Automatic Type Inference for Amortised Heap-Space Analysis

Martin Hofmann and Dulma Rodriguez

[1] University of Munich martin.hofmann@ifi.lmu.de
[2] Monoidics Ltd dulma.rodriguez@monoidics.com

Abstract. We present a fully automatic, sound and modular heap-space analysis for object-oriented programs. In particular, we provide type inference for the system of refinement types RAJA, which checks upper bounds of heap-space usage based on amortised analysis. Until now, the refined RAJA types had to be manually specified. Our type inference increases the usability of the system, as no user-defined annotations are required.
The type inference consists of constraint generation and solving. First, we present a system for generating subtyping and arithmetic constraints based on the RAJA typing rules. Second, we reduce the subtyping constraints to inequalities over infinite trees, which can be solved using an algorithm that we have described in previous work. This paper also enriches the original type system by introducing polymorphic method types, enabling a modular analysis.
Keywords: Type systems, resource analysis, memory management.

1 Introduction

We study the problem of predicting the dynamic memory allocation of an object-oriented program in a freelist-based memory model. In short, we compute a number N such that at any point in the execution of the program the number of "new" instructions executed thus far minus the number of "free" instructions executed thus far does not exceed N. This (perhaps) seemingly simple task is complicated by the following factors:

- The computed bound N should be symbolic, i.e. a closed form expression in the size of the input which is provided, e.g., as a list of strings;
- The control flow of the program depends on the input and on the shape of intermediate data structures like lists or trees;
- The control flow strongly depends on dynamic class tags as is common in class-based object-oriented programming.

We remark that there is nothing special about "new" and "free"; one can in just the same way count the number of "tick" instructions and in this way obtain upper bounds on execution time, or indeed on the expenditure of any other quantifiable resource (number of open connections, text messages sent, real money spent, etc.).

The need for resource prediction has been widely recognised [1–3] and is also intuitively plausible. Just think of software running on small, resource-constrained devices such as smart cards, microcontrollers, phones, or software running on large servers shared between many users as in cloud computing. While there is still some way to go until we can serve these applications at industrial level, there has been considerable progress in the last years.

Approaches based on recurrence solving [1, 4, 5] or on abstract interpretation [6, 2, 3] have matured to a point where programs of several hundred lines of code can be automatically analysed. These techniques work under the assumption that control-flow is either fixed or determined by some easily obtainable numeric parameters such as length or size of input or linear arithmetic functions thereof. Other dependencies of the control flow are over-approximated by simply taking the maximum over all possible runs. This works well for programs which use arrays that are allocated at the beginning with a given size and processed with a simple iteration pattern. This is very useful in embedded systems or scientific computing where most programs have such a shape. It does not work well with object-oriented programs where resource behaviour depends on the dynamic class tags of objects, such as when functional data structures such as lists or trees are implemented using the Composite pattern.

In previous work [7–10] we and others argued that the method of amortised analysis [11, 12] might be of help here. Therein, data structures are assigned non-negative numbers, called *potential*, in an *a priori* arbitrary fashion. If done cleverly, it then becomes possible to obtain *constant* bounds on the "amortised cost" of an individual operation, that is, its actual resource usage plus the difference in potential of the data structure before and after performing the operation. This makes it possible to take into account the effect that an operation might have on the resource usage of subsequent operations and also to merely add up amortised costs without having to explicitly track size and shape of intermediate data structures.

In traditional amortised analysis [11] where the emphasis lies on the manual analysis of algorithms, the potentials were ascribed to particular data structures such as union-find trees by some formula that must be manually provided. When amortised analysis is used for automatic resource analysis one uses refined types to define the potentials — typing rules then ensure that potential and actual resource usage is accounted for correctly. Combined with type inference, it then allows for an automatic inference of the potential functions.

In amortised resource analysis for statically typed functional programs the data structures remain fixed (e.g. lists or trees) and only the potential functions must be found. In the object-oriented case, even the data structures themselves must be discovered by the analysis because objects can be used for just anything be it lists, trees, graphs, etc. As a result, automatic inference becomes considerably more challenging unless one is willing to accept user annotations specifying the way in which objects are to be used, for instance in the form of separation logic annotations [13].

Here, we investigate how far we can go without requiring any such annotations. We build upon a system of refinement types for amortised analysis for a Java-like language called Resource Aware JAva (RAJA) [7]. This is a powerful type system that can capture the heap-space requirements of many programs. Moreover, the type system takes aliasing into account, which means that the resource analysis based on this system is sound for programs which contain shared or even cyclic data structures. In previous work [8], we have described a type checking algorithm for RAJA and an implementation capable of checking user supplied typing annotations which were still quite cumbersome and difficult to come up with and this hindered practical use.

In this paper, we remove this obstacle and show how the refinement types can be inferred, so as to make the analysis fully automatic and eliminate the burden of manual annotations from the programmer. The main contributions of this paper are as follows:

1. We provide for the first time fully automatic amortised resource inference for object-oriented programs, which is sound and modular.
2. We reduce the problem of type inference for RAJA to the problem of satisfiability of inequalities over infinite trees labelled with non-negative real numbers.
3. We validate the type system RAJA and the type inference algorithm with a publicly available implementation and experimental evaluation.

In previous work [14] we presented the novel problem of satisfiability of arithmetic constraints over infinite labelled trees. Moreover, we provided a heuristic algorithm for constraint solving, which consists of reducing the constraints to an equivalent finite set of linear arithmetic constraints. This was possible in many cases when the solutions were regular trees.

