} & q_7 = q_5 & q_8 = q_{14} + M_2^{\mathrm{matT}} \\
q_1 = q_5 - M_1^{\mathrm{let}} & q_8 = q_6 & b = d \\
p = q_2 & q_{11} = q_9 & q_{11} = q_{13} - M_1^{\mathrm{matF}} \\
 & & q_{12} = q_{14} + M_2^{\mathrm{matF}}
\end{array}
$$

We then call the LP solver using the objective function $a + b + q$. It is also possible to iterative solve the LP by first minimizing $a + b$ and then q. If we use the metric in which $M^{\mathrm{cons}} = 1$ and $M^{\mathrm{x}} = 0$ otherwise then one possible solution of the constraint system corresponds to the following function annotation.

$$f : (L^2(1), L^9(B)\ \mathrm{ref}) \xrightarrow{0/0} L^1(L^9(B)\ \mathrm{ref})$$

7 Polynomial Bounds and Higher-Order Functions

To make the material more accessible, we present the potential annotations for references in first order setting. However, the proposed technique scales to higher-order programs and polynomial bounds [24].

Assume we would add references of ML type $T \equiv (T_1 \to T_2)\ \mathit{ref}$ 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 $A \xrightarrow{q/q'} A\ \mathit{ref}$. Like previous work [24] we can basically leave the our rules L:Swap, L:Assign, L:DRef, and L:Ref unchanged; and the same is true for the rules for 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 [25]. We then 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 [22]. 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 stored in the same reference but not for data that is stored across different references.

We have already implemented multivariate polynomial annotations for references for most of RAML. The only remaining part of the implementation is that which concerns recursive functions. We are confident that the development will be completed in the next few weeks.

8 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 [27]. Linear AARA has been extended to work with higher-order functions [31], object-oriented programs [32], lazy evaluation [40], imperative integer programs [10], pointer-based data structures [5], polynomial bounds [22]. Many of the features for strict functional programs have been combined in Resource Aware ML [24], an implementation of AARA for OCaml. In contrast to the presented technique, none of the previous 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 [24, 26]. 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 [24] 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 [5] and object-oriented type systems [28, 30]. 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 sized types [39], linear dependent types [33,34], and type annotations [13,14]. Cicek et al. [12] 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 [7, 15, 16, 18]. None of these analysis systems can deal with side-effects and most have only basic support for automation.

Another research directing is to apply techniques from term rewriting to complexity analysis [6, 9, 36]; sometimes in combination with amortized analysis [29]. However, existing techniques seem to be restricted to purely functional programs and time complexity.

Abstract interpretation based approaches to resource analysis [1, 3, 8, 11, 17, 19, 38, 41] 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.

9 Conclusion

We have extended automatic amortized resource analysis to be able 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 in the type system. In order to still be able to track resource usage that depends on the sizes of data stored in mutable cells, we require that the potential in these cells in invariant during the execution. Furthermore, we introduced 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 introduced reference collections, which allow us to track relevant heap cells while not falling victim to aliasing.

Our analysis preserves all 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 and 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`.

One of the open problems in resource analysis is how to combine automatic techniques with user interaction to derive bounds for functions that cannot be analyzed automatically; furthermore, it is as of yet undecided how to make such bounds available for use in the automation. The introduction of new language features like `swap` is one way of guiding an automatic resource analysis without manually deriving bounds. We plan to further investigate as to what extent language design can aid resource analysis and how user interaction can broaden the scope of AARA.

References

1. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In *16th Euro. Symp. on Prog. (ESOP'07)*, pages 157–172, 2007.
2. E. Albert, R. Bubel, S. Genaim, R. Hähnle, and G. 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. E. Albert, J. C. Fernández, and G. 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. C. Alias, A. Darte, P. Feautrier, and L. Gonnord. Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs. In *17th Int. Static Analysis Symposium (SAS'10)*, pages 117–133, 2010.
5. R. Atkey. Amortised Resource Analysis with Separation Logic. In *19th Euro. Symp. on Prog. (ESOP'10)*, pages 85–103, 2010.
6. M. Avanzini, U. D. Lago, and G. Moser. Analysing the Complexity of Functional Programs: Higher-Order Meets First-Order. In *29th Int. Conf. on Functional Programming (ICFP'15)*, 2012.
7. R. Benzinger. Automated Higher-Order Complexity Analysis. *Theor. Comput. Sci.*, 318(1-2):79–103, 2004.
8. R. Blanc, T. A. Henzinger, T. Hottelier, and L. Kovács. ABC: Algebraic Bound Computation for Loops. In *Logic for Prog., AI., and Reasoning - 16th Int. Conf. (LPAR'10)*, pages 103–118, 2010.
9. M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. 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.
10. Q. Carbonneaux, J. Hoffmann, and Z. Shao. Compositional Certified Resource Bounds. In *36th Conference on Programming Language Design and Implementation (PLDI'15)*, 2015. Artifact submitted and approved.
11. P. Cerný, T. A. Henzinger, L. Kovács, A. Radhakrishna, and J. Zwirchmayr. Segment Abstraction for Worst-Case Execution Time Analysis. In *24th European Symposium on Programming (ESOP'15)*, pages 105–131, 2015.
12. E. Çiçek, D. Garg, and U. A. Acar. Refinement Types for Incremental Computational Complexity. In *24th European Symposium on Programming (ESOP'15)*, pages 406–431, 2015.
13. K. Crary and S. Weirich. Resource Bound Certification. In *27th ACM Symp. on Principles of Prog. Langs. (POPL'00)*, pages 184–198, 2000.
14. N. A. 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.
15. N. Danner, D. R. Licata, and R. Ramyaa. Denotational Cost Semantics for Functional Languages with Inductive Types. In *29th Int. Conf. on Functional Programming (ICFP'15)*, 2012.
16. N. Danner, J. Paykin, and J. S. Royer. A Static Cost Analysis for a Higher-Order Language. In *7th Workshop on Prog. Languages Meets Prog. Verification (PLPV'13)*, pages 25–34, 2013.
17. A. Flores-Montoya and R. Hähnle. Resource Analysis of Complex Programs with Cost Equations. In *Programming Languages and Systems - 12th Asian Symposiu (APLAS'14)*, pages 275–295, 2014.
18. B. Grobauer. Cost Recurrences for DML Programs. In *6th Int. Conf. on Funct. Prog. (ICFP'01)*, pages 253–264, 2001.
19. S. Gulwani, K. K. Mehra, and T. 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.
20. S. Gulwani and F. Zuleger. The Reachability-Bound Problem. In *Conf. on Prog. Lang. Design and Impl. (PLDI'10)*, pages 292–304, 2010.
21. J. Hoffmann. *Types with Potential: Polynomial Resource Bounds via Automatic Amortized Analysis*. PhD thesis, Ludwig-Maximilians-Universität München, 2011.
22. J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate Amortized Resource Analysis. In *38th Symposium on Principles of Programming Languages (POPL'11)*, 2011.

23. J. Hoffmann, K. Aehlig, and M. Hofmann. Resource Aware ML. In *24rd International Conference on Computer Aided Verification (CAV'12)*, 2012.

24. J. Hoffmann, A. Das, and S.-C. Weng. Towards Automatic Resource Bound Analysis for OCaml. In *44th Symposium on Principles of Programming Languages (POPL'17)*, 2017.

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

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

27. M. Hofmann and S. 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.

28. M. Hofmann and S. Jost. Type-Based Amortised Heap-Space Analysis. In *15th Euro. Symp. on Prog. (ESOP'06)*, pages 22–37, 2006.

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

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

31. S. Jost, K. Hammond, H.-W. Loidl, and M. 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.

32. S. Jost, H.-W. Loidl, K. Hammond, N. Scaife, and M. Hofmann. Carbon Credits for Resource-Bounded Computations using Amortised Analysis. In *16th Symp. on Form. Meth. (FM'09)*, pages 354–369, 2009.

33. U. D. Lago and M. Gaboardi. Linear Dependent Types and Relative Completeness. In *26th IEEE Symp. on Logic in Computer Science (LICS'11)*, pages 133–142, 2011.

34. U. D. Lago and B. Petit. The Geometry of Types. In *40th ACM Symp. on Principles Prog. Langs. (POPL'13)*, pages 167–178, 2013.

