Type-Based Amortised Heap Space Analysis

Complete Soundness Proof

Martin Hofmann Steffen Jost Dulma Rodriguez

January 30, 2009

1 Introduction

The prediction of resource consumption in programs has gained interest in the last years. It is important for a number of areas, in particular embedded systems and mobile computing. A variety of approaches to resource analysis have been proposed based in particular on recurrence solving [AAG+07, Gro01], abstract interpretation [GL98, NCQR05], sized types [HP99], and amortised analysis [HJ03, HJ06, Cam08].

The amortised approach which the present paper belongs to is particularly useful in situations where heap-allocated data structures whose size is proportional to parts of the input must be costed. Typical examples are various sorting algorithms where trees, lists, or heaps appear as intermediate data structures. In such cases amortised analysis can infer very good bounds based on intuitive programmer annotations in the form of types and the solution of linear inequations.

In [HJ06] amortised analysis has been applied to a Java-like class-based object-oriented language without garbage collection, but with explicit deallocation in the style of C's `free()`. Such programs may be evaluated by maintaining a set of free memory units, the freelist. Upon object creation a number of heap units required to store the object is taken from the freelist provided it contains enough units; each deallocated heap unit is returned to the freelist. An attempt to create a new object with an insufficient freelist causes unsuccessful abortion of the program.

The goal is to predict a bound to the initial size that the freelist must have so that a given program may be executed without causing unsuccessful abortion due to penury of memory. This has been achieved using a modified amortized analysis [Tar85, Oka98], distinguished from previous work by being reference based.

Essentially each object is ascribed an abstracted portion of the freelist, referred to as *potential*. The potential is just a number, denoting the size of freelist portion associated with the object. Any object creation must be paid for from the potential in scope. The initial potential thus furnishes an upper bound on the total heap consumption.

1.1 Views and potential

We aim at assigning objects a potential so that any object creation must be paid for from the potential in scope and the potential of the initial configuration furnishes an upper bound on the total heap consumption. For an interesting analysis we should be able to assign objects of the same class different potentials. Thus, we need more refined types than classes to assign potential to. To this end, we introduce the *views*. A view on an object shall determine its contribution to the potential. A refined type then consists of a class C and a view r and is written C^r.

The meaning of views is given by three maps $\Diamond()$ defining potentials, A defining views of attributes, and M defining refined method types. More precisely, $\Diamond() : \text{Class} \times \text{View} \to \mathbb{Q}^+$ assigns each class its potential according to the employed view. $\mathsf{A} : \text{Class} \times \text{View} \times \text{Field} \to \text{View} \times \text{View}$ determines the refined types of the fields. A different view may apply according to whether a field is read from (get-view) or written to (set-view). $\mathsf{M} : \text{Class} \times \text{View} \times \text{Method} \to \mathcal{P}(\text{Views of Arguments} \to \text{Effect} \times \text{View of Result})$ assigns refined types and effects to methods. The effect is a pair of numbers representing the potential consumed before and released after method invocation.

1.2 Example: Copying singly-linked lists

Suppose we have defined a class of singly-linked lists in an object-oriented style which harnesses dynamic dispatch to obtain the functionality of pattern-matching.

```
abstract class List { abstract List copy(); }
class Nil  extends List { List copy() { return this; }}
class Cons extends List { int elem; List next;
      List copy() { Cons res = new Cons(); res.elem = this.elem;
                    res.next = this.next.copy(); return res; }}
```

It is clear that the memory consumption of a call `x.copy()` will equal the length of the list `x`. To calculate this formally we construct a view rich which assigns to `List` itself the potential 0, to `Nil` the potential 0 and to `Cons` the potential 1. Another view is needed to describe the result of `copy()` for otherwise we could repeatedly copy lists without paying for it. Thus, we introduce another view poor that assigns potential 0 to all classes. The complete specification of the two views is shown here:

$\Diamond(\cdot)$	rich	poor
`List`	0	0
`Nil`	0	0
`Cons`	1	0

	$\mathsf{Cons}^{\mathsf{rich}}$	$\mathsf{Cons}^{\mathsf{poor}}$
$\mathsf{A}^{\mathrm{get}}(\cdot\,, \mathsf{next})$	rich	poor
$\mathsf{A}^{\mathrm{set}}(\cdot\,, \mathsf{next})$	rich	poor

$$\mathsf{M}\big(\{\mathsf{List}^{\mathsf{rich}}, \mathsf{Cons}^{\mathsf{rich}}, \mathsf{Nil}^{\mathsf{rich}}\}, \mathsf{copy}\big) = () \xrightarrow{0/0} \mathsf{poor}$$

The call `x.copy()` is well-typed and of type $\mathsf{List}^{\mathsf{poor}}$ if `x` has refined type $\mathsf{List}^{\mathsf{rich}}$, $\mathsf{Nil}^{\mathsf{rich}}$ or $\mathsf{Cons}^{\mathsf{rich}}$. It is ill-typed if `x` has refined type, e.g., $\mathsf{List}^{\mathsf{poor}}$.

Its effect 0/0 will not decrement the freelist beyond the amount implicit in the potential of `x` (which equals in this case the length of the list pointed to by `x`) and will not return anything to the freelist beyond the amount implicit in the potential of the result (which equals zero due to its refined type $\mathsf{List}^{\mathsf{poor}}$). Thus, the typing amounts to saying that the memory consumption of this call is equal to the length of `x`.

Let us explain why the potential of `x` indeed equals its length. Suppose for the sake of the example that `x` points to a list of length 2. The potential is worked out as the sum over all access paths emanating from `x` and not leading to `null` or being undefined. In this case, these access paths are $\vec{p}_1 = \texttt{x}, \vec{p}_2 = \texttt{x.next}$, $\vec{p}_3 = \texttt{x.next.next}$. Each of these has a dynamic type: `Cons` for $\vec{p}_1$ and $\vec{p}_2$; `Nil` for $\vec{p}_3$. Each of them also has a view worked out by chaining the view of `x` along the get-views. Here it is view `a` in each case. For each access paths we now look up the potential annotation of its dynamic type under its view. It equals 1 in case of $\vec{p}_1$ and $\vec{p}_2$ given $\Diamond\big(\mathsf{Cons}^{\mathsf{rich}}\big) = 1$ and 0 for $\vec{p}_3$ by $\Diamond\big(\mathsf{Nil}^{\mathsf{rich}}\big) = 0$, yielding a sum of 2. This example and some others are explained in more detail in [HJ06].

1.3 Preliminaries

We denote the nonnegative rationals by $\mathbb{Q}^+$. We put $\mathbb{D} = \mathbb{R}_0^+ \cup \{\infty\}$, i.e., the set of nonnegative real numbers together with an element ∞. Ordering and addition on $\mathbb{R}_0^+$ extend to $\mathbb{D}$ by $\infty + x = x + \infty = \infty$ and $x \leq \infty$. If U is a subset of $\mathbb{D}$ we write $\sum U$ for the (possibly infinite) sum over all its elements. Since $\mathbb{D}$ contains no negative numbers, questions of ordering and non-absolute convergence do not play a role; any subset of $\mathbb{D}$ has a sum, perhaps ∞. We write $\sum_{i\in I} x_i$ for $\sum\{x_i \mid i{\in}I\}$ and use other familiar notations.

The disjoint union of sets is denoted by $\dot\cup$. Since we are dealing with multisets, we write $\uplus$ for the multiset union in order to distinguish it from the ordinary union of sets $\cup$.

For partial maps, we use the following notations when used to model stacks, heaps, or typing contexts: Let f be a partial map. We abbreviate $x \in \mathrm{dom}(f)$ generally by $x \in f$; and sometimes $f(x)$ by writing f_x. Furthermore we denote by $f[x \mapsto y]$ the partial map that sends x to y and acts like f otherwise. Oppositional, $f \setminus x$ denotes the map which is undefined for x and acts like f otherwise. The empty map $\varnothing$ is often omitted, i.e. $[x \mapsto y]$ denotes the singleton map sending x to y.

2 FJ with Update

Our formal model of Java, FJEU, is an extension of Featherweight Java (FJ) [IPW99] with attribute update, conditional and explicit deallocation. It is thus similar to Flatt et al. Classic Java [FKF98].

An FJEU program $\mathscr{C}$ is a partial finite map from class names to class definitions, which we also refer to as *class table*. Each class table $\mathscr{C}$ implies a subtyping relation $<:$ among the class names in the standard way by inheritance.

$$
\begin{array}{rll}
c ::= & \texttt{class}\ C\ [\texttt{extends}\ D]\ \{A_1; \ldots; A_k; M_1 \cdots M_j\} & \\
A ::= & C\ a & \\
M ::= & C_0\ m(C_1\ x_1, \ldots, C_j\ x_j)\{\texttt{return}\ e;\} & \\
e ::= & x & \text{(Variable)} \\
\mid & \mathsf{null} & \text{(Constant)} \\
\mid & \texttt{new}\ C & \text{(Construction)} \\
\mid & \texttt{free}(x) & \text{(Destruction)} \\
\mid & (C)x & \text{(Cast)} \\
\mid & x.a_i & \text{(Access)} \\
\mid & x.a_i\texttt{<-}x & \text{(Update)} \\
\mid & x.m(x_1, \ldots, x_j) & \text{(Invocation)} \\
\mid & \texttt{if}\ x\ \texttt{instanceof}\ C\ \texttt{then}\ e_1\ \texttt{else}\ e_2 & \text{(Conditional)} \\
\mid & \mathsf{let}\ x\ \texttt{=}\ e_1\ \mathsf{in}\ e_2 & \text{(Let)}
\end{array}
$$

Figure 1: The syntax of FJEU

Throughout the following sections we will consider a fixed (but abitrary) class table $\mathscr{C}$ for the ease of notation. The syntax of FJEU is given in Fig. 1. The let-normal form of terms was merely choosen to eliminate boring redundancies from our proofs.