The fact that we can only solve constraints when their solutions are regular trees implies that our analysis is restricted to linear bounds. We shall explain this connection later when we describe how we obtain the bounds from the infinite trees. Since this problem has only been described recently, it is still unknown whether it is decidable. If it is decidable and an algorithm for solving the constraints was found, we would be able to compute non-linear bounds with the same method.

The type system that we present in this paper is a slightly modified version of the original type system from [7]: we present syntax-directed typing rules that make the system more suitable for automatic type inference. We also introduce polymorphic method types that enable a modular analysis. However, we do not allow polymorphic recursion since it would cause many difficulties to the type inference and we have not found useful examples where it is required.

This paper is organised as follows. In the next section we give an informal presentation of the system and show its use in some examples. In Section 3 we describe briefly our target language FJEU and we introduce formally the typing system RAJA. Section 4 describes the type inference algorithm. In Section 5 we show experimental results. Finally we review related work and conclude in Section 6.

```
1  abstract class List {
2      abstract List copy();abstract DList toDList(DList prev);}
3  class Nil extends List {
4      List copy() { return this; }
5      DList toDList(DList prev) {return new DNil();}}
6  class Cons extends List  { List next; int elem;
7      List copy() {
8          Cons res = new Cons();
9          res.elem = this.elem;
10         res.next = this.next.copy(); return res; }
11     DList toDList(DList prev) {
12         DCons res = new DCons();
13         res.elem = this.elem;
14         res.next = this.next.toDList(res);
15         res.prev = prev; return res; }}
16 abstract class DList  { }
17 class DNil extends DList  { }
18 class DCons extends DList { int elem; DList next; DList prev;}
19 class Main {
20     List main_copy(List list) {return list.copy();}
21     DList main_dlist(List list) {return list.toDList(new DNil());} }
```

Fig. 1. Example program

2 Informal presentation and examples

We aim to statically analyse the heap-space consumption of class-based object-oriented programs. Since we wish to abstract from concrete memory models, we assume a simple freelist based model where we maintain a set of free memory units, the freelist. When creating an object, a heap unit required to store it is taken from the freelist, provided it contains enough units. When deallocating an object, the unit returns to the freelist. We remark that we deallocate objects explicitly by means of a free expression, since we assume no garbage collection. We also assume that any attempt to access a previously deallocated object leads to immediate abortion of the program and all resource predictions are on condition that no such abortion takes place. Static analysis for preventing such illicit accesses is an orthogonal problem and not addressed in this paper.

We also note that we can treat free-instructions as no-ops and use a garbage collector. Assuming that the garbage collector discovers all deallocation opportunities and that it is invoked whenever the freelist becomes short then our inferred bounds are also valid in the presence of garbage collection. We have not explored this avenue in detail, however.

We then demonstrate the front end of our method with a couple of small examples. Fig. 1 shows a method for copying a singly-linked-list and a method for converting a singly-linked-list into a doubly-linked-list. Here we use Java syntax to simplify the understanding of the programs; the syntax of our target language FJEU is slightly different. Consider the method main_copy. Running the analysis yields the following results; no annotations by the programmer are required. The length of the input refers to the length of the list given as argument to the method.

```
Program will execute successfully with a free-list >= |input|
```

It is clear that the heap-space consumption of this program is exactly the length of the list. When we analyse the method main_dlist we obtain the following:

`Program will execute successfully with a free-list >= 2 + |input|`

Here the heap-space consumption is the length of the list plus the two DNil objects that represent the two ends of the doubly-linked list.

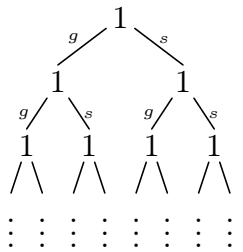
Our goal is to find (statically) an upper bound on the initial size of the freelist so that the given program can be executed without running out of memory. We seek to assign data structures a potential that can be used to pay for any object creation. Then, the potential of the data structures in their initial state will represent an upper bound on the total heap consumption of the program.

We wish to assign different objects of the same class different potentials, thus, we need to *refine* the notion of classes. We introduce the *views*, a set of names, which, together with the classes, build the appropriate refined types to which we will assign potential. Moreover, since classes are compound types consisting of fields and methods, we need to give refined types for these also. A refined type consist of a class C and a view r, written C^r. The potential function $\Diamond(.)$ assigns each refined type a potential, which is a non-negative real number. The functions $\mathsf{A}^{\mathrm{get}}(\cdot\,,\cdot)$ and $\mathsf{A}^{\mathrm{set}}(\cdot\,,\cdot)$ assign views to the fields, where $\mathsf{A}^{\mathrm{get}}(C^r, a)$ represents the view used when reading the field a of class C under the view r, and $\mathsf{A}^{\mathrm{set}}(C^r, a)$ is the view used when writing a.

Thus, views consist of a set of names, together with maps $\Diamond(\cdot), \mathsf{A}^{\mathrm{get}}(\cdot\,,\cdot)$ and $\mathsf{A}^{\mathrm{set}}(\cdot\,,\cdot)$. Alternatively, we can see them as infinite trees, where nodes are labelled by a tuple of non-negative real numbers (one number for each class in the given program), and edges are labelled with elements of the set

$$\{C.a.\mathsf{get}, C.a.\mathsf{set} \mid C \text{ is a class and } a \text{ is a field of } C\}$$

For instance, if we assume that the only class in the program is Cons and g denotes Cons.next.get and s denotes Cons.next.set, then the following tree represents a view:



This view is regular, because it contains only finitely many different subtrees. We define an inequality relation $\sqsubseteq$ on views, which is covariant in the get subtrees and contravariant in the set subtrees. We also define subtyping over refined types: C^r is a subtype of D^s iff C is a subclass of D and $r \sqsubseteq s$.