35. G. Morrisett, A. Ahmed, and M. Fluet. *L3: A Linear Language with Locations*, pages 293–307. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

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

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

38. M. Sinn, F. Zuleger, and H. 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.

39. P. Vasconcelos. *Space Cost Analysis Using Sized Types*. PhD thesis, School of Computer Science, University of St Andrews, 2008.

40. P. B. Vasconcelos, S. Jost, M. Florido, and K. Hammond. Type-Based Allocation Analysis for Co-recursion in Lazy Functional Languages. In *24th European Symposium on Programming (ESOP'15)*, pages 787–811, 2015.

41. F. Zuleger, M. Sinn, S. Gulwani, and H. Veith. Bound Analysis of Imperative Programs with the Size-change Abstraction. In *18th Int. Static Analysis Symp. (SAS'11)*, pages 280–297, 2011.

A Rules Defining the Simple Typing Judgement

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

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

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

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

$$\frac{c \in \text{CID} \qquad T = \mu X.\{\cdots \mid c_i{:}(B_1, \ldots, B_k) \mid \cdots\} \qquad \forall 1 \le j \le k.\, T_j = B_j \vee (T_j = T \wedge B_j = X)}{\Sigma; x_1{:}T_1, \ldots, x_k{:}T_k \vdash c_i\ (x_1, \ldots, x_k) : T}\ \text{(T:Constr)}$$

$$\frac{\begin{array}{c} c \in \text{CID} \qquad T' = \mu X.\{\cdots \mid c_i{:}(B_1, \ldots, B_k) \mid \cdots\} \\ \Sigma; \Gamma, x_1{:}[T'/X]B_1, \ldots, x_k{:}[T'/X]B_k \vdash e_1 : T \qquad \Sigma; \Gamma, x{:}T' \vdash e_2 : T \end{array}}{\Sigma; \Gamma, x : T' \vdash \text{match } x \text{ with } c_i\ \langle x_1, \ldots, x_k \rangle \Rightarrow e_1 \mid e_2 : T}\ \text{(T:Mat)}$$

$$\frac{\Sigma; \Gamma \vdash e : T}{\Sigma; \Gamma, x{:}T' \vdash e : T}\ \text{(T:Weaken)}$$

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

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

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

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

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

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

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

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

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

B Proof of Theorem 3.01

Proof. We go by induction on the derivation of

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

Note that the only rules for which there exists some $\ell \in \text{dom}(H)$ for which $H'(\ell) \neq H(\ell)$ are E:Swap, E:Assign, E:Set, and E:Aswap, so we only show these cases; furthermore, each of these cases follow by virtually the same argument, so we only show the (E:Swap) case. In all other cases, $H'(\ell) = H(\ell)$ for all $\ell \in \text{dom}(H)$, so the claim holds trivially.

(E:Swap) Suppose that the derivation of the evaluation judgement ends with an application of the rule (E:Swap). Then by the typing judgement, we know that $\Gamma(x_1) = T$ ref, and thus by the well-formed environment judgement, we know that $H \vDash (\mathsf{loc}, V(x_1)) : T$ ref. Similarly, we know that $\Gamma(x_2) = T$, and therefore, $H \vDash v : T$ for some value v. From the premise we know that $H' = H[V(x_1) \mapsto v]$, and therefore, $H' \vDash H'(V(x_1)) : T$. By the V:Ref rule, it follows that $H' \vDash (\mathsf{loc}, V(x_1)) : T$ ref. Since no other location $\ell \in \mathrm{dom}(H)$ is changed, we conclude that $H' \vDash V : \Gamma$ holds for this case.

C Proof of Theorem 3.02

Proof. The first part, $H' \vDash V : \Gamma$, follows directly from Theorem 3.01.

The second part, $H' \vDash v : T$, is proved by induction on the derivations of the evaluation judgement and the typing judgement, with the former taking priority.

As many cases remain unchanged from Hoffmann [21], we will only show those concerning the new rules, as well as the (T:Let) case.

(T:Constr) If the type derivation ends with the application of the rule (T:Constr) then the derivation of the evaluation judgement ends with an application of (E:Constr). By the well-formed environment judgement, we have that $H \vDash V : (x_1 : T_1, \ldots, x_k : T_k)$. Furthermore, from the type derivation, we know that

$$T = \mu X.\{\cdots \mid c_i : (B_1, \ldots, B_k) \mid \ldots\}$$

and that

$$\forall 1 \leq j \leq k. T_j = B_j \vee (T_j = T \wedge B_j = X)$$

Therefore, we can apply (V:Constr) and get that $H' \vdash v : T$ and the case holds.

(T:Mat) If the type derivation ends with the application of the rule (T:Mat) then the derivation of the evaluation judgement either ends with (E:Mat1) or (E:Mat2). In both cases we have

$$T' = \mu X.\{\cdots \mid c_i : (B_1, \ldots, B_k) \mid \ldots\}$$

- In the former case, we have that $H \vDash V : (\Gamma, x{:}T')$ and therefore, where $V(x) = \ell$, $H \vDash \ell : T'$. From the evaluation judgement, we know that $H(\ell) = (\mathsf{constr}_c, v_1, \ldots, v_k)$, and from the well-formed location judgement, that

  $$\forall 1 \leq j \leq k. H \vDash v_j : [T'/X]B_j$$

  Therefore, it follows that $H \vDash V[x_1 \mapsto v_1, \ldots, x_k \mapsto v_k] : (\Gamma, x_1 : [T'/X]B_1, \ldots, x_k : [T'/X]B_k)$.
  We can now apply the induction hypothesis to the premises of (T:Mat), (E:Mat1), and the previous result, and get that $H' \vDash v : T$ holds as desired.

– In the latter case, we can immediately apply the induction hypothesis to the premises of (T:Mat) and (E:Mat2) and get that $H' \vDash v : T$ holds as desired.

(T:Ref) If the type derivation ends with the application of the rule (T:Ref) then the derivation of the evaluation judgement ends with an application of (E:Ref). We know from the well-formed environment judgement that $H \vDash V : x{:}T$, so $H \vDash V(x) : T$. Since $H' = H, \ell \mapsto V(x)$ with $\ell \notin \mathrm{dom}(H)$, it follows by the (V:Ref) rule that $H' \vDash \ell : T$ ref and the case holds.

(T:DRef) If the type derivation ends with the application of the rule (T:DRef) then the derivation of the evaluation judgement ends with an application of (E:DRef). We know from the well-formed environment judgement that $H \vDash V : x{:}T$ ref, so $H \vDash V(x) : T$ ref. By inversion on the rule (V:Ref) it follows that $H \vDash \ell : T$ with $\ell = H(V(x))$, and this case holds.

(T:Swap) Similar to the case (T:DRef).

(T:Assign) If the type derivation ends with the application of the rule (T:Assign) then the derivation of the evaluation judgement ends with an application of (E:Assign). The case holds directly from the rule (V:Unit).

(T:Create) If the type derivation ends with the application of the rule (T:Create) then the derivation of the evaluation judgement ends with an application of (E:Create). We know from the well-formed environment judgement that $H \vDash V : x_1{:}N, x_2{:}T$, so $H \vDash V(x_2) : T$. Since $H' = H, \ell \mapsto (\sigma, n)$ with $\ell \notin \mathrm{dom}(H)$ and $\sigma(i) = V(x_2)$ for all $0 \leq i < n$, it follows by the (V:Array) rule that $H' \vDash \ell : T$ array and the case holds.

(T:Get) If the type derivation ends with the application of the rule (T:Get) then the derivation of the evaluation judgement ends with an application of (E:Get). We know from the well-formed environment judgement that $H \vDash V : x_1{:}T$ array$, x_2{:}N$, so $H \vDash V(x_1) : T$ array and $H \vDash V(x_2) : N$. Let $V(x_2) = i$. By inversion on the rule (V:Nat), we get that $i \in \mathbb{N}$, so by inversion on the rule (V:Array) it follows that $H \vDash \ell : T$ with $H(V(x_1)) = (\sigma, n)$ and $\ell = \sigma(i)$, and this case holds.

(T:Aswap) Similar to the case (T:Get).

(T:Set) If the type derivation ends with the application of the rule (T:Set) then the derivation of the evaluation judgement ends with an application of (E:Set). The case holds directly from the rule (V:Unit).