We will use a couple of shorthand notations: If C is declared by $\texttt{class}\ C\ \texttt{extends}\ D\ \{A_1; \ldots; A_k; M_1 \cdots M_j\}$, we write $\mathsf{S}(C)$ to denote the *super-class* D of C, provided that C has a super-class. We write $\mathsf{A}(C)$ to denote the ordered set of attributes of C, including inherited ones, i.e. $\mathsf{A}(C) := \{A_1, \ldots, A_k\} \dot{\cup} \mathsf{A}(D)$. For the sake of simplicity, attributes may not be redefined or hidden within subclasses. We define $\mathsf{A}(\mathsf{S}(C)) := \emptyset$ in the case that C has no super-class. We write $\mathsf{A}(C, a_i)$ to denote the class type of each attribute a_i of class C. An *access path* is a list of attribute names, written $a.b.\cdots.c = \vec{p}$. It convenient to write $\mathsf{A}(C, \vec{p})$ for the class type reached by following the access path $\vec{p}$, i.e. $\mathsf{A}(C, \vec{p}.b) = \mathsf{A}(\mathsf{A}(C, \vec{p}), a)$. Sometimes we will abbreviate $\mathsf{A}(C, \vec{a})$ further by merely writing $C.\vec{a}$; the reason for introducing the long form will become apparent in Sect. 3. We say that an access path $\vec{q}$ is a proper prefix of a path $\vec{p}$, written $\vec{q} \prec \vec{p}$, if and only if there is a non-empty access path $\vec{a}$ such that $\vec{q}.\vec{a} = \vec{p}$. We write $\vec{q} \preceq \vec{p}$ if $\vec{q}$ is a prefix of $\vec{p}$ or $\vec{q} = \vec{p}$. The empty access path is denoted by ε. If $\mathcal{D}$ is a multiset of classes, we simply write $\mathcal{D}.\vec{p}$ for multiset $\{D.\vec{p} \mid D \in \mathcal{D}\}$. We also use the Kleene star for the set of paths possibly containing repetitions, i.e. $a.b.c.b.c.e \in a(.b.c)^*.e$. We allow ourselves to omit the dots in long access path expressions.

Similarly we write $\mathsf{M}(C)$ to denote the set of all defined method names of C, including inherited ones. For a method m of class C we write $\mathsf{M}_{\mathsf{body}}(C, m)$ to denote the term that comprises the *method body* of method m and $\mathsf{M}(C, m)$ to denote the *method type* α of m in class C. We abbreviate $\mathsf{M}(C, m)$ by writing $C.m$ instead. If m is declared in C by $E_0\ m(E_1\ x_1, \ldots, E_j\ x_j)\{\texttt{return}\ e;\}$ then $\mathsf{M}_{\mathsf{body}}(C, m) = e$ and $C.m = E_1, \ldots, E_j \to E_0$. We may refer to $E_1, \ldots, E_j$ as the method's arguments and E_0 as the method result type. If otherwise m is

not defined in C, then $\mathsf{M_{body}}(C,m) = \mathsf{M_{body}}(D,m)$ and $C.m = D.m$, provided that D is the super class of C. In the case that C has no super-class, we also define $\mathsf{M}(\mathsf{S}(C)) := \emptyset$ for the sake of a uniform notation.

Each class has only one implicit constructor, which sets all class attributes to a nil value. Note that we will also write "e_1; e_2" instead of "$\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$" if x does not appear anywhere else inside the program.

The typing judgement of FJEU takes the form $\Gamma \vdash e : C$ where Γ is a finite partial mapping from identifiers to class names. It is defined as a standard extension of the FJ typing rules and is omitted for lack of space.

Please note that the rule for field update differs slightly from Java: Evaluating the term $x.a\mathsf{<-}y$ has the side effect of updating the field a of object x with the value y. However, the whole term itself does not evaluate to the right-hand value y as in Java, but rather to the left-hand value x.

Of course, we can define the Java style update as $\mathsf{let}\ u = (x.a\mathsf{<-}y)\ \mathsf{in}\ y$. The reason for choosing the other primitive is that the proof of type soundness becomes slightly easier and also that it is more natural from a resource-oriented point of view since the updated object (x) should still be available whereas the update value (y) should be considered "consumed" unless explicitely duplicated, in our case by invocation of the sharing rule FJ:SHARE, see Section 3.0.1 below. Note that rule FJ:SHARE is needed to type the encoding of Java style update above.

2.1 Semantics of FJEU

We assume a set Loc of *locations*, ranged over by the letter ℓ, possibly decorated. A *stack value* v is either a location $\ell \in \mathsf{Loc}$ or a special value $\mathsf{0} \notin \mathsf{Loc}$. A *stack* or environment η is a partial mapping from identifiers to stack values.

A *heap value* w is a record consisting of a class name and list of labelled stack values, written $(C, a_1 : v_1, \ldots, a_k : v_k)$, where C is a proper FJEU class contained in $\mathscr{C}$. A *heap* σ is a partial mapping from locations to heap values. In addition to our convential notations for partial mappings, we write $\sigma[\ell.a \mapsto v]$ for the heap $\sigma\big[\ell \mapsto (C, a_1 : v_1, \ldots, a_i : v, \ldots, a_k : v_k)\big]$, provided that $a_i = a$ and $\sigma_\ell = (C, a_1 : v_1, \ldots, a_k : v_k)$.

An *alias* $v.\vec{p}$ is a pair of a stack value v and a (possibly empty) access path $\vec{p}$. For a given heap σ we recursively define $[\![v.\vec{p}]\!]_\sigma$ by

$$[\![v.\vec{p}]\!]_\sigma = \begin{cases} v & \text{if } \vec{p} = \varepsilon \\ v_i & \text{if } \vec{p} = \vec{q}.a_i \wedge \sigma_{[\![v.\vec{q}]\!]_\sigma} = (C, a_1 : v_1, \ldots, a_k : v_k) \\ \mathsf{0} & \text{otherwise} \end{cases}$$

We say that $v.\vec{p}$ is an alias for the location ℓ within σ if $[\![v.\vec{p}]\!]_\sigma = \ell$ holds, i.e. the alias represents a valid path leading to a proper location.

In order to relate an FJEU typing to a heap configuration, we define:

$$\llbracket v.\vec{p} \rrbracket^{\mathrm{dyn}}_{\sigma} = \; C \;\text{ iff }\; \sigma(\llbracket v.\vec{p} \rrbracket_{\sigma}) = (C, a_1 : v_1, \ldots, a_k : v_k)$$

$$\llbracket (v:C).\vec{p} \rrbracket^{\mathrm{stat}}_{\sigma} = \begin{cases} C & \text{if } \vec{p} = \varepsilon \\ D.a & \text{if } \vec{p} = \vec{q}.a \;\wedge\; \llbracket v.\vec{q} \rrbracket^{\mathrm{dyn}}_{\sigma} = D \end{cases}$$

Note that $\llbracket v.\vec{q} \rrbracket^{\mathrm{dyn}}_{\sigma}$ and $\llbracket (v:C).\vec{q}.a \rrbracket^{\mathrm{stat}}_{\sigma}$ are only defined if $\llbracket v.\vec{q} \rrbracket_{\sigma}$ is defined and within the domain of σ.

We base our statical resource analysis on the operational semantics shown in Fig. 2. The judgement $\eta, \sigma \vdash^{m}_{m'} e \rightsquigarrow v, \tau$ shall mean that the expression e evaluates successfully to the value v, beginning with stack η, heap σ and ending with heap τ. The non-negative natural numbers m and m' keep track of the number of unused heap units before and after evaluating e respectively.

A *heap unit* shall represent the smallest portion of memory that can be allocated individually.

Note that rule $\vdash$New is only applicable if there are enough unused heap units available for the allocation of a new object, otherwise the evaluation of a `new` expression gets stuck. Heap units are unconditionally made available again by the `free` expression, which we will comment upon shortly. Our analysis will identify an upper bound on m and a lower bound on m' for a given expression and program. We include the rule $\vdash$Waste which allows us to abandon unused memory units in order to facilitate some proofs.

One possible way to interpret the presented operational semantics is the assumption of a freelist-based memory model. The freelist contains initially a certain number of locations corresponding to unused heap units. Upon memory allocation the required number of locations is taken from the freelist; upon deallocation a corresponding number of locations of freed heap units are returned to the freelist. Issues of alignment and fragmentation will be ignored here.

In the underlying informal physical model it is perfectly possible that such a freelist occasionally contains locations that are still reachable from the current environment by dangling pointers. Therefore a call to `new` may return a (stale) pointer that is therefore aliased from the beginning. Such an accidental runtime alias may even violate type safety, since a standard type system abstracts away from physical memory addresses. However, existing work may already be used to exclude such faulty behaviour (amongst other benificial properties), e.g. the alias types by Walker and Morrisett [WM01], or the bunched implication logic as practised by Ishtiaq and O'Hearn [IO01].

Therefore, we choose the modular approach of switching essentially to a storeless semantics [RBR+05, BIL03, Deu94, Jon81], hence allowing us to assume that a call to `new` will always return a fresh pointer and leave it to the reader to choose his or her preferred method to ensure this.

We admit a special heap value (invalid) to mark freed heap values. Hence locations are never removed from the domain of the heap and all allocated locations are thus trivially fresh. Locations pointing to disposed heap values are treated like dangling pointers. Hence pointers to invalid locations are harmless

$$\eta, \sigma \vdash^{m}_{m} x \rightsquigarrow \eta_x, \sigma \qquad (\vdash\textsc{Var})$$

$$\eta, \sigma \vdash^{m}_{m} \mathsf{null} \rightsquigarrow 0, \sigma \qquad (\vdash\textsc{Null})$$

$$\frac{\ell \notin \mathrm{dom}(\sigma) \qquad \tau = \sigma[\ell \mapsto (C, a_1 : 0, \ldots, a_k : 0)]}{\eta, \sigma \vdash^{m+1}_{m} \mathtt{new}\ C \rightsquigarrow \ell, \tau} \qquad (\vdash\textsc{New})$$

$$\frac{\eta_x = \ell \qquad \sigma_\ell = (C, a_1 : v_1, \ldots, a_k : v_k)}{\eta, \sigma \vdash^{m}_{m+1\circ} \mathtt{free}(x) \rightsquigarrow 0, \sigma[\ell \mapsto (\mathsf{invalid})]} \qquad (\vdash\textsc{Free})$$

$$\frac{\eta_x = \ell \qquad \sigma_\ell = (D, a_1 : v_1, \ldots, a_k : v_k) \qquad D <: C}{\eta, \sigma \vdash^{m}_{m} (C)x \rightsquigarrow \eta_x, \sigma} \qquad (\vdash\textsc{Cast I})$$

$$\frac{\eta_x = 0}{\eta, \sigma \vdash^{m}_{m} (C)x \rightsquigarrow \eta_x, \sigma} \qquad (\vdash\textsc{Cast II})$$

$$\frac{\eta_x = \ell \qquad \sigma_\ell = (C, a_1 : v_1, \ldots, a_k : v_k)}{\eta, \sigma \vdash^{m}_{m} x.a_i \rightsquigarrow v_i, \sigma} \qquad (\vdash\textsc{Access})$$

$$\frac{\begin{array}{c}\eta_x = \ell \qquad \sigma_\ell = (C_0, a_1 : v_1, \ldots, a_k : v_k) \\ a = a_i \qquad \tau = \sigma[\ell.a_i \mapsto \eta_y]\end{array}}{\eta, \sigma \vdash^{m}_{m} x.a\mathtt{<-}y \rightsquigarrow \ell, \tau} \qquad (\vdash\textsc{Update})$$