A monomorphic method type for a method m consists of views for the method's arguments, a view for its result and two numbers representing the potential consumed and released by the method respectively. More concretely, if a method m has a type $C^{r_0}; E_1^{r_1}, \ldots, E_j^{r_j} \xrightarrow{n/n'} H^{r_{j+1}}$, this means that it is defined in the refined type C^{r_0} and 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

$$
\begin{aligned}
\mathsf{Nil.copy}() &= \mathsf{Nil}^{\mathsf{v_{self}}} \xrightarrow{q_1/q_2} \mathsf{List}^{\mathsf{v_{res}}} \;\&\; \mathsf{v_{self}} \sqsubseteq \mathsf{v_{res}} \\
\mathsf{Cons.copy}() &= \mathsf{Cons}^{\mathsf{v_{self}}} \xrightarrow{q_1/q_2} \mathsf{List}^{\mathsf{v_{res}}} \;\& \\
&\quad\; \mathsf{A}^{\mathrm{get}}(\mathsf{Cons}^{\mathsf{v_{self}}}, \mathsf{next}) \sqsubseteq \mathsf{v_{self}} \;\wedge\; \Diamond(\mathsf{Cons}^{\mathsf{v_{self}}}) \geq \Diamond(\mathsf{Cons}^{\mathsf{v_{res}}}) + 1 \\
\mathsf{Main.main\_copy}() &= \mathsf{Main}^{\mathsf{v_{self}}}; \mathsf{List}^{\mathsf{v_l}} \xrightarrow{q_1/q_2} \mathsf{List}^{\mathsf{v_{res}}} \;\&\; \mathsf{v_l} \sqsubseteq \mathsf{v_{res}} \\
&\quad\; \mathsf{A}^{\mathrm{get}}(\mathsf{Cons}^{\mathsf{v_l}}, \mathsf{next}) \sqsubseteq \mathsf{v_l} \wedge \Diamond(\mathsf{Cons}^{\mathsf{v_l}}) \geq \Diamond(\mathsf{Cons}^{\mathsf{v_{res}}}) + 1
\end{aligned}
$$

$\Diamond(\cdot)$	rich	poor
List, Nil, Main	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

Fig. 2. RAJA types for the copy example.

of n. The return value will be of type $H^{r_{j+1}}$, carrying an according potential. In addition to this, a potential of n' units will be returned.

Polymorphic method types are like monomorphic RAJA method types, but views and numbers replaced by variables and constraints upon them. A polymorphic method type consists of view variables for its arguments, a view variable for its result and two number variables. Moreover, it contains a conjunction of subtyping and linear arithmetic constraints that capture the resource consumption of the method. The subtyping constraints show how the views for the arguments and result relate. For instance, the constraint $\mathsf{A}^{\mathrm{get}}(C^v, a) \sqsubseteq w$ means that given a valuation π that maps view variables to views $\pi = \{v \mapsto r, w \mapsto s\}$, the get view of the field a of class C under the view r must be a subtype of s.

One run-time object can have several refined types at once, since it can be regarded through different views at the same time. The overall potential of a run-time configuration is the (possibly infinite) sum over all access paths in scope that lead to an actual object. Thus, if an object has several access paths leading to it (aliasing), it may make several contributions to the total potential. Our type system has an explicit contraction rule: If a variable is used more than once, the associated potential is split by assigning different views to each use.

Analysis of list copy. In the following, we wish to illustrate the system by showing the details of the analysis of main_copy from Fig. 1. We shall explain a simplified form of the constraints obtained by analysing the program. We assume that for each method, we assign the view variable $\mathsf{v_{self}}$ to the variable this and the view variable $\mathsf{v_{res}}$ to the result of the function. When analysing the body of Nil.copy, we obtain the constraint $\mathsf{v_{self}} \sqsubseteq \mathsf{v_{res}}$ (line 4). Further, in the method Cons.copy, line 8 produces the constraint $\Diamond(\mathsf{Cons}^{\mathsf{v_{self}}}) \geq \Diamond(\mathsf{Cons}^{\mathsf{v_{res}}}) + 1$, because the current object needs to pay for the creation of the new Cons object and also for its potential. Moreover, since the method is called recursively with the next item in the list (line 10), the refined type of the next node must be a subtype of the refined type of the current node, which is expressed in the constraint $\mathsf{A}^{\mathrm{get}}(\mathsf{Cons}^{\mathsf{v_{self}}}, \mathsf{next}) \sqsubseteq \mathsf{v_{self}}$. The method List.copy is abstract, so we obtain no constraints. However, a virtual call to it may be resolved to a call to Nil.copy() or to Cons.copy(). Thus, to ensure soundness, we need to add the constraints of Cons.copy() and Nil.copy() to List.copy(). Then, when we call the method List.copy() in main_copy, we obtain the appropriate constraints after variable substitution (see Fig. 2).