(T:Len) If the type derivation ends with the application of the rule (T:Len) then the derivation of the evaluation judgement ends with an application of (E:Len). We know from the well-formed environment judgement that $H \vDash V : x{:}T$ array, so $H \vDash V(x) : T$ array. By inversion on the rule (V:Array), we get that $H(V(x)) = (\sigma, n)$ with $n \in \mathbb{N}$, so by (V:Nat), we get that $H \vDash n : N$, and this case holds.

(T:Let) If the type derivation ends with the application of the rule (T:Let) then the derivation of the evaluation judgement ends with an application of (E:Let). We have $\Sigma; \Gamma_1 \vdash e_1 : T'$ from the premises of (T:Let) and also $H \vDash V : \Gamma_1$ from $H \vDash V : (\Gamma_1, \Gamma_2)$. We can now apply the induction hypothesis to $F, V, H_M \vdash e_1 \Downarrow (v_1, H') \mid (q, q')$ and derive $H' \vDash v_1 : T'$.
We now case on the last rule used in the derivation of $\Sigma; \Gamma_1 \vdash e_1 : T'$.

– If the type derivation ends with the application of the rule (T:Swap), then the derivation of the evaluation judgement ends with an application of (E:Swap). By Lemma 3.01, it follows that $H' \vDash V : \Gamma$. As a result, since $H' \vDash v_1 : T'$ and $\mathrm{img}(V) \subseteq \mathrm{dom}(H)$, we conclude that $H' \vDash V[x \mapsto v_1] : (\Gamma, x : T')$.
– If the type derivation ends with the application of the rule (T:Assign), we follow the argument for the previous case.
– In all other cases, $H'(\ell) = H(\ell)$ for all $\ell \in \mathrm{dom}(H)$, so $H' \vDash V : \Gamma$. The proof now proceeds as above.

In all cases, $H' \vDash V[x \mapsto v_1] : (\Gamma, x : T')$ holds. In addition, we have $\Sigma; \Gamma_2, x : T' \vdash e_2 : T$ from the premises of (T:Let). We now apply the induction hypothesis a second time to $F, V[x \mapsto v_1], H'_M \vdash e_2 \Downarrow (v_2, H'') \mid (p, p')$ and derive $H'' \vDash v_2 : T$, and the case holds.

D Proof of the Soundness Theorem

(L:Weaken) If the derivation of $\Sigma; \Gamma \left|\frac{q}{q'}\right. e : T$ ends with an application of the rule L:Weaken then we have $\Sigma; \Gamma' \left|\frac{q}{q'}\right. e : T$ for a context Γ' with $\Gamma', x{:}T' = \Gamma$. From the premise $H, \Delta \vDash V : \Gamma', x{:}T'$ it follows that $H, \Delta \vDash V : \Gamma'$ by the L:Weaken1 rule. Thus we can apply the induction hypothesis and derive $p \leq \Phi_{V,H}(\Gamma') + \Phi^D_H(\Delta_0) + q$ and

$$p - p' \leq \Phi_{V,H}(\Gamma') + \Phi^D_H(\Delta) + q \\ - (\Phi_{H'}(v : T) + \Phi^D_{H'}(\Delta') + q') \quad (10)$$

with
– $H', \Delta \vDash V : \Gamma'$
– $H', \Delta_{\mathrm{ret}} \vDash v : T$
– $\Delta' = \Delta \otimes \Delta_{\mathrm{ret}}$

By the definition of Φ, we also have

$$\Phi_{V,H}(\Gamma') \leq \Phi_{V,H}(\Gamma) \quad (11)$$

Thus it follows by (10) and (11) that

$$\begin{aligned} p - p' &\leq \Phi_{V,H}(\Gamma') + \Phi^D_H(\Delta) + q \\ &\quad - (\Phi_{H'}(v : T) + \Phi^D_{H'}(\Delta') + q') \\ &\leq \Phi_{V,H}(\Gamma) + \Phi^D_H(\Delta) + q \\ &\quad - (\Phi_{H'}(v : T) + \Phi^D_{H'}(\Delta') + q') \end{aligned}$$

and the claim holds.

(L:Relax) We apply the induction hypothesis to

$$F, V, H_M \vdash e \Downarrow (v, H') \mid (p, p')$$

and to the premise $\Sigma; \Gamma \vdash^{r}_{r'} e : T$ of L:Relax. Then we have $p \leq \Phi_{V,H}(\Gamma) + \Phi^D_H(\Delta) + r$ and $p - p' \leq \Phi_{V,H}(\Gamma) + \Phi^D_H(\Delta) + r - (\Phi_{H'}(v : T) + \Phi^D_{H'}(\Delta') + r')$. From the premise of L:Relax we have $q \geq r$ and $q - r \geq q' - r'$ and thus $q - q' \geq r - r'$. Thus the claim follows.

(L:Share) Assume that e is a variable sharing of the form share x as (x_1, x_2) in e'. The evaluation of e then ends with an application of the rule E:Share. Thus we have

$$F, V, H_M \vdash \text{share } x \text{ as } (x_1, x_2) \text{ in } e \Downarrow (v, H') \mid (p, p')$$

The derivation of the type judgement for e ends with an application of L:Share. Thus we have that $V(x) = \ell$, $\Gamma = \Gamma', x{:}T'$ and

$$\Sigma; \Gamma, x{:}T' \vdash^{q}_{q'} \text{share } x \text{ as } (x_1, x_2) \text{ in } e : T$$

From the premise $H, \Delta \vDash V : \Gamma', x{:}T'$ we have that $H, \Delta \vDash V : \Gamma'$ and $H, \Delta \vDash \ell : T'$.
Let $V' = (V \setminus x) \cup \{x_1 \mapsto \ell, x_2 \mapsto \ell\}$ for $x_1{:}T_1$ and $x_2{:}T_2$ with $T' \curlyvee (T_1, T_2)$. By Lemma 2, we get that $H, \Delta \vDash V(x_1) : T_1$, $H, \Delta \vDash V(x_2) : T_2$, and by the construction of the reference collection, $\Delta = \Delta \otimes \Delta \otimes \Delta$. Therefore, we see that $H, \Delta \vDash V' : \Gamma', x_1{:}T_1, x_2{:}T_2$ and can now apply the induction hypothesis.
We thus derive

$$p \leq \Phi_{V',H}(\Gamma', x_1{:}T_1, x_2{:}T_2) + \Phi^D_H(\Delta) + q \tag{12}$$

and

$$p \leq \Phi_{V',H}(\Gamma', x_1{:}T_1, x_2{:}T_2) + \Phi^D_H(\Delta) + q - (\Phi_{H'}(v{:}T) + \Phi^D_{H'}(\Delta') + q') \tag{13}$$

By Lemma 1 we have that

$$\Phi_{V',H}(x_1{:}T_1) + \Phi_{V',H}(x_2{:}T_2) = \Phi_{V,H}(x{:}T')$$

and hence

$$\Phi_{V',H}(\Gamma', x_1{:}T_1, x_2{:}T_2) = \Phi_{V,H}(\Gamma', x{:}T') \tag{14}$$

by the definition of Φ. The claim follows from (12), (13), and (14).

(L:Var) Assume that e is a variable x that has been evaluated with the rule E:Var. Assume first that $M^{\text{var}} \geq 0$. Then it follows by definition that $p = M^{\text{var}}$ and $p' = 0$. The type judgement $\Sigma; x{:}T \vdash^{q}_{q'} x : T$ has been derived by a single application of the rule L:Var. Thus we have $0 \leq q' = q - M^{\text{var}}$ and $p = M^{\text{var}} \leq q \leq \Phi_{V,H}(x{:}T) + \Phi^D_H(\Delta) + q$. In addition, it follows from E:Var that $v = V(x)$ and thus $p - p' = M^{\text{var}} = q - q' = \Phi_{V,H}(x{:}T) + \Phi^D_H(\Delta) + q -$

$(\Phi_{V,H}(v{:}T)+\Phi^D_H(\Delta)+q')$. Note that Δ does not change under this evaluation since v is included in Γ.

Assume now that $M^{\text{var}} < 0$. Then it follows by definition that $p = 0$ and $p' = -M^{\text{var}}$. Thus $p = 0 \leq \Phi_{V,H}(x{:}T) + \Phi^D_H(\Delta) + q$. We have again that $q - q' = M^{\text{var}} = p - p'$. Therefore the second part of the claim follows as in the previous case.