$$\frac{\begin{array}{c}\eta_x = \ell \qquad \sigma_\ell = (C, a_1 : v_1, \ldots, a_k : v_k) \qquad \mathsf{M}_{\mathsf{body}}(C, m) = e_0 \\ [\mathtt{this} \mapsto \ell, x_1 \mapsto \eta_{y_1}, \ldots, x_j \mapsto \eta_{y_j}], \sigma \vdash^{m}_{m'} e_0 \rightsquigarrow v, \tau\end{array}}{\eta, \sigma \vdash^{m}_{m'} x.m(y_1, \ldots, y_j) \rightsquigarrow v, \tau} \qquad (\vdash\textsc{Invocation})$$

$$\frac{\begin{array}{c}\eta_x = \ell \qquad \sigma(\ell) = (D, a_1 : v_1, \ldots, a_k : v_k) \\ D <: E \qquad \eta, \sigma \vdash^{m}_{m'} e_1 \rightsquigarrow v, \tau\end{array}}{\eta, \sigma \vdash^{m}_{m'} \mathtt{if}\ x\ \mathtt{instanceof}\ E\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \rightsquigarrow v, \tau} \qquad (\vdash\textsc{Conditional I})$$

$$\frac{\begin{array}{c}\eta_x = \ell \qquad \sigma(\ell) = (D, a_1 : v_1, \ldots, a_k : v_k) \\ D \not<: E \qquad \eta, \sigma \vdash^{m}_{m'} e_2 \rightsquigarrow v, \tau\end{array}}{\eta, \sigma \vdash^{m}_{m'} \mathtt{if}\ x\ \mathtt{instanceof}\ E\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \rightsquigarrow v, \tau} \qquad (\vdash\textsc{Conditional II})$$

$$\frac{\eta_x = 0 \qquad \eta, \sigma \vdash^{m}_{m'} e_2 \rightsquigarrow v, \tau}{\eta, \sigma \vdash^{m}_{m'} \mathtt{if}\ x\ \mathtt{instanceof}\ E\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \rightsquigarrow v, \tau} \qquad (\vdash\textsc{Conditional III})$$

$$\frac{\eta, \sigma \vdash^{m}_{m'} e_1 \rightsquigarrow v_1, \rho \qquad \eta[x \mapsto v_1], \rho \vdash^{m'}_{m''} e_2 \rightsquigarrow v_2, \tau}{\eta, \sigma \vdash^{m}_{m''} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \rightsquigarrow v_2, \tau} \qquad (\vdash\textsc{Let})$$

$$\frac{\eta, \sigma \vdash^{m}_{m'+d} e \rightsquigarrow v, \tau}{\eta, \sigma \vdash^{m}_{m'} e \rightsquigarrow v, \tau} \qquad (\vdash\textsc{Waste})$$

Figure 2: Operational Semantics of FJEU

as long as they are never dereferenced, so cyclic data structures can be deallocated. It is convenient to redefine the shorthand $\ell \in \sigma$ by "$\ell \in \mathrm{dom}(\sigma)$ and $\sigma(\ell) \neq (\mathsf{invalid})$". We explicitely write $\ell \in \mathrm{dom}(\sigma)$ otherwise.

One may note that one way of directly modelling these altered semantics lies in the use of indirect pointers (symbolic handles) used by earlier implementations of the Sun JVM for the compacting garbage collector.

Definition 2.1 (Sound Heap) *We write $\sigma \vDash \eta : \Gamma$ to say that a memory configuration consisting of heap σ and stack η is sound and satisfies an FJEU typing context Γ, if*

$$\mathrm{ran}(\eta) \subseteq \mathrm{dom}(\sigma) \cup \{\mathsf{0}\} \tag{2.1}$$

$$\begin{aligned} \forall \ell \in \mathrm{dom}(\sigma)\,.\,\sigma(\ell) = (C, a_1 : v_1, \ldots, a_k : v_k) \implies \\ \forall i \in \{1, \ldots, k\}\,.\,v_i \in \mathrm{dom}(\sigma) \cup \{\mathsf{0}\} \qquad (2.2) \\ \wedge\, \mathsf{A}(C) = \{a_1, \ldots, a_k\} \end{aligned}$$

$$\forall \eta_x.\vec{p} \in \sigma\,.\,[\![\eta_x.\vec{p}]\!]^{\mathrm{dyn}}_{\sigma} <: [\![(\eta_x : \Gamma_x).\vec{p}]\!]^{\mathrm{stat}}_{\sigma} \tag{2.3}$$

In other words, each object record contains exactly the attribute labels as specified in the classtable and furthermore, for each alias this actual class is a subtype of the statical class as derivable from the typing context. We write $\sigma \vDash v : C$ as abbreviation for $\sigma \vDash [x \mapsto v] : x : C$.

Klein and Nipkow [KN04] and other authors define the equivalent of our relation $\sigma \vDash v : C$ in a slightly different manner, namely they require (in our notation) that if $\sigma(v) = (D, \ldots)$ then $D <: C$ and whenever $\sigma(\ell) = (C, a_1 : v_1, \ldots, a_k : v_k)$ and $\sigma(v_i) = (E, \ldots)$ then $E <: C.a_i$.

Our notion is equivalent to Nipkow's when restricted to the reachable fragment of σ. We prefer our path-oriented version for two reasons: first, it allows us to give a precise meaning to static types of heap locations which, as is easily seen, depend on the access path leading up to the location. Furthermore, this definition generalises more smoothly to the annotated version RAJA of FJEU to be defined in Section 3 below. Indeed, in RAJA the static type will depend on the entire access path and not only on its last hop as is the case in FJEU. It is interesting to note that some of our constructs involving access paths are actually quite similar to constructs used in [BIL03] which deal with a proper storeless semantics.

Definition 2.2

$$[\sigma, v, \vec{p} \between \ell, a] \overset{\mathrm{def}}{\iff} \exists \vec{q} \prec \vec{p}.\ [\![v.\vec{q}]\!]_{\sigma} = \ell \wedge \vec{q}.a \preceq \vec{p}$$

Lemma 2.3 *If $\tau = \sigma[\ell.a \mapsto w]$ then*

$$\begin{aligned} \neg[\sigma, v, \vec{p} \between \ell, a] &\implies [\![v.\vec{p}]\!]_{\sigma} = [\![v.\vec{p}]\!]_{\tau} \\ [\sigma, v, \vec{p} \between \ell, a] &\iff [\tau, v, \vec{p} \between \ell, a] \end{aligned}$$

Lemma 2.4

$$\eta, \sigma \mathrel{\vdash^{m}_{m'}}{}^{\circ}\ e \rightsquigarrow v, \tau \quad \textit{implies} \quad \tilde{\eta}, \tilde{\sigma} \mathrel{\vdash^{m}_{m'}} e \rightsquigarrow \tilde{v}, \tilde{\tau}$$

Lemma 2.5 *If* $\eta, \sigma \vdash^{m}_{m'} e \rightsquigarrow v, \tau$ *and* $\mathrm{dom}(\eta) \cap \mathrm{dom}(\eta') = \emptyset$ *then also* $\eta\eta', \sigma \vdash^{m}_{m'} e \rightsquigarrow v, \tau$.

Definition 2.6 (Unannotated Semantics)

$$\eta, \sigma \vdash e \rightsquigarrow v, \tau \stackrel{\text{def}}{\iff} \exists m, m' \,.\, \eta, \sigma \vdash^{m}_{m'} e \rightsquigarrow v, \tau$$

It is easy to see that the relation $\eta, \sigma \vdash e \rightsquigarrow v, \tau$ can also be inductively defined by rules shown in Fig. 2 with the numerical annotations removed.

Lemma 2.7

$$\forall d \in \mathbb{N} \,.\, \eta, \sigma \vdash^{m}_{m'} e \rightsquigarrow v, \tau \implies \eta, \sigma \vdash^{d+m}_{d+m'} e \rightsquigarrow v, \tau$$

Lemma 2.8 *Let* $\eta, \sigma \vdash^{m}_{m'} e \rightsquigarrow v, \tau$. *Then, if* $n \geq m$ *also* $\eta, \sigma \vdash^{n}_{m'} e \rightsquigarrow v, \tau$.

Proof. We have $n = m + x$ for some x. By Lemma 2.7 we have $\eta, \sigma \vdash^{n}_{m'+x} e \rightsquigarrow v, \tau$, and by rule ($\vdash$Waste) we get $\eta, \sigma \vdash^{n}_{m'} e \rightsquigarrow v, \tau$. □

Definition 2.9 *We extend the operational semantics to rational annotations* $t, t' \in \mathbb{Q}^{+}$ *in the following way:*

$$\begin{aligned} \eta, \sigma \vdash^{t}_{t'} e \rightsquigarrow v, \tau \stackrel{\text{def}}{\iff} & \\ & \forall u \in \mathbb{Q}^{+}, m \in \mathbb{N} \,.\, m \geq t + u \implies \\ & \qquad \exists m' \in \mathbb{N} \,.\, m' \geq t' + u \land \eta, \sigma \vdash^{m}_{m'} e \rightsquigarrow v, \tau \end{aligned}$$

Lemma 2.10 *Each rule in Fig.2 holds as an implication for rational annotations as defined in Def.2.9 as well, e.g.* $\eta, \sigma \vdash^{t}_{t'} e_1 \rightsquigarrow v_1, \rho$ *and* $\eta[x \mapsto v_1], \rho \vdash^{t'}_{t''} e_2 \rightsquigarrow v_2, \tau$ *imply* $\eta, \sigma \vdash^{t}_{t''} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \rightsquigarrow v_2, \tau$.

Proof. We will only exemplarily prove the claim for rule $\vdash$Let here: Fix m,u arbitrarily so that $t \geq m + u$ is satisfied. Hence by assumption there exists a m' satisfying $t' \geq m' + u$ and $\eta, \sigma \vdash^{m}_{m'} e_1 \rightsquigarrow v_1, \rho$. Furthermore by the second assumption, there exists a m'' so that $t'' \geq m'' + u$ and $\eta[x \mapsto v_1], \rho \vdash^{m'}_{m''} e_2 \rightsquigarrow v_2, \tau$ Hence we deduce by $\vdash$Let that $\eta, \sigma \vdash^{m}_{m''} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \rightsquigarrow v_2, \tau$ holds. Since m and u were arbitrarily chosen, this completes the claim for t and t''.