The valuation $\pi = (\{\mathsf{v_{self}} \mapsto \mathsf{rich}, \mathsf{v_{res}} \mapsto \mathsf{poor}\}, \{q_1 \mapsto 0, q_2 \mapsto 0\})$ builds the best possible solution for the constraints. Our algorithm infers the following monomorphic method type for main_copy: $\mathsf{Main^{rich}}; \mathsf{List^{rich}} \xrightarrow{0/0} \mathsf{List^{poor}}$. This type says that the heap consumption of main_copy is bounded by the potential of the list l. The potential of l is calculated as the sum over all access paths starting from l and not leading to null. Each of these has a dynamic type: Cons, or Nil for the end of the list. Each also has a view that can be computed by chaining the view of l along the get views, which is the view rich in each case. For each access path, we look up the potential annotation of its dynamic type under its view. Given $\Diamond(\mathsf{Cons^{rich}}) = 1$ and $\Diamond(\mathsf{Nil^{rich}}) = 0$, this is 1 in every case except for the path leading to Nil. The resulting sum is the length of the list $|l|$.

Now, imagine that the view rich was defined differently, as the first element of the following family of views:

$$\Diamond(\mathsf{Cons^{rich_i}}) = 2^i, \mathsf{A^{get}}(\mathsf{Cons^{rich_i}}; \mathsf{next}) = \mathsf{rich}_{i+1}, \mathsf{A^{set}}(\mathsf{Cons^{rich_i}}; \mathsf{next}) = \mathsf{rich}_{i+1}, i \geq 0$$

Then, the potential of the list l would be $2^{|l|} - 1$, thus we could obtain an exponential bound for the heap requirements of the method. Also notice that the view rich would not be regular. Therefore, we cannot compute such bounds at the moment, because of the restrictions of our constraint solver.

To conclude this example, we wish to give an intuition for the need for refined types. Imagine that we could give potential only to the class Cons. Line 8 would then produce the constraint $\Diamond(\mathsf{Cons}) \geq \Diamond(\mathsf{Cons}) + 1$, which is unsatisfiable. We require more sophisticated types to achieve a more refined behaviour: a Cons object with potential 1 can be copied, but the result is a Cons object with potential 0, which cannot.

3 System RAJA

FJ with Update. Our formal model of Java, FJEU, is an extension of Featherweight Java (FJ) [15] with attribute update, conditional and explicit deallocation. An FJEU program $\mathcal{P} = (\mathscr{C}, \mathsf{main})$ consists of a partial finite map from class names to class definitions $\mathscr{C}$, and a distinguished method main to be executed when running the program. 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 fields of C, including inherited ones. We write $C.a$ to denote the class type of each field a of class C. Similarly we write $\mathsf{Meth}(C)$ to denote the set of all defined method names of C, including inherited ones. For a method name m of class C we write $\mathsf{M_{body}}(C, m)$ to denote the term that comprises the *method body* of method m and $C.m$ to denote the *method type* of m in class C. 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. Each class has only one implicit constructor, which sets all class attributes to a null value.

We now extend FJEU to an annotated version, Resource Aware JAva (RAJA). We set $\mathbb{D} = \mathbb{R}_0^+ \cup \{\infty\}$, i.e., the set of non-negative real numbers together with an element ∞. Ordering and addition on $\mathbb{R}_0^+$ extend to $\mathbb{D}$ by $\infty + x = x + \infty = \infty$ and $x \leq \infty$.

Definition 1. *We define the set $\mathscr{V}$ of* views *coinductively by*

- $\Diamond\!(\cdot)$ *assigns to each view $r \in \mathscr{V}$ and class $C \in \mathscr{C}$ a number $\Diamond\!(C^r)$.*
- $\mathsf{A}^{\mathrm{get}}(\cdot\,,\cdot)$ *assigns to each view $r \in \mathscr{V}$ and class $C \in \mathscr{C}$ and field $a \in \mathsf{A}(C)$ a view $s = \mathsf{A}^{\mathrm{get}}(C^r, a)$.*
- $\mathsf{A}^{\mathrm{set}}(\cdot\,,\cdot)$ *assigns to each view $r \in \mathscr{V}$ and class $C \in \mathscr{C}$ and field $a \in \mathsf{A}(C)$ a view $s' = \mathsf{A}^{\mathrm{set}}(C^r, a)$.*

The following inequality relation $\sqsubseteq$ is covariant in the get views and contravariant in the set views.

Definition 2 ($\mathbf{r \sqsubseteq s}$). *Let $r, s \in \mathscr{V}$. We define $r \sqsubseteq s$ coinductively by*

$$\forall C \in \mathscr{C}\,.\,\Diamond\!(C^r) \geq \Diamond\!(C^s)$$
$$\forall C \in \mathscr{C}\,\forall a \in \mathsf{A}(C)\,.\,\mathsf{A}^{\mathrm{get}}(C^r, a) \sqsubseteq \mathsf{A}^{\mathrm{get}}(C^s, a)$$
$$\forall C \in \mathscr{C}\,\forall a \in \mathsf{A}(C)\,.\,\mathsf{A}^{\mathrm{set}}(C^s, a) \sqsubseteq \mathsf{A}^{\mathrm{set}}(C^r, a)$$