(L:True) Similar to the case L:Var.

(L:False) Similar to the case L:Var.

(L:Triv) Similar to the case L:Var.

(L:Constr) If the type derivation ends with an application of L:Cons then e has the form $c_i\langle x_1, \ldots, x_k\rangle$ and has been evaluated by E:Cons. It follows by definition that

$$F, V, H_M \vdash c_i\langle x_1, \ldots, x_k\rangle \Downarrow (\ell, H') \mid M^{\text{cons}}$$

$x_1, \ldots, x_k \in \text{dom}(V)$, $H' = H, \ell \mapsto (\mathsf{constr}_{c_i}, V(x_1), \ldots, V(x_k))$, and $\ell \notin \text{dom}(H)$. Thus,

$$p = M^{\text{cons}} \text{ and } p' = 0 \tag{15}$$

or (if $M^{\text{cons}} < 0$)

$$p = 0 \text{ and } p' = -M^{\text{cons}} \tag{16}$$

We have $T = \mu X^S.\{\cdots \mid c_i : (B_1, \ldots, B_k) \mid \ldots\}$,

$$\Sigma; x_1{:}T_1, \ldots, x_k{:}T_k \left|\frac{q}{q'}\right. c_i\ (x_1, \ldots, x_k) : T$$

and

$$\forall 1 \leq j \leq k. T_j = B_j \vee (T_j = T \wedge B_j = X)$$

by a single application of L:Cons; thus

$$0 \leq q' = q - S[i] - M^{\text{cons}} \tag{17}$$

If $p = 0$ then $p \leq \Phi_{V,H}(\Gamma) + q$ holds since $q \geq 0$ always. Otherwise $p = M^{\text{cons}} \leq q \leq \Phi_{V,H}(\Gamma) + q$.

We briefly note that H' only differs from H by a single additional location that adds no new values with an underlying reference type that were not already accessible from the context. Therefore, $\Delta' = \Delta$ and $\Phi^D_H(\Delta) = \Phi^D_{H'}(\Delta)$ in this case.

Let $v = (\mathsf{constr}_{c_i}, V(x_1), \ldots, V(x_k))$. From the definition of Φ it follows that

$$S[i] + \Phi_{V,H}(x_1{:}T_1, \ldots, x_k{:}T_k) = \Phi_{H'}(\ell : T) \tag{18}$$

Therefore,

$$\begin{aligned}
\Phi_{V,H}(\Gamma) + q \quad &= \quad \Phi_{V,H}(x_1{:}T_1, \ldots, x_k{:}T_k) + q \\
&\overset{(17)}{=} \quad \Phi_{V,H}(x_1{:}T_1, \ldots, x_k{:}T_k) \\
&\qquad + q' + s + M^{\text{cons}} \\
&\overset{(18)}{=} \quad q' + M^{\text{cons}} + \Phi_{H'}(\ell : T)
\end{aligned}$$

and thus $\Phi_{V,H}(\Gamma)+\Phi^D_H(\Delta)+q-(q'+\Phi^D_{H'}(\Delta)+\Phi_{H'}(\ell : T)) = M^{\text{cons}} = p-p'$.

(L:Fun) Assume that e is a function application of the form $f(x_1, \ldots, x_n)$. The evaluation of e then ends with an application of the rule E:Fun. Thus we have $V(x_i) = v_i$ and $F, [y_1 \mapsto v_1, \ldots, y_n \mapsto v_n], H_M \vdash e_f \Downarrow (v, H') \mid (r, r')$ for some r, r' with

$$(p, p') = M_1^{\mathrm{fun}_n} \cdot (r, r') \cdot M_2^{\mathrm{fun}_n} \tag{19}$$

The derivation of the type judgement for e ends with an application of L:Fun. Therefore it is true that $\Gamma = x_1{:}T_1, \ldots, x_n{:}T_n$, $(T_1, \ldots, T_n) \xrightarrow{s/s'} T \in \Sigma(f)$, and

$$q = s + M_1^{\mathrm{fun}_n} \text{ and } q' = s' - M_2^{\mathrm{fun}_n} \tag{20}$$

In order to apply the induction hypothesis to the evaluation of the function body we recall from the definition of a well-formed program that $(T_1, \ldots, T_n) \xrightarrow{s/s'} T \in \Sigma(f)$ implies that $\Sigma; y_1{:}T_1, \ldots, y_n{:}T_n \vdash^{s}_{s'} e_f{:}T$. Since $H, \Delta \vDash V : x_1{:}T_1, \ldots, x_n{:}T_n$ and $V(x_i) = v_i$, it follows that $H, \Delta \vDash [y_1 \mapsto v_1, \ldots, y_n \mapsto v_n] : y_1{:}T_1, \ldots, y_n{:}T_n$ since all locations remain accounted for and reachable. We obtain by induction that

$$r \leq \Phi_{[y_1 \mapsto v_1, \ldots, y_n \mapsto v_n], H}(y_1{:}T_1, \ldots, y_n{:}T_n) + \Phi_H^D(\Delta) + s \tag{21}$$

$$r - r' \leq \Phi_{[y_1 \mapsto v_1, \ldots, y_n \mapsto v_n], H}(y_1{:}T_1, \ldots, y_n{:}T_n) + \Phi_H^D(\Delta) + s - (\Phi_{H'}(v{:}T) + \Phi_{H'}^D(\Delta') + s') \tag{22}$$

Now let

$$(u, u') = M_1^{\mathrm{fun}_n} \cdot (\Phi_{[y_1 \mapsto v_1, \ldots, y_n \mapsto v_n], H}(y_1{:}T_1, \ldots, y_n{:}T_n) + \Phi_H^D(\Delta) + s, \Phi_{H'}(v{:}T) + \Phi_{H'}^D(\Delta') + s') \cdot M_2^{\mathrm{fun}_n} \tag{23}$$

Per definition and since $s' \geq M_2^{\mathrm{fun}_n}$, it follows that

$$u = \max(0, M_1^{\mathrm{fun}_n} + \Phi_{[y_1 \mapsto v_1, \ldots, y_n \mapsto v_n], H}(y_1{:}T_1, \ldots, y_n{:}T_n) + \Phi_H^D(\Delta) + s)$$

From Proposition 3.01 applied to (21), (23), and (19) we derive $u \geq p$. If

$$M_1^{\mathrm{fun}_n} + \Phi_{[y_1 \mapsto v_1, \ldots, y_n \mapsto v_n], H}(y_1{:}T_1, \ldots, y_n{:}T_n) + \Phi_H^D(\Delta) + s \leq 0$$

then $u = p = 0$ and

$$q + \Phi_{[y_1 \mapsto v_1, \ldots, y_n \mapsto v_n], H}(y_1{:}T_1, \ldots, y_n{:}T_n) + \Phi_H^D(\Delta) \geq p$$

trivially holds. Otherwise it then follows from (20) that

$$\begin{aligned} q + \Phi_{[y_1\mapsto v_1,\ldots,y_n\mapsto v_n],H}(y_1{:}T_1,\ldots,y_n{:}T_n) + \Phi^D_H(\Delta) & \\ = s + M_1^{\mathrm{fun}_n} + q + & \\ \Phi_{[y_1\mapsto v_1,\ldots,y_n\mapsto v_n],H}(y_1{:}T_1,\ldots,y_n{:}T_n) + \Phi^D_H(\Delta) & \\ = u \geq p & \end{aligned}$$

Similarly, we apply Proposition 3.01 to (19) and use (22) and (20) to see that

$$\begin{aligned} p - p &= r - r' + M_1^{\mathrm{fun}_n} + M_2^{\mathrm{fun}_n} \\ &\leq \Phi_{[y_1\mapsto v_1,\ldots,y_n\mapsto v_n],H}(y_1{:}T_1,\ldots,y_n{:}T_n) \\ &\quad + \Phi^D_H(\Delta) + s - (\Phi_{H'}(v{:}T) + \Phi^D_{H'}(\Delta') + s') \\ &\quad + M_1^{\mathrm{fun}_n} + M_2^{\mathrm{fun}_n} \\ &\leq \Phi_{[y_1\mapsto v_1,\ldots,y_n\mapsto v_n],H}(y_1{:}T_1,\ldots,y_n{:}T_n) \\ &\quad + \Phi^D_H(\Delta) + (s + M_1^{\mathrm{fun}_n}) \\ &\quad - (\Phi_{H'}(v{:}T) + \Phi^D_{H'}(\Delta') + (s' - M_2^{\mathrm{fun}_n})) \\ &= \Phi_{[y_1\mapsto v_1,\ldots,y_n\mapsto v_n],H}(y_1{:}T_1,\ldots,y_n{:}T_n) \\ &\quad + \Phi^D_H(\Delta) + q \\ &\quad - (\Phi_{H'}(v{:}T) + \Phi^D_{H'}(\Delta') + q') \end{aligned}$$