Note that the proof for most other rules relies on Lem.2.7 and application of rule $\vdash$Waste. □

Remark 2.11 *One may also characterize the rational annotations for the operational semantics as defined in Def.2.9 by*

$$\eta, \sigma \vdash_{t'}^{t} e \rightsquigarrow v, \tau \overset{\text{def}}{\iff} \eta, \sigma \vdash_{\lceil t' \rceil + \epsilon}^{\lceil t \rceil} e \rightsquigarrow v, \tau$$

where $\epsilon := \begin{cases} 1 & \text{if } \lceil t \rceil - t > \lceil t' \rceil - t' \\ 0 & \text{otherwise} \end{cases}$

Lemma 2.12 *For all* $t, t', u, u' \in \mathbb{Q}^+$ *holds:*

$$\eta, \sigma \vdash_{t'}^{t} e \rightsquigarrow v, \tau \wedge u \geq t \wedge u' + t \leq t' + u \implies \eta, \sigma \vdash_{u'}^{u} e \rightsquigarrow v, \tau$$

Lemma 2.13

$$\eta, \sigma \vdash_{t'}^{t} e \rightsquigarrow v, \tau \implies \eta, \sigma \vdash_{\lceil t' \rceil}^{\lceil t \rceil} e \rightsquigarrow v, \tau$$

3 Extending FJEU to RAJA

We now extend FJEU to an annotated version, RAJA, (Resource Aware JAva) as announced in the Introduction. A *RAJA program* is an annotation of an FJEU class table $\mathscr{C}$ in the form of a sextuple
$\mathscr{R} = (\mathscr{C}, \mathscr{V}, \Diamond(\cdot), \mathsf{A}^{\text{get}}(\cdot,\cdot), \mathsf{A}^{\text{set}}(\cdot,\cdot), \mathsf{M}(\cdot,\cdot))$ specified as follows:

1. $\mathscr{V}$ is a possibly infinite set of views. For each class $C \in \text{dom}(\mathscr{C})$ and for each view $r \in \mathscr{V}$ the pair C^r is called a RAJA class. We refer to RAJA classes also as *(refined) types*.

2. $\Diamond(\cdot)$ assigns to each RAJA class C^r a number $\Diamond(C^r) \in \mathbb{D}$, where $\mathbb{D} = \mathbb{Q}^+ \cup \infty$. This number will be used to define the potential of a heap configuration under a given static RAJA typing.

3. $\mathsf{A}^{\text{get}}(\cdot,\cdot)$ and $\mathsf{A}^{\text{set}}(\cdot,\cdot)$ assign to each RAJA class C^r and attribute $a \in \mathsf{A}(C)$ two views $q = \mathsf{A}^{\text{get}}(C^r, a)$ and $s = \mathsf{A}^{\text{set}}(C^r, a)$. The intention is that if $D = C.a$ is the FJEU type of attribute a in C then the RAJA type D^q will be the type of an access to a, whereas the (intendedly stronger) type D^s must be used when updating a. The stronger typing is needed since an update will possibly affect several aliases.

4. $\mathsf{M}(\cdot,\cdot)$ assigns to each RAJA class C^r and method $m \in \mathsf{M}(C)$ having method type $E_1, \ldots, E_j \to E_0$ a j-ary polymorphic RAJA method type $\mathsf{M}(C^r, m)$.

   A *j-ary polymorphic RAJA method type* is a (possibly empty or infinite) set of j-ary monomorphic RAJA method types. A *j-ary monomorphic*

RAJA method type consists of $j+1$ views and two numbers $p, q \in \mathbb{D}$, written $r_1, \ldots, r_j \xrightarrow{p/q} r_0$.

The idea is that if m (of FJEU-type $E_1, \ldots, E_j \to E_0$) has (among others) the monomorphic RAJA method type $r_1, \ldots, r_j \xrightarrow{p/q} r_0$ then it may be called with arguments $v_1 : E_1^{r_1}, \ldots, v_j : E_j^{r_j}$, whose associated potential will be consumed, as well as an additional potential of p. Upon successful completion the return value will be of type $E_0^{r_0}$ hence carry an according potential. In addition to this a potential of another q units will be returned.

We sometimes write $E_1^{r_1}, \ldots, E_j^{r_j} \xrightarrow{p/q} E_0^{r_0}$ to denote an FJEU method type combined with a corresponding monomorphic RAJA method type.

We will now define when such a RAJA-annotation of an FJEU class table is indeed valid; in particular this will require that each method body is typable with each of the monomorphic RAJA method types given in the annotation.

Definition 3.1 (Subtyping of RAJA types) *We define a preorder $<:$ on RAJA types C^r, D^s where $C <: D$ in $\mathscr{C}$ and $r, s \in \mathscr{V}$, as the largest relation $(C^r <: D^s)$ such that:*

$$C^r <: D^s \iff \textit{for each } \; E, F <: D \; \textit{ with } \; E <: F :$$

$$\Diamond\!(E^r) \geq \Diamond\!(F^s) \tag{3.1}$$

$$\forall a \in \mathsf{A}(F) \,.\, (F.a)^{\mathsf{A}^{\mathsf{get}}(E^r, a)} <: (F.a)^{\mathsf{A}^{\mathsf{get}}(F^s, a)} \tag{3.2}$$

$$\forall a \in \mathsf{A}(F) \,.\, (F.a)^{\mathsf{A}^{\mathsf{set}}(F^s, a)} <: (F.a)^{\mathsf{A}^{\mathsf{set}}(E^r, a)} \tag{3.3}$$

$$\forall m \in \mathsf{M}(F) \,.\, \forall \beta \in \mathsf{M}(F^s, m) \,.\, \exists \alpha \in \mathsf{M}(E^r, m) \,.\, (F.m)^{\alpha} <: (F.m)^{\beta} \tag{3.4}$$

where we extend $<:$ to monomorphic RAJA method types as follows:

Definition 3.2 (Subtyping of RAJA methods) *If $D.m = E_1, \ldots, E_j \to E_0$, $\alpha = r_1, \ldots, r_j \xrightarrow{p/q} r_0$ and $\beta = s_1, \ldots, s_j \xrightarrow{t/u} s_0$ then $(D.m)^{\alpha} <: (D.m)^{\beta}$ is defined as $p \leq t$ and $q \geq u$ and $E_0^{r_0} <: E_0^{s_0}$ and $E_i^{s_i} <: E_i^{r_i}$ for $i = 1, \ldots, j$.*

If Γ and Θ are contexts we write $\Gamma <: \Theta$ if $\forall x \in \Theta \,.\, \Gamma_x <: \Theta_x$.

The following definition is important for correctly using variables more than once. If a variable is to be used more than once, e.g., as an argument to a method, then the different occurrences must be given different types which are chosen such that the individual potentials assigned to each occurrence add up to the total potential available.

Definition 3.3 (Sharing Relation) *We define the* sharing relation *between a single RAJA type C^r and a multiset of RAJA types $D^{s_1}, \ldots, D^{s_n}$ written*

$\curlyvee(C^r\,|D^{s_1},\ldots,D^{s_n})$ *as the largest relation* $\curlyvee$*, such that if* $\curlyvee(C^r\,|D^{s_1},\ldots,D^{s_n})$ *then for all* $E, F <: D$ *with* $E <: F$*:*

$$\Diamond\!(E^r) \geq \sum_i \Diamond\!(F^{s_i}) \tag{3.5}$$

$$\forall i\,.\,E^r <: F^{s_i} \tag{3.6}$$

$$\forall a \in \mathrm{dom}(\mathsf{A}(F))\,.\,\curlyvee\Big((F.a)^{\mathsf{A}^{\mathsf{get}}\curlyvee(E^r,a)}\,\Big|\,(F.a)^{\mathsf{A}^{\mathsf{get}}\curlyvee(F^{s_1},a)},\ldots,(F.a)^{\mathsf{A}^{\mathsf{get}}\curlyvee(F^{s_n},a)}\Big) \tag{3.7}$$

Lemma 3.4

$$\curlyvee(C^r\,|\emptyset) \tag{3.8}$$

$$\curlyvee(C^r\,|\{C^r\}) \tag{3.9}$$

$$\curlyvee(C^r\,|\mathcal{D}) \iff \forall \mathcal{E} \subset \mathcal{D}\,.\,\curlyvee(C^r\,|\mathcal{E}) \tag{3.10}$$

$$\curlyvee(C^r\,|\mathcal{D}\cup\{C^s\}) \wedge \curlyvee(C^s\,|\mathcal{E}) \implies \curlyvee(C^r\,|\mathcal{D}\cup\mathcal{E}) \tag{3.11}$$

$$C^{r'} <: C^r \wedge C^{s'} <: C^s \wedge \curlyvee\Big(C^r\,\Big|\,\mathcal{D}\cup\{C^{s'}\}\Big) \implies \curlyvee\Big(C^{r'}\,|\,\mathcal{D}\cup\{C^s\}\Big) \tag{3.12}$$

$$C^r <: C^s \iff \curlyvee(C^r\,|C^s) \tag{3.13}$$

3.0.1 Typing RAJA

We now give the formal definition of the RAJA-typing judgement. See Figure 3. The type system allows us to derive assertions of the form $\Gamma \vdash^{n}_{n'} e : C^r$ where e is an expression or program phrase, C is a Java class, r is a view (so C^r is a refined type). Γ maps variables occurring in e to refined types; we often write Γ_x instead of $\Gamma(x)$. Finally n, n' are nonnegative numbers. The meaning of such a judgement is as follows. If e terminates successfully in some environment η and heap σ with unbounded memory resources available then it will also terminate successfully with a bounded freelist of size at least n plus the potential ascribed to η, σ with respect to the typings in Γ.

Furthermore, the freelist size upon termination will be at least n' plus the potential of the result with respect to the view r.

The typing rules are standard. The most interesting ones are $(\Diamond Share)$ and $(\Diamond Waste)$. First we notice that they are not syntax directed, thus, they need to be eliminated when it comes to implement the system in the next section. The purpose of the $(\Diamond Share)$ rule is to ensure that a variable can be used twice without duplication of potential.

The judgement $\vdash m : \alpha\ \mathsf{ok}$ where m is the name of a method in a RAJA class C^r, and α is a RAJA type for this method means that α is a valid RAJA type for m if the method body of m can be typed with the arguments, return type and effects as specified in α. There is another interesting point here: in order to use the potential of the variable this, we put it in the context with a modified type. For example, if we have $\mathsf{this} : C^r$ and $\Diamond\!(C^r) = 1$, if we can find two views such that $\curlyvee(C^r\,|C^q, C^s)$ with $\Diamond\!(C^q) = 0$ and $\Diamond\!(C^s) = 1$, then