We define the operations on views $\oplus : \mathscr{V} \times \mathscr{V} \to \mathscr{V}$ and $\boxplus : \mathscr{V} \times \mathscr{V} \to \mathscr{V}$ simultaneously as follows. Let $s_1, s_2 \in \mathscr{V}$ then, for each $C \in \mathscr{C}$, $a \in \mathsf{A}(C)$ we set:

$$\begin{array}{ll} \Diamond\!\left(C^{s_1 \oplus s_2}\right) = \Diamond\!(C^{s_1}) + \Diamond\!(C^{s_2}) & \Diamond\!\left(C^{s_1 \boxplus s_2}\right) = \mathsf{min}(\Diamond\!(C^{s_1}), \Diamond\!(C^{s_2})) \\ \mathsf{A}^{\mathrm{get}}\!\left(C^{s_1 \oplus s_2}, a\right) = \mathsf{A}^{\mathrm{get}}(C^{s_1}, a) \oplus \mathsf{A}^{\mathrm{get}}(C^{s_2}, a) & \mathsf{A}^{\mathrm{get}}\!\left(C^{s_1 \boxplus s_2}, a\right) = \mathsf{A}^{\mathrm{get}}(C^{s_1}, a) \boxplus \mathsf{A}^{\mathrm{get}}(C^{s_2}, a) \\ \mathsf{A}^{\mathrm{set}}\!\left(C^{s_1 \oplus s_2}, a\right) = \mathsf{A}^{\mathrm{set}}(C^{s_1}, a) \boxplus \mathsf{A}^{\mathrm{set}}(C^{s_2}, a) & \mathsf{A}^{\mathrm{set}}\!\left(C^{s_1 \boxplus s_2}, a\right) = \mathsf{A}^{\mathrm{set}}(C^{s_1}, a) \oplus \mathsf{A}^{\mathrm{set}}(C^{s_2}, a) \end{array}$$

Let $\dot{-} : \mathbb{D} \times \mathbb{D} \to \mathbb{D}$ be defined by: $n \dot{-} m = \begin{cases} n - m & \text{if } n - m \geq 0 \\ 0 & \text{otherwise} \end{cases}$.

We define an operation $(s \dot{-} \mathsf{n})_D : \mathscr{V} \times \mathbb{D} \times \mathscr{C} \to \mathscr{V}$ that takes a view s and a number $n \in \mathbb{D}$ and class D and returns another view that is just like s, except for the potential of D^s, which is $\Diamond\!(D^s) \dot{-} n$. We set for each $C \in \mathscr{C}$ and each $a \in \mathsf{A}(C)$:

$$\Diamond\!\left(C^{(s \dot{-} \mathsf{n})_D}\right) = \begin{cases} \Diamond\!(C^s) \dot{-} n & \text{if } C = D \\ \Diamond\!(C^s) & \text{otherwise} \end{cases} \qquad \begin{array}{l} \mathsf{A}^{\mathrm{get}}\!\left(C^{(s \dot{-} \mathsf{n})_D}, a\right) = \mathsf{A}^{\mathrm{get}}(C^s, a) \\ \mathsf{A}^{\mathrm{set}}\!\left(C^{(s \dot{-} \mathsf{n})_D}, a\right) = \mathsf{A}^{\mathrm{set}}(C^s, a) \end{array}$$

A refined type consists of a class C and a view r and is written C^r. We extend the subtyping of FJEU classes to refined types as follows. Since both $\sqsubseteq$ and $<:$ on FJEU are reflexive and transitive so is $<:$ on RAJA.

Definition 3 ($\mathbf{C^r <: D^s}$). *We extend subtyping to refined types by $C^r <: D^s$ iff $C <: D$ and $r \sqsubseteq s$.*

In the following grammar, we define subtyping and arithmetic constraints. tt is the empty constraint, i.e. a constraint that is always satisfied. Moreover, $n \in \mathbb{D}$, v ranges over view variables and p over arithmetic variables.

$$\begin{array}{ll} \mathsf{vexp} ::= v \mid \mathsf{A}^{\mathrm{get}}(C^v, a) \mid \mathsf{A}^{\mathrm{set}}(C^v, a) \mid v \oplus v & \mathcal{TC} ::= C^{\mathsf{vexp}} <: D^{\mathsf{vexp}} \\ \mathsf{ae} \quad ::= n \mid p \mid \Diamond\!(C^v) \mid \mathsf{ae} + \mathsf{ae} & \mathcal{AC} ::= \mathsf{ae}_1 \geq \mathsf{ae}_2 \mid \mathsf{ae}_1 \leq \mathsf{ae}_2 \\ & \mathcal{C} \quad ::= \mathcal{AC} \mid \mathcal{TC} \mid \mathcal{C} \wedge \mathcal{C} \mid \mathsf{tt} \end{array}$$

Let $\pi = (\pi_v, \pi_a)$ be a pair of maps: π_v is map from view variables to views and π_a is a map from number variables to numbers. We then define *the meaning*

of arithmetic expressions $\pi(\mathsf{ae})$ in the obvious way, e.g. $\pi(\Diamond(C^v)) = \Diamond\big(C^{\pi_v(v)}\big)$. *The meaning of view expressions* $\pi(\mathsf{vexp})$ is defined as one might expect, e.g. $\pi(\mathsf{A}^{\mathrm{get}}(C^v, a)) = \mathsf{A}^{\mathrm{get}}\big(C^{\pi(v)}, a\big)$. We say that π satisfies a conjunction of constraints $\mathcal{C}$, written $\pi \models \mathcal{C}$, if π satisfies each constraint in $\mathcal{C}$.

If $\boldsymbol{v} = v_0, \ldots, v_{n+1}$ is a vector of length $n+2$, with $n \geq 0$, we write $\mathfrak{v}$ for meaning the (possibly empty) vector $v_1, \ldots, v_n$.

Definition 4 (An n-*ary monomorphic RAJA method type*).
An n-ary monomorphic RAJA method type T *consists of* $n+2$ *views* $\boldsymbol{s}$ *and two numbers* m_1, m_2 *written* $T = s_0; \mathfrak{s} \xrightarrow{m_1/m_2} s_{n+1}$.