(L:Mat) Assume that the type derivation of e ends with an application of the rule L:Mat. Then e is a pattern match of the form match x with $c_i\,\langle x_1,\ldots,x_k\rangle \Rightarrow e_1 \mid e_2$ whose evaluation ends with an application of the rule E:Mat1 or E:Mat2. The latter case follows a similar argument to the case (L:App). So assume the derivation of the evaluation judgement ends with an application of E:Mat1.
Then $V(x) = \ell$, $H(\ell) = (\mathsf{constr}_{c_i}, v_1, \ldots, v_k)$, and $F, V', H_M \vdash e_1 \Downarrow (v, H') \mid (r, r')$ for $V' = V[x_1 \mapsto v_1, \ldots, x_k \mapsto v_k]$ and some r, r' with

$$(p, p') = M_1^{\mathrm{matT}} \cdot (r, r') \cdot M_2^{\mathrm{matT}} \tag{24}$$

Since the derivation of $\Sigma; \Gamma \left|\frac{q}{q'}\right. e{:}T$ ends with an application of L:Mat, we have $\Gamma = \Gamma',\ T' = \mu X^T.\{\cdots \mid c_i : (B_1,\ldots,B_k) \mid \ldots\},\ x{:}T'$,

$$\Sigma; \Gamma, x : T' \left|\frac{s}{s'}\right. \text{match } x \text{ with } c_i\,\langle x_1,\ldots,x_k\rangle \Rightarrow e_1 \mid e_2 : T$$

and

$$q = s + M_1^{\mathrm{matT}} - T[i] \text{ and } q' = s' - M_2^{\mathrm{matT}} \tag{25}$$

Note that $H, \Delta \vDash V' : (\Gamma', x_1{:}[T'/X]B_1, \ldots, x_k{:}[T'/X]B_k)$ holds by (L:Constr).

By the definition of Φ it holds that $\Phi_H(v{:}T') = T[i] + \sum_{j=1}^{k} \Phi_H(v_j{:}T_j)$ and therefore

$$\Phi_{V,H}(\Gamma) = T[i] + \Phi_{V',H}(\Gamma', x_1{:}[T'/X]B_1, \ldots, x_k{:}[T'/X]B_k) \tag{26}$$

Since $H, \Delta \vDash V' : \Gamma', x_1{:}[T/X]B_1, \ldots, x_k{:}[T/X]B_k$ we can apply the induction hypothesis to $F, V', H_M \vdash (v, H') \mid (r, r')$ and obtain (with (26))

$$r \leq \Phi_{V,H}(\Gamma) + \Phi^D_H(\Delta) - T[i] + s \tag{27}$$

$$\begin{aligned} r - r' \leq \Phi_{V,H}(\Gamma) + \Phi^D_H(\Delta) - T[i] + s \\ - (\Phi_{H'}(v{:}T) + \Phi^D_{H'}(\Delta') + s') \end{aligned} \tag{28}$$

Note that $\Phi_{V,H}(\Gamma) - T[i] \geq 0$ and let

$$\begin{aligned} (u, u') = M_1^{\mathrm{matT}} \cdot (\Phi_{V,H}(\Gamma) + \Phi^D_H(\Delta) - T[i] + s, \\ \Phi_{H'}(v{:}T) + \Phi^D_{H'}(\Delta') + s') \cdot M_2^{\mathrm{matT}} \end{aligned} \tag{29}$$

Per definition and from (25) it follows that

$$u = \max(0, \Phi_{V,H}(\Gamma) + \Phi^D_H(\Delta) - T[i] + s + M_1^{\mathrm{matT}})$$

From Proposition 3.01 applied to (27), (29), and (24) we derive $u \geq p$. If $\Phi_{V,H}(\Gamma) + \Phi^D_H(\Delta) - T[i] + s + M_1^{\mathrm{matT}} \leq 0$ then $u = p \geq 0$ trivially holds. If $\Phi_{V,H}(\Gamma) + \Phi^D_H(\Delta) - T[i] + s + M_1^{\mathrm{matT}} > 0$ then it follows from (25) that

$$\begin{aligned} q + \Phi_{V,H}(\Gamma) + \Phi^D_H(\Delta) \\ = \Phi_{V,H}(\Gamma) + \Phi^D_H(\Delta) - T[i] + s + M_1^{\mathrm{matT}} \\ = u \geq p \end{aligned}$$

Finally, we apply Proposition 3.01 to (24) to see that

$$\begin{aligned} p - p' &= r - r' + M_1^{\mathrm{matT}} + M_2^{\mathrm{matT}} \\ &\overset{(28)}{\leq} \Phi_{V,H}(\Gamma) + \Phi^D_H(\Delta) - T[i] + s \\ &\quad - (\Phi_{H'}(v{:}T) + \Phi^D_{H'}(\Delta') + s') + M_1^{\mathrm{matT}} + M_2^{\mathrm{matT}} \\ &= \Phi_{V,H}(\Gamma) + \Phi^D_H(\Delta) + (s + M_1^{\mathrm{matT}} - T[i]) \\ &\quad - (\Phi_{H'}(v{:}T) + \Phi^D_{H'}(\Delta') + (s' - M_2^{\mathrm{matT}})) \\ &\overset{(25)}{\leq} \Phi_{V,H}(\Gamma) + \Phi^D_H(\Delta) + q \\ &\quad - (\Phi_{H'}(v{:}T) + \Phi^D_{H'}(\Delta') + q') \end{aligned}$$

(L:Let) If the type derivation ends with an application of L:Let then e is a let expression of the form let $x = e_1$ in e_2 that has eventually been evaluated

with the rule E:Let. Then it follows that $F, V, H_m \vdash e_1 \Downarrow (v_1, H_1) \mid (r, r')$ and $F, V', H_{1M} \vdash e_2 \Downarrow (v_2, H_2) \mid (t, t')$ for $V' = V[x \mapsto v_1]$ and r, r', t, t' with

$$(p, p') = M_1^{\text{let}} \cdot (r, r') \cdot M_2^{\text{let}} \cdot (t, t') \cdot M_3^{\text{let}} \tag{30}$$

The derivation of the type judgement for e ends with an application of L:Let. Hence $\Gamma = \Gamma_1, \Gamma_2,\ \Sigma; \Gamma_1 \left|\frac{s_1}{s_1'}\right. e_1 : T',\ \Sigma; \Gamma_2, x{:}T' \left|\frac{s_2}{s_2'}\right. e_2 : T$, and

$$q = s_1 + M_1^{\text{let}} \tag{31}$$

$$s_1' = s_2 + M_2^{\text{let}} \tag{32}$$

$$q' = s_2' - M_3^{\text{let}} \tag{33}$$

It follows from the definition of Φ that

$$\Phi_{V,H}(\Gamma) = \Phi_{V,H}(\Gamma_1) + \Phi_{V,H}(\Gamma_2) \tag{34}$$

Since $H, \Delta \vDash V : \Gamma$ we also have $H, \Delta \vDash V : \Gamma_1$ and can thus apply the induction hypothesis for the evaluation judgement for e_1 to derive

$$r \leq \Phi_{V,H}(\Gamma_1) + \Phi_H^D(\Delta) + s_1 \tag{35}$$

$$r - r' \leq \Phi_{V,H}(\Gamma_1) + \Phi_H^D(\Delta) + s_1 - (\Phi_{H_1}(v_1{:}T') + \Phi_{H_1}^D(\Delta') + s_1') \tag{36}$$

and $H_1, \Delta \vDash V : \Gamma_1,\ H_1, \Delta_{\text{ret}} \vDash v : T'$, and $\Delta' = \Delta_1 \otimes \Delta_{\text{ret}}$.
We now case for each location reachable from the variables in Γ_2 on whether or not it is reachable from the variables in Γ_1.