RAJA Typing $\boxed{\Gamma \vdash^{n}_{n'} e : C^r}$

$$\frac{}{\varnothing \vdash^{\Diamond(C^r)+1}_{0} \texttt{new}\ C : C^r}\ (\Diamond New) \qquad \frac{}{x{:}C^r \vdash^{0}_{\Diamond(C^r)+1} \texttt{free}(x) : E^r}\ (\Diamond Free)$$

$$\frac{C <: E}{x{:}E^r \vdash^{0}_{0} (C)x : C^r}\ (\Diamond Cast) \qquad \frac{}{\varnothing \vdash^{0}_{0} \mathsf{null} : C^r}\ (\Diamond Null) \qquad \frac{}{x{:}C^r \vdash^{0}_{0} x : C^r}\ (\Diamond Var)$$

$$\frac{s = \mathsf{A}^{\mathrm{get}}(C^r, a) \qquad D = C.a}{x{:}C^r \vdash^{0}_{0} x.a : D^s}\ (\Diamond Access) \qquad \frac{\mathsf{A}^{\mathrm{set}}(C^r, a) = s \qquad C.a = D}{x{:}C^r, y{:}D^s \vdash^{0}_{0} x.a\texttt{<-}y : C^r}\ (\Diamond Update)$$

$$\frac{\Gamma_1 \vdash^{n}_{n'} e_1 : D^s \qquad \Gamma_2, x{:}D^s \vdash^{n'}_{n''} e_2 : C^r}{\Gamma_1, \Gamma_2 \vdash^{n}_{n''} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : C^r}\ (\Diamond Let)$$

$$\frac{\left(E_1^{q_1}, \ldots, E_j^{q_j} \xrightarrow{n/n'} E_0^{q_0}\right) \in \mathsf{M}(C^r, m)}{x{:}C^r, y_1{:}E_1^{q_1}, \ldots, y_j{:}E_j^{q_j} \vdash^{n}_{n'} x.m(y_1, \ldots, y_j) : E_0^{q_0}}\ (\Diamond Invocation)$$

$$\frac{x \in \Gamma \qquad \Gamma \vdash^{n}_{n'} e_1 : C^r \qquad \Gamma \vdash^{n}_{n'} e_2 : C^r}{\Gamma \vdash^{n}_{n'} \texttt{if}\ x\ \texttt{instanceof}\ E\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : C^r}\ (\Diamond Conditional)$$

$$\frac{\curlyvee(D^s \,|D^{q_1}, \ldots, D^{q_n}) \qquad \Gamma, y_1{:}D^{q_1}, \ldots, y_n{:}D^{q_n} \vdash^{n}_{n'} e : C^r}{\Gamma, x{:}D^s \vdash^{n}_{n'} e[x/y_1, \ldots, x/y_n] : C^r}\ (\Diamond Share)$$

$$\frac{n \geq u \quad n + u' \geq n' + u \quad \Theta \vdash^{u}_{u'} e : D^s \quad \Gamma <: \Theta \quad D^s <: C^r}{\Gamma \vdash^{n}_{n'} e : C^r}\ (\Diamond Waste)$$

RAJA Method Typing $\boxed{\vdash m : \alpha\ \mathsf{ok}}$

$$\frac{m \in \mathsf{M}(C) \qquad \alpha = E_1^{r_1}, \ldots, E_j^{r_j} \xrightarrow{n/n'} E_0^{r_0} \in \mathsf{M}(C^r, m) \qquad \curlyvee(C^r \,|C^q, C^s)}{\texttt{this}{:}C^q, x_1{:}E_1^{r_1}, \ldots, x_j{:}E_j^{r_j} \vdash^{n + \Diamond(C^s)}_{n'} \mathsf{M}_{\mathsf{body}}(C, m) : E_0^{r_0}}\ (\Diamond MBody)$$

Figure 3: Typing RAJA

we put this in the context with type this : C^q and we can use the potential 1 of C^s when typing the method body. The rule gives no information about how to find the views q and s, but in our implementation these views can be found automatically.

Definition 3.5 (Well-typed RAJA-program) *A RAJA-program* $\mathscr{R} = (\mathscr{C}, \mathscr{V}, \Diamond(\cdot), \mathsf{A}^{\mathrm{get}}(\cdot\,,\cdot)\,, \mathsf{A}^{\mathrm{set}}(\cdot\,,\cdot)\,, \mathsf{M}(\cdot,\cdot))$ *is* well-typed *if for all* $C \in \mathscr{C}$ *and* $r \in \mathscr{V}$ *the following conditions are satisfied:*

1. $\mathsf{S}(C) = D \Rightarrow C^r <: D^r$
2. $\forall a \in \mathsf{A}(C)\ .\ (C.a)^{\mathsf{A}^{\mathrm{set}}(C^r,a)} <: (C.a)^{\mathsf{A}^{\mathrm{get}}(C^r,a)}$
3. $\forall m \in \mathsf{M}(C)\ .\forall \alpha \in \mathsf{M}(C^r,m)\ .\ \vdash m : \alpha\ \mathsf{ok}$

4 Heap soundness and Potential

Throughout this section we assume given a well-typed RAJA program $\mathscr{R}$.

Recall that in FJEU we assign both a static and a dynamic type to each accessible location in a heap specified by a variable and an access path.

We emphasise that RAJA uses the same runtime model as FJEU, in particular no RAJA-typing information will be attached to objects in the heap. Therefore, we will be able to define a static RAJA type in analogy with the static FJEU type, but no dynamic RAJA type will be available. Of course, as in the case of FJEU the static RAJA type of a value and an access path will depend on the heap. In RAJA it will depend on the entire access path and not just on its last component.

Definition 4.1 (Static RAJA Type) *For a given heap* σ*, a stack value* v*, a (possibly empty) access path* $\vec{p}$ *and a RAJA type* C^r *such that* $\sigma \vDash v : C$ *we recursively define* $\llbracket (v\!:\!r).\vec{p} \rrbracket_\sigma^{\mathrm{stat}}$ *by*

$$\llbracket (v\!:\!r).\vec{p} \rrbracket_\sigma^{\mathrm{stat}} = \begin{cases} D^r & \text{if } \vec{p} = \varepsilon \\ D^s & \text{if } \vec{p} = \vec{q}.a \text{ and } \llbracket (v\!:\!r).\vec{q} \rrbracket_\sigma^{\mathrm{stat}} = E^t \text{ and } \mathsf{A}^{\mathrm{get}}\big(E^t, a\big) = s \end{cases}$$

where $D = \lfloor\!\lfloor v.\vec{p} \rfloor\!\rfloor_\sigma^{\mathrm{dyn}}$. Note that $\llbracket (v\!:\!r).\vec{q}.a \rrbracket_\sigma^{\mathrm{stat}}$ is defined if $v.\vec{q} \in \sigma$.

Furthermore, assume $\sigma \vDash \eta : |\Gamma|$ holds for a RAJA context Γ and an environment η. We then define the multiset of RAJA types associated with all aliases of a location $\ell \in \sigma$ by

$$\mathscr{R}_{\sigma,\eta,\Gamma}(\ell) = \{\, \llbracket (\eta_x\!:\!\langle\!\langle \Gamma_x \rangle\!\rangle).\vec{p} \rrbracket_\sigma^{\mathrm{stat}} \mid x \in \Gamma,\ [\![\eta_x.\vec{p}]\!]_\sigma = \ell,\ \}$$

Lemma 4.2 *Let* $r \in \mathscr{V}$ *and assume* $\sigma \vDash v : C$*. The static RAJA type* $\llbracket (v : r).\vec{p} \rrbracket_\sigma^{\mathrm{stat}}$ *is defined whenever the static FJEU type* $\lfloor\!\lfloor (v\!:\!C).\vec{p} \rfloor\!\rfloor_\sigma^{\mathrm{stat}}$ *is defined.*

Lemma 4.3 *If* $\tau = \sigma[\ell.a \mapsto v]$ *and* $\neg[\sigma, v, \vec{p} \between \ell, a]$ *then* $\llbracket (v\!:\!r).\vec{p} \rrbracket_\sigma^{\mathrm{stat}} = \llbracket (v : r).\vec{p} \rrbracket_\tau^{\mathrm{stat}}$.

Definition 4.4 (Sound Heap) *We say that a memory configuration consisting of heap σ and stack η satisfies a RAJA-context Γ, written $\sigma \vDash \eta : \Gamma$, if*

$$\sigma \vDash \eta : |\Gamma| \tag{4.1}$$

$$\forall \ell \in \sigma \,.\, \exists C \in \mathscr{C} \text{ and } r \in \mathscr{V} \,.\, \curlyvee(C^r \,|\mathscr{R}_{\sigma,\eta,\Gamma}(\ell)\,) \tag{4.2}$$

We define $\sigma \vDash v : C^r$ as $\sigma \vDash [x{\mapsto}v] : x{:}C^r$.

One may wonder how the mere existence of a "common" RAJA type C^r for all the aliases of some location could be of any use; the answer lies in the following lemma:

Lemma 4.5 *If $\curlyvee(C^r\,|C^{s_1}, \ldots, C^{s_n}\,)$ then for each $D <: C$ and $a \in \mathsf{A}(C)$ and s_i holds $\curlyvee\Big((C.a)^{\mathsf{A}^{\mathrm{set}}(D^{s_i}a)} \,\Big|\, (C.a)^{\mathsf{A}^{\mathrm{get}}(D^{s_1}a)}, \ldots, (C.a)^{\mathsf{A}^{\mathrm{get}}(D^{s_n}a)}\Big)$.*

Definition 4.6 (Potential) *If $\sigma \vDash v : C^s$ we define $\Phi_\sigma(v : r) \in \mathbb{D}$ by*

$$\Phi_\sigma(v : r) = \sum_{\vec{p}} \phi_\sigma\big((v{:}r).\vec{p}\big)$$

where $\phi_\sigma\big((v{:}r).\vec{p}\big) = \begin{cases} \Diamond(D^s) & \text{if } [\![(v{:}r).\vec{p}]\!]^{\mathrm{stat}}_\sigma = D^s \\ 0 & \text{otherwise} \end{cases}$

The potential is extended to environments and contexts as follows:

$$\Phi_\sigma(\eta : \Gamma) = \sum_{x \in \Gamma} \Phi_\sigma(\eta_x : \langle\!\langle \Gamma_x \rangle\!\rangle)$$

where we assume $\sigma \vDash \eta : |\Gamma|$.

This sum is possibly infinite, e.g. in the presence of circular data structures or an infinite heap. Furthermore each alias of a location makes a different contribution to the sum, whose value depends on the static RAJA type of that alias. We remark that in the absence of subtyping and inheritance the potential can be defined more simply as a sum over access paths, which does not depend on the dynamic type.

The potential interacts with subtyping and sharing as might be expected:

Lemma 4.7 *If $\sigma \vDash v : C^r$ and $C^r <: D^s$ and $[\![v.\vec{p}]\!]_\sigma \in \sigma$ and $\curlyvee(C^r\,|C^{s_1}, \ldots, C^{s_n}\,)$ then:*

$$[\![(v{:}r).\vec{p}]\!]^{\mathrm{stat}}_\sigma <: [\![(v{:}s).\vec{p}]\!]^{\mathrm{stat}}_\sigma \tag{4.3}$$

$$\curlyvee\Big([\![(v{:}r).\vec{p}]\!]^{\mathrm{stat}}_\sigma \,\Big|\, [\![(v{:}s_1).\vec{p}]\!]^{\mathrm{stat}}_\sigma, \ldots, [\![(v{:}s_n).\vec{p}]\!]^{\mathrm{stat}}_\sigma\Big) \tag{4.4}$$

$$\Phi_\sigma(v : r) \geq \Phi_\sigma(v : s) \tag{4.5}$$

$$\Phi_\sigma(v : r) \geq \sum_i \Phi_\sigma(v : s_i) \tag{4.6}$$

Proof. (4.3) and (4.4) by induction on $|\vec{p}|$. Then (4.5) follows from (4.3) and (4.6) follows from (4.4). □

Lemma 4.8 (Properties of a RAJA sound heap)

1. *If* $\sigma \vDash \eta : x{:}C^r, \Gamma$ *and* $C^r <: D^s$ *then also* $\sigma \vDash \eta : x{:}D^s, \Gamma$.
2. *Let* $\Gamma = x : C^r, \Delta$ *and* $\sigma \vDash \eta : \Gamma$ *and* $\eta' = \eta[y_1 \mapsto \eta_x, \ldots, y_n \mapsto \eta_x]$ *and* $\Gamma' = y_1{:}C^{s_1}, \ldots, y_n{:}C^{s_n}, \Delta$ *and* $\curlyvee(C^r\,|C^{s_1}, \ldots, C^{s_n})$ *then* $\sigma \vDash \eta' : \Gamma'$.