We also write $C^{s_0}; \boldsymbol{E}^{\mathfrak{s}} \xrightarrow{m_1/m_2} H^{s_{n+1}}$ to denote an FJEU method type combined with a corresponding monomorphic RAJA method type.

Definition 5 (An n-*ary polymorphic RAJA method type*).
An n-ary polymorphic RAJA method type ϕ *consists of* $n+2$ *view variables* $\boldsymbol{v}$ *and two arithmetic variables* $\boldsymbol{q} = q_1, q_2$ *and existentially quantified (view and arithmetic) variables* $\boldsymbol{w}$, $\boldsymbol{t}$ *and a conjunction of subtyping and arithmetic constraints on them written* $\phi = \forall \boldsymbol{v}, \boldsymbol{q}\ \exists\, \boldsymbol{w}, \boldsymbol{t}\,.\, v_0; \mathfrak{v} \xrightarrow{q_1/q_2} v_{n+1}\ \&\ \mathcal{C}(\boldsymbol{v}, \boldsymbol{q}, \boldsymbol{w}, \boldsymbol{t})$.

We often write $\forall \boldsymbol{v}, \boldsymbol{q}\ \exists\, \boldsymbol{w}, \boldsymbol{t}\,.\, C^{v_0}; \boldsymbol{E}^{\mathfrak{v}} \xrightarrow{q_1/q_2} H^{v_{n+1}}\ \&\ \mathcal{C}(\boldsymbol{v}, \boldsymbol{q}, \boldsymbol{w}, \boldsymbol{t})$ to denote an FJEU method type combined with a corresponding polymorphic RAJA method type. A polymorphic method type stands for the set of all monomorphic types that satisfy its constraints. Because this type does not depend on the method's callers, the type inference for the method can be performed modularly.

Definition 6 (Instance of a polymorphic method type).
Let $T = C^{s_0}; \boldsymbol{E}^{\mathfrak{s}} \xrightarrow{m_1/m_2} H^{s_{n+1}}$ *be a monomorphic RAJA method type and* $\phi = \forall \boldsymbol{v}, \boldsymbol{q}\ \exists\, \boldsymbol{w}, \boldsymbol{t}\,.\, C^{v_0}; \boldsymbol{E}^{\mathfrak{v}} \xrightarrow{q_1/q_2} H^{v_{n+1}}\ \&\ \mathcal{C}(\boldsymbol{v}, \boldsymbol{q}, \boldsymbol{w}, \boldsymbol{t})$ *a polymorphic RAJA method type. We say that* T *is an instance of* ϕ*, written: "*T $\mathsf{instanceof}$ ϕ*" iff there exists a valuation* π *with* $\pi \models \mathcal{C}$ *such that* $\pi(v_i) = s_i$ *for* $i \in \{0, \ldots, n+1\}$ *and* $\pi(q_j) = m_j$ *for* $j \in \{1, 2\}$.

We define trivial polymorphic RAJA method types with no constraints for a given class C and method m, by: $\top^{(C,m)} = \forall \boldsymbol{v}, \boldsymbol{q}\,.\, v_0; \mathfrak{v} \xrightarrow{q_1/q_2} v_{n+1}\ \&\ \mathsf{tt}$.

Definition 7 (Subtyping of monomorphic method types).
If $T = \boldsymbol{r} \xrightarrow{n_1/n_2} r_{n+1}$ *and* $T' = \boldsymbol{s} \xrightarrow{m_1/m_2} s_{n+1}$ *then* $T <: T'$ *is defined as* $n_1 \leq m_1$ *and* $n_2 \geq m_2$ *and* $r_0 = s_0$ *and* $s_i \sqsubseteq r_i$ *for* $i = 1, \ldots, n$ *and* $r_{n+1} \sqsubseteq s_{n+1}$.

Definition 8 (Subtyping of polymorphic method types).
Let $\mathscr{C} \models C <: D$ *and let* ϕ *and* ψ *be polymorphic RAJA method types refining a FJEU method type of method* m *in class* C *and* D*, respectively. Then* $\phi <: \psi$ *iff:* $\forall\, T'$ *with* T' $\mathsf{instanceof}$ $\psi\,.\, \exists\, T$ *with* T $\mathsf{instanceof}$ ϕ *such that* $T <: T'$.

We call a polymorphic RAJA method type *empty* if its constraints are unsatisfiable and *nonempty* if they can be satisfied.

Definition 9 (RAJA program).
A RAJA program *is an annotation of an FJEU program* $\mathcal{P} = (\mathscr{C}, \mathsf{main})$ *in the form of a tuple* $\mathscr{R} = (\mathscr{C}, \mathsf{main}, \mathsf{M})$ *where* M *assigns to each class* C *and method* $m \in \mathsf{Meth}(C)$ *with* n *arguments an* n*-ary polymorphic RAJA method type* $\mathsf{M}(C, m)$.

3.1 Typing RAJA

The RAJA-typing judgement is formally defined by the rules in Figure 3. The type system allows us to derive assertions of the form $\mathsf{M}; \Xi; \Gamma \vdash^{n}_{n'} e : C^r$ where e is an expression or program phrase, C is an FJEU class, r is a view (so C^r is a refined type). Moreover, Ξ is a map from classes and methods to monomorphic RAJA method types. Finally n, n' are non-negative 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 typing in Γ.