- If ℓ is reachable from Γ_1 and has type T, we get from $H', \Delta \vDash V : \Gamma_1$ that $H', \Delta \vDash \ell : T$.
- Otherwise, if ℓ is not reachable from Γ_1 and has type T, its underlying value must not have been changed by the evaluation of e_1, and therefore $H', \Delta \vDash \ell : T$ holds.

Since $H', \Delta \vDash \ell : T$ holds for all locations ℓ reachable from Γ_2, it follows that $H_1, \Delta \vDash V : \Gamma_2$.
From the above and $H_1, \Delta_{\text{ret}} \vDash v : T'$, we get that $H_1, \Delta' \vDash V' : \Gamma_2, x{:}T'$ and thus again by induction

$$t \leq \Phi_{V,H}(\Gamma_2, x{:}T') + \Phi_{H_1}^D(\Delta') + s_2 \tag{37}$$

$$t - t' \leq \Phi_{V,H}(\Gamma_2, x{:}T') + \Phi_{H_1}^D(\Delta') + s_2 - (\Phi_{H_2}(v_2{:}T) + \Phi_{H_2}^D(\Delta'') + s_2') \tag{38}$$

Now let

$$\begin{aligned}(u,u') =& M_1^{\text{let}} \cdot (\Phi_{V,H}(\Gamma_1) + \Phi_H^D(\Delta) + s_1,\\ & \Phi_{H_1}(v_1{:}T') + \Phi_{H_1}^D(\Delta') + s_1')\cdot\\ & M_2^{\text{let}} \cdot (\Phi_{V,H}(\Gamma_2, x{:}T') + \Phi_{H_1}^D(\Delta') + s_2,\\ & \Phi_{H_2}(v_2{:}T) + \Phi_{H_2}^D(\Delta'') + s_2') \cdot M_3^{\text{let}}\end{aligned}$$

Then it follows that

$$\begin{aligned}(u,u) \overset{(32,33)}{=}& M_1^{\text{let}} \cdot (\Phi_{V,H}(\Gamma_1) + \Phi_H^D(\Delta) + s_1,\\ & \Phi_{H_1}(v_1{:}T') + \Phi_{H_1}^D(\Delta') + s_1' - M_2^{\text{let}})\cdot\\ & (\Phi_{V,H}(\Gamma_2, x{:}T') + \Phi_{H_1}^D(\Delta') + s_2,\\ & \Phi_{H_2}(v_2{:}T) + \Phi_{H_2}^D(\Delta'') + s_2' - M_3^{\text{let}})\\ =& \quad M_1^{\text{let}} \cdot (v + \Phi_{V,H}(\Gamma_1) + \Phi_H^D(\Delta) + s_1, v')\end{aligned}$$

for $v, v' \in \mathbb{Q}_0^+$ with

$$\begin{aligned}v \;\leq\;& \Phi_{V',H_1}(\Gamma_2, x{:}T') + \Phi_{H_1}^D(\Delta') + s_2\\ & - (\Phi_{H_1}(v_1{:}T') + \Phi_{H_1}^D(\Delta') + s_1' - M_2^{\text{let}})\\ =\;& \Phi_{V',H_1}(\Gamma_2) + \Phi_{H_1}^D(\Delta') + s_2\\ & - (\Phi_{H_1}^D(\Delta') + s_1' - M_2^{\text{let}})\\ \overset{(32)}{=}\;& \Phi_{V',H_1}(\Gamma_2) + \Phi_{H_1}^D(\Delta') - \Phi_{H_1}^D(\Delta')\\ =\;& \Phi_{V',H_1}(\Gamma_2)\end{aligned}$$

and thus

$$\begin{aligned}u \leq\; & \max(0,\\ & \Phi_{V,H}(\Gamma_1) + \Phi_H^D(\Delta) + \Phi_{V',H_1}(\Gamma_2) + s_1 + M_1^{\text{let}})\\ \overset{(Lem.3)}{\leq}\; & \max(0,\\ & \Phi_{V,H}(\Gamma_1) + \Phi_H^D(\Delta) + \Phi_{V,H_1}(\Gamma_2) + s_1 + M_1^{\text{let}})\\ \overset{(31)}{\leq}\; & \Phi_{V,H}(\Gamma) + \Phi_H^D(\Delta) + q\end{aligned}$$

Finally, it follows with Proposition 3.01 applied to (35), (37), and (30) that $u \geq p$.
For the second part of the statement we apply Proposition 3.01 to (30) and derive the following.

$$
\begin{aligned}
p - p' &= r - r' + t - t' + M_1^{\mathrm{let}} + M_2^{\mathrm{let}} + M_3^{\mathrm{let}} \\
&\overset{(38,36)}{\leq} \Phi_{V,H}(\Gamma_1) + \Phi_H^D(\Delta) + s_1 \\
&\qquad - (\Phi_{H_1}(v_1{:}T') + \Phi_{H_1}^D(\Delta) + s_1') \\
&\qquad + \Phi_{V',H}(\Gamma_2, x{:}T') + \Phi_{H_1}^D(\Delta') + s_2 \\
&\qquad - (\Phi_{H_2}(v_2{:}T) + \Phi_{H_2}^D(\Delta'') + s_2') \\
&\qquad + M_1^{\mathrm{let}} + M_2^{\mathrm{let}} + M_3^{\mathrm{let}} \\
&= \Phi_{V,H}(\Gamma_1) + \Phi_{V',H}(\Gamma_2) + \Phi_H^D(\Delta) + \Phi_{H_1}^D(\Delta') \\
&\qquad - \Phi_{H_1}^D(\Delta') + s_1 + (s_2 + M_2^{\mathrm{let}} - s_1') \\
&\qquad - (\Phi_{H_2}(v_2{:}T) + \Phi_{H_2}^D(\Delta'') + s_2') \\
&\qquad + M_1^{\mathrm{let}} + M_3^{\mathrm{let}} \\
&= \Phi_{V,H}(\Gamma_1) + \Phi_{V',H}(\Gamma_2) + \Phi_H^D(\Delta) \\
&\qquad + s_1 + (s_2 + M_2^{\mathrm{let}} - s_1') \\
&\qquad - (\Phi_{H_2}(v_2{:}T) + \Phi_{H_2}^D(\Delta'') + s_2') \\
&\qquad + M_1^{\mathrm{let}} + M_3^{\mathrm{let}} \\
&\overset{(32)}{=} \Phi_{V,H}(\Gamma_1) + \Phi_{V',H}(\Gamma_2) + \Phi_H^D(\Delta) + s_1 \\
&\qquad - (\Phi_{H_2}(v_2{:}T) + \Phi_{H_2}^D(\Delta'') + s_2') \\
&\qquad + M_1^{\mathrm{let}} + M_3^{\mathrm{let}} \\
&\overset{(Lem.3)}{\leq} \Phi_{V,H}(\Gamma_1) + \Phi_{V,H}(\Gamma_2) + \Phi_H^D(\Delta) + s_1 \\
&\qquad - (\Phi_{H_2}(v_2{:}T) + \Phi_{H_2}^D(\Delta'') + s_2') \\
&\qquad + M_1^{\mathrm{let}} + M_3^{\mathrm{let}} \\
&= \Phi_{V,H}(\Gamma) + \Phi_H^D(\Delta) + s_1 + M_1^{\mathrm{let}} \\
&\qquad - (\Phi_{H_2}(v_2{:}T) + \Phi_{H_2}^D(\Delta'') + s_2' - M_3^{\mathrm{let}}) \\
&\overset{(31,33)}{\leq} \Phi_{V,H}(\Gamma) + \Phi_H^D(\Delta) + q \\
&\qquad - (\Phi_{H_2}(v_2{:}T) + \Phi_{H_2}^D(\Delta'') + q')
\end{aligned}
$$

(L:Ref) Assume that the type derivation ends with an application of the rule L:Ref. Then e has the form ref x and the evaluation consists of an application of the rule E:Ref. Assume first that $M^{\mathrm{ref}} \geq 0$. Then it follows by definition that $p = M^{\mathrm{ref}}$ and $p' = 0$. Thus we have $0 \leq q' = q - M^{\mathrm{ref}}$ and therefore $p = M^{\mathrm{ref}} \leq q \leq \Phi_{V,H}(x{:}T) + \Phi_H^D(\Delta) + q$.