Proof.

1. We only show (4.2) $\forall \ell \in \sigma\,.\,\exists E \in \mathscr{C}$ and $q \in \mathscr{V}\,.\,\curlyvee(E^q\,|\mathscr{R}_{\sigma,\eta,\Gamma}(\ell))$, where

$$\mathscr{R}_{\sigma,\eta,\Gamma}(\ell) = \{\, \mathbb{\|}(\eta_x : \langle\!\langle \Gamma_x \rangle\!\rangle).\vec{p}\mathbb{\|}^{\text{stat}}_{\sigma} \mid x \in \Gamma,\ [\![\eta_x.\vec{p}]\!]_\sigma = \ell \}$$

   We show for every path $\vec{p}$ with $\ell = [\![\eta_x.\vec{p}]\!]_\sigma$: $\curlyvee(E^q\,|\mathscr{R}_{\sigma,\eta,x:C^r}(\ell))$ implies $\curlyvee(E^q\,|\mathscr{R}_{\sigma,\eta,x:D^s}(\ell))$. We have by Lemma 4.7, item (4.3), $\mathbb{\|}(\eta_x : r).\vec{p}\mathbb{\|}^{\text{stat}}_{\sigma} <: \mathbb{\|}(\eta_x : s).\vec{p}\mathbb{\|}^{\text{stat}}_{\sigma}$ thus, we are done by transitivity of sharing (Lemma 3.4, item (3.11)).

2. We need to show (4.1) and (4.2). We first notice that $\text{ran}(\eta') = \text{ran}(\eta)$, so (4.1) follows. Next we show (4.2). We have by assumption:
   $\forall \ell \in \sigma\,.\,\exists C \in \mathscr{C}$ and $r \in \mathscr{V}\,.\,\curlyvee(C^r\,|\mathscr{R}_{\sigma,\eta,\Delta}(\ell) \cup \mathscr{R}_{\sigma,\eta,x:C^r}(\ell))$ and
   $\curlyvee(\mathscr{R}_{\sigma,\eta,x:C^r}(\ell)\,|\mathscr{R}_{\sigma,\eta,y_1:C^{s_1}}(\ell), \ldots, \mathscr{R}_{\sigma,\eta,y_n:C^{s_n}}(\ell))$ by Lemma 4.7, item (4.4), thus, we are done by transitivity of sharing (Lemma 3.4, item (3.11)).

□

Theorem 4.9 (Main Result) *Fix a well-typed RAJA program* $\mathscr{R}$. *If*

$$\Gamma \overset{n}{\underset{n'}{\vdash}} e : C^r \tag{4.1}$$

$$\eta, \sigma \vdash^{\circ} e \rightsquigarrow v, \tau \tag{4.2}$$

$$\sigma \vDash \eta : (\Gamma, \Delta) \tag{4.3}$$

$$\eta, \sigma \overset{n + \Phi_\sigma(\eta : \Gamma) + \Phi_\sigma(\eta : \Delta)}{\underset{n' + \Phi_\tau(v : r) + \Phi_\tau(\eta : \Delta)}{\vdash}}{}^{\circ} e \rightsquigarrow v, \tau \tag{4.7}$$

$$\tau \vDash \eta[x_{res} \mapsto v] : (\Delta, x_{res} : C^r) \tag{4.8}$$

where x_{res} *is assumed to be an unused auxiliary variable, i.e.* $x_{res} \notin \Gamma, \Delta$. *Note that* (4.3) *implies* $\text{dom}(\Gamma) \cap \text{dom}(\Delta) = \emptyset$ *by definition of notation.*

Proof. The proof is by induction on the derivation of premise (4.2) and a subordinate induction on the derivation of premise (4.1). For establishing (4.8) we will address only part 4.2 of Def. 4.4 and elide the proof for 4.1 throughout, since this is trivial in all cases and merely states FJEU type soundness.

◊Share Without loss of generality, we assume $x, y_1, \ldots, y_n \notin \Delta$. Let $\Gamma = \Theta, x{:}\, D^s$, $e = e_0[x/y_1, \ldots, x/y_n]$ for some e_0 and $\Gamma' = \Theta, y_1 : D^{q_1}, \ldots, y_n : D^{q_n}$. From (4.1) and rule ◊Share, we have $\Gamma' \vdash^{n}_{n'} e_0 : C^r, \curlyvee(D^s \,|D^{q_1}, \ldots, D^{q_n})$. Furthermore let $\eta' = \eta[y_1 \mapsto \eta_x, \ldots, y_n \mapsto \eta_x]$.

From (4.2) we obtain $\eta', \sigma \vdash\!\circ\ e_0 \rightsquigarrow v, \tau$ by a derivation of the same length. $\sigma \vDash \eta' : (\Gamma', \Delta)$ holds by Lem. 4.8, item 2. Applying the induction hypothesis now yields

$$\eta', \sigma \vdash^{n + \Phi_\sigma(\eta' : \Gamma') + \Phi_\sigma(\eta' : \Delta)}_{n' + \Phi_\tau(v : r) + \Phi_\tau(\eta' : \Delta)} e_0 \rightsquigarrow v, \tau$$

By Lem. 4.7, item (4.6) holds $\Phi_\sigma(\eta_x : s) \geq \sum_i \Phi_\sigma(\eta_x : q_i)$ and hence $\Phi_\sigma(\eta : \Gamma) \geq \Phi_\sigma(\eta' : \Gamma')$. Since η and η' coincide on Δ, we also have $\Phi_\sigma(\eta : \Delta) = \Phi_\sigma(\eta' : \Delta)$ and $\Phi_\tau(\eta : \Delta) = \Phi_\tau(\eta' : \Delta)$. Hence we obtain

$$\eta, \sigma \vdash^{n + \Phi_\sigma(\eta : \Gamma) + \Phi_\sigma(\eta : \Delta)}_{n' + \Phi_\tau(v : r) + \Phi_\tau(\eta : \Delta)} e \rightsquigarrow v, \tau$$

by Lemma 2.8, as required.

◊Waste Assume that (4.1) was established in the last step by application of rule ◊Waste, so $\Theta \vdash^{u}_{u'} e : D^s$ and $\Gamma <: \Theta$ and $D^s <: C^r$. Furthermore $n \geq u$ and $n' + u \leq u' + n$. Since $\Gamma <: \Theta$ we have by Lem. 4.8, item 1, $\sigma \vDash \eta : (\Theta, \Delta)$. The induction hypothesis gives $\tau \vDash \eta[x_{res} \mapsto v] : (\Delta, x_{res} : D^s)$ from which we get $\tau \vDash \eta[x_{res} \mapsto v] : (\Delta, x_{res} : C^r)$ by Lem. 4.8, item 1. We also obtain $\eta, \sigma \vdash^{u + \Phi_\sigma(\eta : \Theta) + \Phi_\sigma(\eta : \Delta)}_{u' + \Phi_\tau(v : s) + \Phi_\tau(\eta : \Delta)} e \rightsquigarrow v, \tau$. We have by Lem. 4.7, item (4.5) $\Phi_\sigma(\eta : \Gamma) \geq \Phi_\sigma(\eta : \Theta)$, so we get by Lem. 2.8 and Lem. 2.7 $\eta, \sigma \vdash^{u + (n-u) + \Phi_\sigma(\eta : \Gamma) + \Phi_\sigma(\eta : \Delta)}_{u' + (n-u) + \Phi_\tau(v : s) + \Phi_\tau(\eta : \Delta)} e \rightsquigarrow v, \tau$. and finish with rule ⊢∘Waste.

⊢∘New By (4.1) we have $e = \mathtt{new}\ C$, $n = \Diamond\!(C^r) + \mathsf{Size}(C)$, $n' = 0$, $\Gamma = \varnothing$ and $v = \ell$. We have $\Phi_\tau(v : r) = \Diamond\!(C^r)$ since $[\![v.a_i]\!]_\tau = 0$ by rule ⊢∘New. Furthermore $\Phi_\sigma(\eta : \Delta) = \Phi_\tau(\eta : \Delta)$ since existent locations are not altered and even more importantly $\ell \notin \mathrm{dom}(\sigma)$ and (4.3) guarantees by Def. 2.2 and 2.1 that ℓ does not occur anywhere in η and σ. Therefore (4.7) follows. For (4.8) we note that $\mathscr{R}_{\tau, \eta[x_{res} \mapsto \ell], \Gamma}(\ell) = \{r\}$, but clearly $\curlyvee(r \,|\{r\})$.

⊢∘Free Similar to the case for rule ⊢∘New. The only noteworthy point is that τ, η satisfy 2.2 and 2.1 due to the difference between ⊢Free and ⊢∘Free as already discussed in Sect. 2.1.

⊢∘Update Suppose that (4.1) and (4.2) have been derived by rules ◊Update and ⊢∘Update. So we have $\eta_x = \ell$, $\tau = \sigma[\ell.a \mapsto \eta_y]$, $C_0 = [\![\eta_x]\!]^{\mathrm{dyn}}_\sigma$, $s_0 = \mathsf{A}^{\mathrm{set}}(C_0^r, a)$, $s = \mathsf{A}^{\mathrm{set}}(C^r, a)$, and $\Gamma = x : C^r, y : D^s$. Let furthermore $\Theta := \Delta, x : C^r$.

We assume $\eta_y \in \sigma$ for otherwise

$$\forall z \in \Theta, \vec{p}.\ [\![\eta_z.\vec{p}]\!]_\tau \in \tau \implies [\![\eta_z.\vec{p}]\!]_\tau = [\![\eta_z.\vec{p}]\!]_\sigma$$

hence the claim is trivially true. The case $\eta_y \in \sigma$ is more interesting for then τ may contain new valid path, possibly circular ones, whose effects on the potential and heap soundness we will now study.