We present here a syntax-directed version of the original typing system from [7], which contains the following rule ($\Diamond Share$) to ensure that a variable can be used more than once without duplication of potential.

$$\frac{\curlyvee(s \,|s_1, \ldots, s_j\,) \qquad \Gamma, \boldsymbol{y}{:}\boldsymbol{D^s} \vdash^{n}_{n'} e : C^r}{\Gamma, x{:}D^s \vdash^{n}_{n'} e[x/y_1, \ldots, x/y_j] : C^r} \; (\Diamond Share)$$

Here we integrate the rule ($\Diamond Share$) into the rule ($\Diamond Let$) using the fact that $\curlyvee(r \,|s_1, s_2\,)$ is equivalent to $r \sqsubseteq s_1 \oplus s_2$. This result has been omitted in this paper for lack of space; details can be found in [16]. We do not integrate ($\Diamond Share$) in other rules such as ($\Diamond Invocation$) or ($\Diamond Update$) because, for simplicity, we require that in those expressions a variable appears only once.

Monomorphic vs. polymorphic recursion. In type systems with polymorphic types and recursion, polymorphic recursion is possible. Polymorphic recursion means that, in recursive calls, any instance of the polymorphic type can be used, whereas in monomorphic recursion only one instance can be used: the same instance that the polymorphic type is being type-checked with.

Here we allow only monomorphic recursion. The reason for not treating polymorphic recursion is that type inference in the presence of polymorphic recursion is difficult, in particular we would need to compute a fixpoint when generating constraints for recursive functions. We decided to develop a simpler type inference algorithm, that does not require a fixpoint computation, because we did not find useful examples where the polymorphic recursion is required.

We need to distinguish between recursive and non-recursive method calls. With non-recursive methods calls, we can use any instance of the polymorphic type of the called method. That is why there are two rules for method invocation: ($\Diamond PInv.$) for polymorphic method invocation and ($\Diamond MInv.$) for monomorphic method invocation. In the rule ($\Diamond PInv.$) we assume that the called method is not mutually recursive with the method we are currently analysing, and consequently, we can use any instance of its polymorphic type. On the other

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

$$\frac{\forall a \in \mathsf{A}(D)\,.\,\mathsf{A}^{\mathrm{set}}(D^r,a) \sqsubseteq \mathsf{A}^{\mathrm{get}}(D^r,a) \quad D <: C \quad n \geq \Diamond(D^r)+1 \quad n' \leq n - \Diamond(D^r) - 1}{\mathsf{M};\Xi;\Gamma \vdash^{n}_{n'} \mathsf{new}\ D : C^r}$$

$$\frac{n' \leq n + \min\{\Diamond(D^r) \mid D <: C\} + 1}{\mathsf{M};\Xi;\Gamma, x{:}C^r \vdash^{n}_{n'} \mathsf{free}\,(x) : E^s}\ (\Diamond Free) \qquad \frac{D <: E \quad D^r <: C^s \quad n' \leq n}{\mathsf{M};\Gamma, x{:}E^r \vdash^{n}_{n'} (D)x : C^s}\ (\Diamond Cast)$$

$$\frac{n' \leq n}{\mathsf{M};\Xi;\Gamma \vdash^{n}_{n'} \mathsf{null} : C^s}\ (\Diamond Null) \qquad \frac{E^r <: C^s \quad n' \leq n}{\mathsf{M};\Xi;\Gamma, x{:}E^r \vdash^{n}_{n'} x : C^s}\ (\Diamond Var)$$

$$\frac{\forall F <: C\,.\,\mathsf{A}^{\mathrm{get}}(F^r,a) \sqsubseteq s \quad C.a <: D \quad n' \leq n}{\mathsf{M};\Xi;\Gamma, x{:}C^r \vdash^{n}_{n'} x.a : D^s}\ (\Diamond Access)$$

$$\frac{\forall G <: E\,.\,s \sqsubseteq \mathsf{A}^{\mathrm{set}}(G^r,a) \quad F <: E.a \quad E^r <: C^q \quad n' \leq n}{\mathsf{M};\Xi;\Gamma, x{:}E^r, y{:}F^s \vdash^{n}_{n'} x.a \leftarrow y : C^q}\ (\Diamond Upd.)$$

$$\frac{x \in \Gamma \quad \mathsf{M};\Xi;\Gamma \vdash^{n}_{n'} e_1 : C^r \quad \mathsf{M};\Xi;\Gamma \vdash^{n}_{n'} e_2 : C^r}{\mathsf{M};\Xi;\Gamma \vdash^{n}_{n'} \mathsf{if}\ x\ \mathsf{instanceof}\ E\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : C^r}\ (\Diamond Cond.)$$

$$\frac{\mathsf{M};\Xi;\boldsymbol{y}{:}\boldsymbol{F^p} \vdash^{n}_{n'} e_1 : D^s \quad \mathsf{M};\Xi;\boldsymbol{y}{:}\boldsymbol{F^q}, x{:}D^s \vdash^{n'}_{n''} e_2 : C^r \quad r_i \sqsubseteq p_i \oplus q_i}{\mathsf{M};\Xi;\boldsymbol{y}{:}\boldsymbol{F^r} \vdash^{n}_{n''} \mathsf{let}\ D\,x = e_1\ \mathsf{in}\ e_2 : C^r}\ (\Diamond Let)$$

$$\frac{\begin{array}{c}\left(G^{s_0};\boldsymbol{E^s} \xrightarrow{t/t'} H^{s'}\right)\ \mathsf{instanceof}\ \mathsf{M}(G\,,m) \\ G^{r_0} <: G^{s_0} \qquad F_i^{r_i} <: E_i^{s_i} \quad H^{s'} <: C^{r'} \quad n \geq t \quad n' \leq t' + n - t\end{array}}{\mathsf{M};\Xi;\Gamma, x{:}G^{r_0}, \boldsymbol{y}{:}\boldsymbol{F^r} \vdash^{n}_{n'} x.m\,(y_1,\ldots,y_j) : C^{r'}}\ (\Diamond PInv.)$$