Furthermore it follows from E:Ref that we have ℓ with $H'(\ell) = V(x)$; by definition of Φ, $\Phi_{H'}(\ell{:}T \text{ ref}) = 0$. In this case, H' differs from H and Δ' from Δ by the addition of the extra reference, so $\Phi_{H'}^D(g') = \Phi_H^D(\Delta) + \Phi_{V,H}(x{:}T)$.

Thus

$$\begin{aligned} p - p' = M^{\mathrm{ref}} &= q - q' \\ &= q - q' + (\Phi_H^D(\Delta) + \Phi_{V,H}(x{:}T) - \Phi_{H'}^D(\Delta')) \\ &\quad - \Phi_{H'}(\ell{:}T\ \mathrm{ref}) \\ &= \Phi_{V,H}(x{:}T) + \Phi_H^D(\Delta) + q \\ &\quad - (\Phi_{H'}(\ell{:}T\ \mathrm{ref}) + \Phi_{H'}^D(\Delta') + q') \end{aligned}$$

Assume now that $M^{\mathrm{ref}} < 0$. Then it follows by definition that $p = 0$ and $p' = -M^{\mathrm{ref}}$. Thus $p = 0 \leq \Phi_{V,H}(x{:}T) + \Phi_H^D(\Delta) + q$. We have again that $q - q' = M^{\mathrm{ref}} = p - p'$ and the claim follows as above.

(L:DRef) Assume that the type derivation ends with an application of the rule L:DRef. Then e has the form $!x$ and the evaluation consists of an application of the rule E:DRef. Assume first that $M^{\mathrm{dref}} \geq 0$. Then it follows by definition that $p = M^{\mathrm{dref}}$ and $p' = 0$. Thus we have $0 \leq q' = q - M^{\mathrm{dref}}$ and therefore $p = M^{\mathrm{dref}} \leq q \leq \Phi_{V,H}(x{:}T'\ \mathrm{ref}) + \Phi_H^D(\Delta) + q$.
Furthermore it follows from E:DRef that we have $\ell = H(V(x))$ such that $H \vDash \ell : T'$. If T' is a list type, then for $|T| = |T'|$ and $T \not\curlyvee (T, T)$ to hold, it must follow that $\Phi_H(\ell : T) = 0$. Otherwise by definition of Φ, $\Phi_H(\ell : T) = 0$ holds. We note that since x points to ℓ, ℓ must have been already included in Δ and therefore $\Delta' = \Delta$ in this case. Thus,

$$\begin{aligned} p - p' = M^{\mathrm{dref}} &= q - q' \\ &\leq \Phi_{V,H}(x{:}T'\ \mathrm{ref}) + \Phi_H^D(\Delta) + q \\ &\quad - (\Phi_H(\ell{:}T) + \Phi_H^D(\Delta) + q') \end{aligned}$$

Assume now that $M^{\mathrm{dref}} < 0$. Then it follows by definition that $p = 0$ and $p' = -M^{\mathrm{dref}}$. Thus $p = 0 \leq \Phi_{V,H}(x{:}T'\mathrm{ref}) + \Phi_H^D(\Delta) + q$. We have again that $q - q' = M^{\mathrm{dref}} = p - p'$ and the claim follows as above.

(L:Assign) Assume that the type derivation ends with an application of the rule L:Assign. Then e has the form $x_1 := x_2$ and the evaluation consists of an application of the rule E:Assign. Assume first that $M^{\mathrm{assign}} \geq 0$. Then it follows by definition that $p = M^{\mathrm{assign}}$ and $p' = 0$. Thus we have $0 \leq q' = q - M^{\mathrm{assign}}$ and therefore $p = M^{\mathrm{set}} \leq q \leq \Phi_{V,H}(x_1{:}T\ \mathrm{ref}, x_2{:}T) + \Phi_H^D(\Delta) + q$. Let e be the potential reachable by following references from x_1 but not reachable by following x_2; intuitively, these are the locations that we lose access to when overwriting x_1. Thus,

$$\Phi_{V,H}(x_1{:}T\ \mathrm{ref}, x_2{:}T) + \Phi_H^D(\Delta) = \Phi_{H'}^D(\Delta) + e \tag{39}$$

and therefore

$$\Phi_{V,H}(x_1{:}T\ \mathrm{ref}, x_2{:}T) + \Phi_H^D(\Delta) \geq \Phi_{H'}^D(\Delta) \tag{40}$$

since we assign $V(x_2)$ to be pointed to by x_1 under H'. Furthermore, $\ell = \mathrm{Null}$ and therefore $\Phi_{H'}(\ell{:}1) = 0$; it follows that $\Delta' = \Delta$ in this case since Null

does not contribute to the reference collection. Thus,

$$\begin{aligned} p - p' &= M^{\text{assign}} = q - q' \\ &\overset{(40)}{\leq} (\Phi_{V,H}(x_1{:}T \text{ ref}, x_2{:}T) + \Phi_H^D(\Delta) - \Phi_{H'}^D(\Delta)) \\ &\quad + q - q' \\ &= \Phi_{V,H}(x_1{:}T \text{ ref}, x_2{:}T) + \Phi_H^D(\Delta) + q \\ &\quad - (\Phi_{H'}(\ell{:}1) + \Phi_{H'}^D(\Delta) + q') \end{aligned}$$

Assume now that $M^{\text{assign}} < 0$. Then it follows by definition that $p = 0$ and $p' = -M^{\text{assign}}$. Thus $p = 0 \leq \Phi_{V,H}(x_1{:}T \text{ ref}, x_2{:}T) + \Phi_H^D(\Delta) + q$. We have again that $q - q' = M^{\text{assign}} = p - p'$ and the claim follows as above.

(L:Swap) The L:Swap case follows as in the L:Assign case with $\ell = H(V(x_1))$. As a result, no locations become unreachable, and therefore

$$\begin{aligned} &\Phi_{V,H}(x_1{:}T \text{ ref}, x_2{:}T) + \Phi_H^D(\Delta) \\ &\qquad = \Phi_{H'}(\ell{:}T) + \Phi_{H'}^D(\Delta') \end{aligned} \quad (41)$$

since we assign $V(x_2)$ to be pointed to by x_1 under H' and set ℓ to be the target of x_1 under H.
Thus,

$$\begin{aligned} p - p' &= M^{\text{swap}} = q - q' \\ &\overset{(41)}{=} (\Phi_{V,H}(x_1{:}T \text{ ref}, x_2{:}T) + \Phi_H^D(\Delta) - \Phi_{H'}(\ell{:}T) \\ &\quad - \Phi_{H'}^D(\Delta')) + q - q' \\ &= \Phi_{V,H}(x_1{:}T \text{ ref}, x_2{:}T) + \Phi_H^D(\Delta) + q \\ &\quad - (\Phi_{H'}(\ell{:}T) + \Phi_{H'}^D(\Delta') + q') \end{aligned}$$

and the argument follows as in the L:Assign case.

(L:Nat) Assume that e is a natural number n that has been evaluated with the rule E:Nat. Assume first that $M^{\text{nat}} \geq 0$. Then it follows by definition that $p = M^{\text{nat}}$ and $p' = 0$. The type judgement $\Sigma; \emptyset \vdash^{q}_{q'} n : N$ has been derived by a single application of the rule L:Nat. Thus we have $0 \leq q' = q - M^{\text{nat}}$ and $p = M^{\text{nat}} \leq q \leq \Phi_{V,H}(\emptyset) + \Phi_H^D(\Delta) + q$. In addition, it follows from E:Nat that $v = n$ and thus $p - p' = M^{\text{nat}} = q - q' = \Phi_{V,H}(\emptyset) + \Phi_H^D(\Delta) + q - (\Phi_{V,H}(n{:}N) + \Phi_H^D(\Delta) + q')$. Note that Δ does not change under this evaluation since n is not of a reference type.
Assume now that $M^{\text{nat}} < 0$. Then it follows by definition that $p = 0$ and $p' = -M^{\text{nat}}$. Thus $p = 0 \leq \Phi_{V,H}(\emptyset) + \Phi_H^D(\Delta) + q$. We have again that $q - q' = M^{\text{nat}} = p - p'$. Therefore the second part of the claim follows as in the previous case.