For any locations ℓ_1, ℓ_2 let

$$\mathcal{P}(\ell_1, \ell_2) = \left\{ \vec{p} \;\middle|\; [\![\ell_1.\vec{p}]\!]_\sigma = \ell_2 \wedge \neg[\sigma, \ell_1, \vec{p} \between \ell, a] \right\} \tag{4.9}$$

denote the set of access paths from ℓ_1 to ℓ_2, which do not pass through $\ell.a$. Notice that by Lem. 2.3 we have

$$\left\{ \vec{p} \;\middle|\; [\![\ell_1.\vec{p}]\!]_\sigma = \ell_2 \wedge \neg[\sigma, \ell_1, \vec{p} \between \ell, a] \right\} = \left\{ \vec{p} \;\middle|\; [\![\ell_1.\vec{p}]\!]_\tau = \ell_2 \wedge \neg[\tau, \ell_1, \vec{p} \between \ell, a] \right\}$$

or in other words, the set $\mathcal{P}(\ell_1, \ell_2)$ remains unchanged if we replace σ by τ within the defining equation (4.9) for $\mathcal{P}(\cdot\,,\cdot)$. We now define for $z \in \Gamma, \Delta$

$$\mathcal{D}_z = \mathcal{P}(\eta_z, \ell) \qquad\qquad \mathcal{N}_z = \bigcup_{\ell_2 \in \sigma} \mathcal{P}(\eta_z, \ell_2)$$

Notice $\varepsilon \in \mathcal{D}_x$ and $\forall z\,.\,\mathcal{D}_z \subseteq \mathcal{N}_z$. Observe for $z \in \Theta$ that $\left\{ \vec{p} \;\middle|\; [\![\eta_z.\vec{p}]\!]_\tau \in \tau \right\} = \mathcal{N}_z \,\dot\cup\, (\mathcal{D}_z(.a.\mathcal{D}_y)^*.a.\mathcal{N}_y)$. For the "$\subseteq$"-direction, use the fact that a path $\vec{p}$ such that $[\![\eta_z.\vec{p}]\!]_\tau \in \tau$ either does not pass through $\ell.a$ in which case $\vec{p} \in \mathcal{N}_z$, or else it passes through $\ell.a$ a finite number of times and can the be decomposed as $\vec{p} = \vec{d}.a.\vec{e_1}\ldots.a.\vec{e_k}.\vec{r}$ where $\vec{d} \in \mathcal{D}_z$, $\vec{e_i} \in \mathcal{D}_y$ and $\vec{r} \in \mathcal{N}_y$.

Let $\Lambda = \Gamma, \Delta$. By assumption (4.3) and Def. 4.2 there exists C^r with $\curlyvee(C^r \,|\mathscr{R}_{\sigma,\eta,\Lambda}(\ell)\,)$ where

$$\begin{aligned}
\mathscr{R}_{\sigma,\eta,\Lambda}(\ell) &= \left\{ [\![(\eta_z : \Lambda_z).\vec{p}]\!]^{\mathrm{stat}}_\sigma \mid z \in \Lambda, [\![\eta_z.\vec{p}]\!]_\sigma = \ell \right\} \\
&\supseteq \left\{ [\![(\eta_z : \Lambda_z).\vec{p}]\!]^{\mathrm{stat}}_\sigma \mid z \in \Lambda, \vec{p} \in \mathcal{D}_z \right\} \\
&= \left\{ [\![(\eta_z : \Lambda_z).\vec{p}]\!]^{\mathrm{stat}}_\tau \mid z \in \Lambda, \vec{p} \in \mathcal{D}_z \right\}
\end{aligned}$$

where the last step follows since $[\![(\eta_z : \Lambda_z).\vec{p}]\!]^{\mathrm{stat}}_\sigma = [\![(\eta_z : \Lambda_z).\vec{p}]\!]^{\mathrm{stat}}_\tau$ for $\vec{p} \in \mathcal{D}_z$. Therefore by Lem. 3.4, item (3.10) we obtain

$$\curlyvee\left(C^r \;\middle|\; \left\{ [\![(\eta_z : \Lambda_z).\vec{p}]\!]^{\mathrm{stat}}_\tau \mid z \in \Lambda, \vec{p} \in \mathcal{D}_z \right\} \right)$$

and since $x \in \Lambda$ and $\varepsilon \in \mathcal{D}_x$ by Lem. 4.5

$$\curlyvee\left((C.a)^{\mathsf{A}^{\mathrm{set}}(C_0^r, a)} \;\middle|\; \left\{ [\![(\eta_z : \Lambda_z).\vec{p}.a]\!]^{\mathrm{stat}}_\tau \mid z \in \Lambda, \vec{p} \in \mathcal{D}_z \right\} \right)$$

By (4.3) we have $C_0 <: C$ and hence $D^s = (C.a)^{\mathsf{A}^{\mathrm{set}}(C^r a)} <: (C.a)^{\mathsf{A}^{\mathrm{set}}(C_0^r, a)}$ from (3.3). Therefore by Lem. 3.4, item (3.12)

$$\curlyvee\left(D^s \;\middle|\; \left\{ [\![(\eta_z : \Theta_z).\vec{q}]\!]^{\mathrm{stat}}_\tau \mid z \in \Theta, \vec{q} \in \mathcal{D}_z.a \right\} \;\uplus\; \left\{ [\![(\eta_y : s).\vec{q}]\!]^{\mathrm{stat}}_\tau \mid \vec{q} \in \mathcal{D}_y.a \right\} \right) \tag{4.10}$$

We will now prove

$$\curlyvee\Big(D^s \,\Big|\, \big\{ \mathbb{\llbracket}(\eta_z : \Theta_z).\vec{q} \mathbb{\rrbracket}_\tau^{\mathrm{stat}} \;\big|\; z \in \Theta, \vec{q} \in \mathcal{D}_z(a\mathcal{D}_y)^* a \big\} \Big) \tag{4.11}$$

by proving $\forall n \,.\, \mathrm{IH}(n)$ where we have

$$\mathrm{IH}(n) \equiv \curlyvee\Big(D^s \,\Big|\, \big\{ \mathbb{\llbracket}(\eta_z : \Theta_z).\vec{q} \mathbb{\rrbracket}_\tau^{\mathrm{stat}} \;\big|\; z \in \Theta, \vec{q} \in \mathcal{D}_z(a\mathcal{D}_y)^{\leq n} a \big\} \Big)$$

as our induction hypothesis.

IH(0) follows directly from (4.10) and Lem. 3.4, item (3.10).

Now suppose IH(n) holds for some n. For any $\vec{q_2} \in \mathcal{D}_y.a$ we have $\llbracket \eta_y.\vec{q_2} \rrbracket_\tau = \eta_y \in \tau$. By Lem. 4.7, item (4.4) we have

$$\curlyvee\Big(\mathbb{\llbracket}(\eta_y : s).\vec{q_2} \mathbb{\rrbracket}_\tau^{\mathrm{stat}} \;\Big|\; \big\{ \mathbb{\llbracket}(\eta_z : \Theta_z).\vec{q}.\vec{q_2} \mathbb{\rrbracket}_\tau^{\mathrm{stat}} \;\big|\; z \in \Theta, \vec{q} \in \mathcal{D}_z(a\mathcal{D}_y)^{\leq n} a \big\} \Big)$$

Hence we may replace the multiset $\Big\{ \mathbb{\llbracket}(\eta_y : s).\vec{q} \mathbb{\rrbracket}_\tau^{\mathrm{stat}} \mid \vec{q} \in \mathcal{D}_y.a \Big\}$ by $\big\{ \mathbb{\llbracket}(\eta_z : \Theta_z).\vec{q}.\vec{q_2} \mathbb{\rrbracket}_\tau^{\mathrm{stat}} \;\big|\; z \in \Theta, \vec{q} \in \mathcal{D}_z(a\mathcal{D}_y)^{\leq n} a \big\}$ in (4.10) thanks to Lem. 3.4, item (3.11) yielding IH($n+1$) as required.

We will now prove (4.7) which specialises to

$$\Phi_\sigma(\eta_y : s) + \Phi_\sigma(\eta : \Theta) \geq \Phi_\tau(\eta : \Theta)$$

In order to prove this equation, we expand

$$\Phi_\tau(\eta : \Theta) = \sum_{z \in \Theta} \sum_{\vec{p}} \phi_\tau\big((\eta_z : \Theta_z).\vec{p}\big)$$

which can be split by our earlier observations into

$$= \sum_{z \in \Theta} \sum_{\vec{p} \in \mathcal{N}_z} \phi_\tau\big((\eta_z : \Theta_z).\vec{p}\big) + \sum_{z \in \Theta} \sum_{\vec{q} \in \mathcal{D}_z(a\mathcal{D}_y)^* a} \sum_{\vec{r} \in \mathcal{N}_y} \phi_\tau\big((\eta_z : \Theta_z).\vec{q}.\vec{r}\big)$$

For the first addend we calculate

$$\begin{aligned} & \sum_{z \in \Theta} \sum_{\vec{p} \in \mathcal{N}_z} \phi_\tau\big((\eta_z : \Theta_z).\vec{p}\big) \\ = & \sum_{z \in \Theta} \sum_{\vec{p} \in \mathcal{N}_z} \phi_\sigma\big((\eta_z : \Theta_z).\vec{p}\big) \\ \leq & \, \Phi_\sigma(\eta : \Theta) \end{aligned}$$

and for the second addend we calculate

$$\begin{aligned}
&\sum_{z\in\Theta}\sum_{\vec{q}\in\mathcal{D}_z(a\mathcal{D}_y)^*a}\sum_{\vec{r}\in\mathcal{N}_y}\phi_\tau\big((\eta_z:\Theta_z).\vec{q}.\vec{r}\big)\\
=&\sum_{z\in\Theta}\sum_{\vec{q}\in\mathcal{D}_z(a\mathcal{D}_y)^*a}\sum_{\vec{r}\in\mathcal{N}_y}\phi_\tau\big((\eta_y:[\![(\eta_z:\Theta_z).\vec{q}]\!]^{\mathrm{stat}}_\tau).\vec{r}\big)\\
=&\sum_{z\in\Theta}\sum_{\vec{q}\in\mathcal{D}_z(a\mathcal{D}_y)^*a}\sum_{\vec{r}\in\mathcal{N}_y}\phi_\sigma\big((\eta_y:[\![(\eta_z:\Theta_z).\vec{q}]\!]^{\mathrm{stat}}_\tau).\vec{r}\big)\\
\leq&\sum_{z\in\Theta}\sum_{\vec{q}\in\mathcal{D}_z(a\mathcal{D}_y)^*a}\Phi_\sigma\Big(\eta_y:[\![(\eta_z:\Theta_z).\vec{q}]\!]^{\mathrm{stat}}_\tau\Big)\\
\leq&\,\Phi_\sigma(\eta_y:s)
\end{aligned}$$

where the last inequality follows by (4.11) and Lem. 4.7, item (4.6).

For proving (4.8) let ℓ_2 be an arbitrary location in $\mathrm{dom}(\tau)=\mathrm{dom}(\sigma)$. From (4.3) we have the existence of an E^q such that $\curlyvee(E^q\,|\mathscr{R}_{\sigma,\eta,\Lambda}(\ell)\,)$. Therefore by Lem. 3.4, item (3.10)

$$\curlyvee\left(E^q\;\middle|\;\begin{array}{c}\big\{[\![(\eta_z:\Theta_z).\vec{p}]\!]^{\mathrm{stat}}_\tau\;\big|\;z\in\Theta,\vec{p}\in\mathcal{P}(\eta_z,\ell_2)\big\}\\ \uplus\;\big\{[\![(\eta_y:s).\vec{p}]\!]^{\mathrm{stat}}_\tau\;\big|\;\vec{p}\in\mathcal{P}(\eta_y,\ell_2)\big\}\end{array}\right)\tag{4.12}$$

Note the change from σ to τ which is legitimate since only access paths in $\mathcal{P}(\cdot,\cdot)$ are considered.

From (4.11) and Lem. 4.7, item (4.4) we obtain

$$\forall\vec{p}\in\mathcal{P}(\eta_y,\ell_2)\,.\,\curlyvee\Big([\![(\eta_y:s).\vec{p}]\!]^{\mathrm{stat}}_\tau\,\Big|\big\{[\![(\eta_z:\Theta_z).\vec{q}.\vec{p}]\!]^{\mathrm{stat}}_\tau\;\big|\;z\in\Theta,\vec{q}\in\mathcal{D}_z(a\mathcal{D}_y)^*a,\big\}\Big)$$

Hence from (4.12) and Lem. 3.4, item (3.11) we obtain

$$\curlyvee\left(E^q\;\middle|\;\begin{array}{c}\big\{[\![(\eta_z:\Theta_z).\vec{p}]\!]^{\mathrm{stat}}_\tau\;\big|\;z\in\Theta,\vec{p}\in\mathcal{P}(\eta_z,\ell_2)\big\}\\ \uplus\;\big\{[\![(\eta_z:\Theta_z).\vec{p}]\!]^{\mathrm{stat}}_\tau\;\big|\;z\in\Theta,\vec{p}\in\mathcal{D}_z(a\mathcal{D}_y)^*a\mathcal{P}(\eta_y,\ell_2)\big\}\end{array}\right)$$

and therefore $\curlyvee(E^q\,|\mathscr{R}_{\tau,\eta,\Theta}(\ell_2)\,)$ as required, since

$$\begin{aligned}
\mathscr{R}_{\tau,\eta,\Theta}(\ell_2)&=\big\{[\![(\eta_z:\Theta_z).\vec{p}]\!]^{\mathrm{stat}}_\tau\;\big|\;z\in\Theta,[\![\eta_z.\vec{p}]\!]_\tau=\ell_2\big\}\\
&=\big\{[\![(\eta_z:\Theta_z).\vec{p}]\!]^{\mathrm{stat}}_\tau\;\big|\;z\in\Theta,\vec{p}\in\mathcal{P}(\eta_z,\ell_2)\cup\mathcal{D}_z(a\mathcal{D}_y)^*a\mathcal{P}(\eta_y,\ell_2)\big\}
\end{aligned}$$

The rest is now standard and omitted for lack of space.

$\vdash\circ$Invocation In this case

$$e=x.m(y_1,\ldots,y_j)\tag{4.13}$$

$$\Gamma=x:C^r,y_1:E_1^{q_1},\ldots y_j:E_j^{q_j}\tag{4.14}$$

$$\big(E_1^{q_1},\ldots,E_j^{q_j}\xrightarrow{n/n'}E_0^{q_0}\big)\in\mathsf{M}(C^r,m)\tag{4.15}$$

The premises of (4.2) yield:

$$\eta_x \in \sigma \tag{4.16}$$
$$\sigma(\eta_x) = (D, a_1 : v_1, \ldots, a_k : v_k) \tag{4.17}$$
$$\mathsf{M}_{\mathsf{body}}(D, m) = e_0 \tag{4.18}$$
$$\eta', \sigma \vdash e_0 \rightsquigarrow v, \tau \tag{4.19}$$

where $\eta' = [\mathtt{this} \mapsto \eta_x, x_1 \mapsto \eta_{y_1}, \ldots, x_j \mapsto \eta_{y_j}]$. Without loss of generality $\mathrm{dom}(\eta) \cap \mathrm{dom}(\eta') = \varnothing$.

We have $D <: C$ by (4.3). From the conditions on the class table ($\Diamond MBody$) we obtain q, s such that $\curlyvee(D^r \,|D^q, D^s)$.

$$\begin{aligned}
&\Phi_\sigma\big(\eta : \Delta, x : C^r, y_1 : E_1^{q_1}, \ldots, y_j : E_j^{q_j}\big) \\
&= \Phi_\sigma\big(\eta\eta' : \Delta, x_1 : E_1^{q_1}, \ldots, x_j : E_j^{q_j}\big) + \Phi_\sigma(\eta_x : r) \\
&\geq \Phi_\sigma\big(\eta\eta' : \Delta, x_1 : E_1^{q_1}, \ldots, x_j : E_j^{q_j}\big) + \Phi_\sigma(\eta_x : q) + \Phi_\sigma(\eta_x : s) \\
&\geq \Phi_\sigma\big(\eta\eta' : \Delta, \mathtt{this} : D^q, x_1 : E_1^{q_1}, \ldots x_j : E_j^{q_j}\big) + \Diamond(D^s)
\end{aligned} \tag{4.20}$$

where the crucial part of the last step follows by $[\![\eta_x.\varepsilon]\!]_\sigma^{\mathrm{dyn}} = D$ and therefore $\Diamond(D^s)$ is a part of the sum $\Phi_\sigma(\eta_x : s)$.

Again by instantiating the class table rule ($\Diamond MBody$) for class D^r we obtain

$$\mathtt{this} : D^q, x_1 : E_1^{q_1}, \ldots, x_j : E_j^{q_j} \;\vdash^{\frac{n + \Diamond(D^s)}{n'}}\; e_0 : E_0^{q_0}$$

by application of $\Diamond$WASTE, $D <: C$ and method subtyping. Note that we do not in general have this judgement with D replaced by C, since $\Diamond(D^s) \geq \Diamond(C^s)$.

By (4.2) and Lem.2.5 we have $\eta\eta', \sigma \vdash e_0 \rightsquigarrow v, \tau$. Application of the induction hypothesis then yields

$$\eta\eta', \sigma \;\vdash^{\frac{n_0}{n' + \Phi_\tau(\eta[x_{res} \mapsto v] \,:\, \Delta, x_{res})}}\; e_0 \rightsquigarrow v, \tau \tag{4.21}$$

where $n_0 = n + \Diamond(D^s) + \Phi_\sigma\big(\eta\eta' : \Delta, \mathtt{this} : D^q, x_1 : E_1^{q_1}, \ldots, x_j : E_j^{q_j}\big)$.

By (4.20) then follows the conclusion.

□

This corollary is a direct consequence of the main result and it is in this form that they intend to use it.

Corollary 4.10 *Suppose that $\mathscr{C}$ is an FJEU program containing (in Java notation) a class* `List` *of singly-linked lists with boolean entries, a class* `C` *containing a method* `void C.main(List args)`*, and arbitrary other classes and methods.*

Suppose furthermore, that there exists a RAJA-annotation of this program containing a view a where $\Diamond(\mathtt{List}^a) = k \in \mathbb{N}$ and $\mathsf{A}^{\mathrm{get}}(\mathtt{List}^a, \mathtt{next}) = a$ then evaluating `C.main`$(args)$ *in a heap where* `args` *points to a linked list of length l requires at most kl memory cells.*

References

[AAG+07] Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. COSTA: Design and implementation of a cost and termination analyzer for java bytecode. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *FMCO*, volume 5382 of *Lecture Notes in Computer Science*, pages 113–132. Springer, 2007.

[BIL03] Marius Bozga, Radu Iosif, and Yassine Laknech. Storeless semantics and alias logic. In *Proceedings of the 2003 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation (PEPM)*, pages 55–65, New York, NY, USA, 2003. ACM Press.

[Cam08] Brian Campbell. *Type-based amortized stack memory prediction.* PhD thesis, University of Edinburgh, 2008.

[Deu94] Alain Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. *ACM SIGPLAN Notices*, 29(6):230–241, 1994.

[FKF98] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '98)*, pages 171–183, New York, January 1998. Association for Computing Machinery.

[GL98] Gustavo Gómez and Yanhong A. Liu. Automatic accurate cost-bound analysis for high-level languages. In Frank Mueller and Azer Bestavros, editors, *Languages, Compilers, and Tools for Embedded Systems, ACM SIGPLAN Workshop LCTES'98, Montreal, Canada.* Springer, 1998. LNCS 1474.

[Gro01] Bernd Grobauer. *Topics in Semantics-based Program Manipulation.* PhD thesis, BRICS Aarhus, 2001.

[HBH+07] Christoph A. Herrmann, Armelle Bonenfant, Kevin Hammond, Steffen Jost, Hans-Wolfgang Loidl, and Robert Pointon. Automatic amortised worst-case execution time analysis. In *7th Int'l Workshop on Worst-Case Execution Time (WCET) Analysis, Proceedings*, pages 13–18, 2007.

[HDF+05] Kevin Hammond, Roy Dyckhoff, Christian Ferdinand, Reinhold Heckmann, Martin Hofmann, Steffen Jost, Hans-Wolfgang Loidl, Greg Michaelson, Robert F. Pointon, Norman Scaife, Jocelyn Srot, and Andy Wallace. The embounded project (project start paper). In Marko C. J. D. van Eekelen, editor, *Trends in Functional Programming*, volume 6 of *Trends in Functional Programming*, pages 195–210. Intellect, 2005.

[HJ03] Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *POPL: 30th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2003.

[HJ06] Martin Hofmann and Steffen Jost. Type-based amortised heap-space analysis (for an object-oriented language). In Peter Sestoft, editor, *Proceedings of the 15th European Symposium on Programming (ESOP), Programming Languages and Systems*, volume 3924 of *LNCS*, pages 22–37. Springer, 2006.

[HP99] John Hughes and Lars Pareto. Recursion and dynamic data-structures in bounded space:, June 21 1999.

[IO01] Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 14–26, New York, NY, USA, 2001. ACM Press.

[IPW99] Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In Loren Meissner, editor, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34(10), pages 132–146, N. Y., 1999.

[Jon81] H.B.M. Jonkers. Abstract storage structures. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 321–343. IFIP, North Holland, 1981.

[KN04] Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. Technical Report 0400001T.1, National ICT Australia, Sydney, March 2004.

[NCQR05] Huu Hai Nguyen, Wei Ngan Chin, Shengchao Qin, and Martin C. Rinard. Memory usage inference for object-oriented programs. January 2005.

[Oka98] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[RBR+05] Noam Rinetzky, Jörg Bauer, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. A semantics for procedure local heaps and its abstractions. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 296–309, New York, NY, USA, 2005. ACM Press.

[Tar85] Robert E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, April 1985.

[WM01] David Walker and Greg Morrisett. Alias types for recursive data structures. *Lecture Notes in Computer Science*, 2071:177+, 2001.