(L:Create) Assume that the type derivation ends with an application of the rule L:Create. Then e has the form $\text{create}(x_1, x_2)$ and the evaluation consists of an application of the rule E:Create. Assume first that $M^{\text{create}} \geq 0$. Then

it follows by definition that $p = M^{\text{create}}$ and $p' = 0$. Thus we have $0 \leq q' = q - M^{\text{create}}$ and therefore $p = M^{\text{create}} \leq q \leq \Phi_{V,H}(x_1{:}N, x_2{:}T) + \Phi_H^D(\Delta) + q$. Furthermore it follows from E:Create that we have σ with $\sigma(i) = V(x_2)$ for $i < n$ and ℓ with $H'(\ell) = (\sigma, H(V(x_1)))$; by definition of Φ, $\Phi_{H'}(\ell{:}T \text{ array}) = 0$. In this case, H' differs from H and Δ' from Δ by the addition of the extra array, so $\Phi_{H'}^D(\Delta') = \Phi_H^D(\Delta) + \Phi_{V,H}(x_2{:}T)$. Note that the x_2 term does not get multiplied by n since all array cells initially alias to the same data. Thus

$$\begin{aligned}
p - p' &= M^{\text{ref}} = q - q' \\
&= q - q' + (\Phi_H^D(\Delta) + \Phi_{V,H}(x_2{:}T) - \Phi_{H'}^D(\Delta')) \\
&\quad - \Phi_{H'}(\ell{:}T \text{ array}) \\
&= \Phi_{V,H}(x_2{:}T) + \Phi_H^D(\Delta) + q \\
&\quad - (\Phi_{H'}(\ell{:}T \text{ array}) + \Phi_{H'}^D(\Delta') + q') \\
&= \Phi_{V,H}(x_1{:}N, x_2{:}T) + \Phi_H^D(\Delta) + q \\
&\quad - (\Phi_{H'}(\ell{:}T \text{ array}) + \Phi_{H'}^D(\Delta') + q')
\end{aligned}$$

Assume now that $M^{\text{create}} < 0$. Then it follows by definition that $p = 0$ and $p' = -M^{\text{create}}$. Thus $p = 0 \leq \Phi_{V,H}(x_1{:}N, x_2{:}T) + \Phi_H^D(\Delta) + q$. We have again that $q - q' = M^{\text{create}} = p - p'$ and the claim follows as above.

(L:Get) Assume that the type derivation ends with an application of the rule L:Get. Then e has the form $\text{get}(x_1, x_2)$ and the evaluation consists of an application of the rule E:Get. Assume first that $M^{\text{get}} \geq 0$. Then it follows by definition that $p = M^{\text{get}}$ and $p' = 0$. Thus we have $0 \leq q' = q - M^{\text{get}}$ and therefore $p = M^{\text{get}} \leq q \leq \Phi_{V,H}(x_1{:}T' \text{ array}, x_2{:}N) + \Phi_H^D(\Delta) + q$.
Furthermore it follows from E:Get that we have
$H(V(x_1)) = (\sigma, n)$ and $H(V(x_2)) = i$. Therefore, if $0 \leq i < n$ we have $\ell = \sigma(i)$ such that $H \vDash \ell : T'$; otherwise, the rule does not apply and program execution must have failed as desired. If T' is a list type, then for $|T| = |T'|$ and $T \curlyvee (T, T)$ to hold, it must follow that $\Phi_H(\ell : T) = 0$. Otherwise by definition of Φ, $\Phi_H(\ell : T) = 0$ holds. We note that since x points to an array, which in turn points to ℓ, ℓ must have been already included in Δ and therefore $\Delta' = \Delta$ in this case. Thus,

$$\begin{aligned}
p - p' &= M^{\text{get}} = q - q' \\
&\leq \Phi_{V,H}(x_1{:}T' \text{ array}, x_2{:}N) + \Phi_H^D(\Delta) + q \\
&\quad - (\Phi_H(\ell{:}T) + \Phi_H^D(\Delta) + q')
\end{aligned}$$

Assume now that $M^{\text{get}} < 0$. Then it follows by definition that $p = 0$ and $p' = -M^{\text{get}}$. Thus $p = 0 \leq \Phi_{V,H}(x_1{:}T' \text{ array}, x_2{:}N) + \Phi_H^D(\Delta) + q$. We have again that $q - q' = M^{\text{get}} = p - p'$ and the claim follows as above.

(L:Set) Assume that the type derivation ends with an application of the rule L:Set. Then e has the form
$\text{set}(x_1, x_2, x_3)$ and the evaluation consists of an application of the rule E:Set.

Assume first that $M^{\text{set}} \geq 0$. Then it follows by definition that $p = M^{\text{set}}$ and $p' = 0$. Thus we have $0 \leq q' = q - M^{\text{set}}$ and therefore $p = M^{\text{set}} \leq q \leq \Phi_{V,H}(x_1{:}T \text{ array}, x_2{:}N, x_3{:}T) + \Phi_H^D(\Delta) + q$.
By the premises of E:Set, we know that $H(V(x_1)) = (\sigma, n)$ and $H(V(x_2)) = i$. Note that $0 \leq i < n$ as otherwise the rule would not apply and program execution must have failed as desired. Let e be the potential reachable by following references from $\sigma(i)$ but not reachable by following $H(V(x_3))$; intuitively, these are the locations that we lose access to when overwriting σ_i. Thus,

$$\Phi_{V,H}(x_1{:}T \text{ array}, x_2{:}N, x_3{:}T) + \Phi_H^D(\Delta) = \Phi_{H'}^D(\Delta) + e \quad (42)$$

and therefore

$$\Phi_{V,H}(x_1{:}T \text{ array}, x_2{:}N, x_3{:}T) + \Phi_H^D(\Delta) \geq \Phi_{H'}^D(\Delta) \quad (43)$$

since we assign $V(x_3)$ to be pointed to by $\sigma'(i)$ under H'. Furthermore, $\ell = \text{Null}$ and therefore $\Phi_{H'}(\ell{:}1) = 0$; it follows that $\Delta' = \Delta$ in this case since Null does not contribute to the reference collection. Thus,

$$\begin{aligned} p - p' = M^{\text{set}} &= q - q' \\ &\overset{(43)}{\leq} (\Phi_{V,H}(x_1{:}T \text{ array}, x_2{:}N, x_3{:}T) + \Phi_H^D(\Delta) \\ &\quad - \Phi_{H'}^D(\Delta)) + q - q' \\ &= \Phi_{V,H}(x_1{:}T \text{ array}, x_2{:}N, x_3{:}T) + \Phi_H^D(\Delta) \\ &\quad + q - (\Phi_{H'}(\ell{:}1) + \Phi_{H'}^D(\Delta) + q') \end{aligned}$$

Assume now that $M^{\text{set}} < 0$. Then it follows by definition that $p = 0$ and $p' = -M^{\text{set}}$. Thus $p = 0 \leq \Phi_{V,H}(x_1{:}T \text{ array}, x_2{:}N, x_3{:}T) + \Phi_H^D(\Delta) + q$. We have again that $q - q' = M^{\text{set}} = p - p'$ and the claim follows as above.

(L:Aswap) The L:Aswap case follows as in the L:Set case with $H(V(x_1)) = (\sigma, n)$, $H(V(x_2)) = i$, and $\ell = \sigma(i)$. As a result, no locations become unreachable, and
therefore

$$\Phi_{V,H}(x_1{:}T \text{ array}, x_2{:}N, x_3{:}T) + \Phi_H^D(\Delta) = \Phi_{H'}(\ell{:}T) + \Phi_{H'}^D(\Delta') \quad (44)$$

holds since we assign $V(x_3)$ to be pointed to by $\sigma'(i)$ under H' and set ℓ to be the target of $\sigma(i)$ under H.

Thus,

$$\begin{aligned} p - p' = M^{\text{aswap}} &= q - q' \\ &\overset{(44)}{=} (\Phi_{V,H}(x_1{:}T \text{ array}, x_2{:}N, x_3{:}T) + \Phi_H^D(\Delta) \\ &\quad - \Phi_{H'}(\ell{:}T) - \Phi_{H'}^D(\Delta')) + q - q' \\ &= \Phi_{V,H}(x_1{:}T \text{ array}, x_2{:}N, x_3{:}T) + \Phi_H^D(\Delta) + q \\ &\quad - (\Phi_{H'}(\ell{:}T) + \Phi_{H'}^D(\Delta') + q') \end{aligned}$$

and the argument follows as in the L:Set case.