$$\frac{\begin{array}{c}\left(G^{s_0};\boldsymbol{E^s} \xrightarrow{t/t'} H^{s'}\right) \in \Xi(G\,,m) \\ G^{r_0} <: G^{s_0} \qquad F_i^{r_i} <: E_i^{s_i} \quad H^{s'} <: C^{r'} \quad n \geq t \quad n' \leq t' + n - t\end{array}}{\mathsf{M};\Xi;\ x{:}G^{r_0}, \boldsymbol{y}{:}\boldsymbol{F^r} \vdash^{n}_{n'} x.m\,(y_1,\ldots,y_j) : C^{r'}}\ (\Diamond MInv.)$$

RAJA Method Typing $\boxed{\vdash_{\mathsf{m}} \mathsf{M}\ \mathsf{ok}}$

$$\frac{\begin{array}{c}\vdash_{\mathsf{m}} \mathsf{M}'\ \mathsf{ok} \quad \mathrm{dom}(\Xi) = \mathrm{dom}(\mathsf{M}'') \quad \forall\,(C,m) \in \mathsf{M}'' \quad \Xi(C,m) = T \\ \forall\,T = (C^{r_0};\boldsymbol{E^{\mathbf{t}}} \xrightarrow{n/n'} H^{r_{n+1}})\ \mathsf{instanceof}\ \mathsf{M}''(C,m) \qquad (r_0 \dot{-} \mathsf{p})_C \sqsubseteq s_0 \\ \mathsf{M}';\Xi;\mathtt{this}{:}C^{s_0},\ x_1{:}E_1^{s_1},\ldots,x_j{:}E_j^{s_j} \vdash^{n+p}_{n'} e : H^{r_{j+1}} \quad r_i \sqsubseteq s_i \qquad \Diamond(C^{r_0}) \geq p\end{array}}{\vdash_{\mathsf{m}} \mathsf{M}' \uplus \mathsf{M}''\ \mathsf{ok}}$$

Fig. 3. RAJA Typing.

hand, we apply the rule ($\Diamond MInv.$) when the called method appears in the map $\Xi : \forall C \in \mathscr{C} \,.\, \mathsf{Meth}(C) \to \mathsf{MonoType}$ which means that this method and the method whose body we are analysing are mutually recursive.

The judgement for typing the body of a method ($\vdash_{\mathsf{m}} \mathsf{M}\ \mathsf{ok}$ of Fig. 3) shall mean that all the methods in the domain of the map M are well-typed.

Also notice in the judgement $\vdash_{\mathsf{m}} \mathsf{M}\ \mathsf{ok}$ the number p. It represents the amount of items of potential that we take from the potential of the refined type of this in the type T for using in the method's body. Thus, we need to check that the potential of the refined type of this is at least p.

Definition 10 (Well-typed RAJA-program).
A RAJA-program $\mathscr{R} = (\mathscr{C}, \mathsf{main}, \mathsf{M})$ *is* well-typed *if the following conditions are satisfied:*

1. $\vdash_{\mathsf{m}} \mathsf{M}\ \mathsf{ok}$
2. $\forall C \in \mathscr{C}, m \in \mathsf{Meth}(C) \,.\, \mathsf{M}(C, m)$ *is nonempty.*
3. $\forall C, D \in \mathscr{C}$ *with* $S(C) = D \Rightarrow \mathsf{M}(C, m) <: \mathsf{M}(D, m)$.

A full soundness proof for this system is given in [16]. It consists of a small modification of the soundness proof for the original RAJA system [7].

4 Type inference for RAJA

4.1 Constraint generation

In the following we present rules for generating subtyping and arithmetic constraints from FJEU programs. The rules (Fig. 4) describe a constraint generation judgement $\mathsf{M}; \Xi; \Gamma \vdash^{p}_{p'} e : C^v \ \&\ \mathcal{C}$ where e is an expression, Γ maps variables to FJEU types refined with view variables, C^v is an FJEU type refined with a view variable, p and p' are arithmetic variables and $\mathcal{C}$ is a conjunction of subtyping and arithmetic constraints. Further, Ξ is a map from classes and methods with n arguments to $n + 2$ view variables and two arithmetic variables.

We write $\pi(\Xi)$ to mean the map from classes and methods to monomorphic RAJA method types that is obtained after substituting every view and arithmetic variable in Ξ with its value in the valuation π. Similarly, $\pi(\Gamma)$ means the context that we obtain after substituting the view variables in Γ with their values in π. In addition, we use the notations $|\Xi|$ and $|\Gamma|$ for meaning the following. If Ξ is a map from classes and method names to monomorphic RAJA types, then $|\Xi|$ denotes a map from classes and method names to view and arithmetic variables with $\mathrm{dom}(|\Xi|) = \mathrm{dom}(\Xi)$. Similarly, if Γ is an FJEU context, then $|\Gamma|$ is a context from program variables to FJEU types refined with view variables with the same domain as Γ. The judgement reads: expression e has type C^v in the context Γ, subject to the constraints $\mathcal{C}$. Moreover, the judgement defines a *total* function generateConstraints that generates constraints for an expression:

$$\mathsf{generateConstraints}(\mathsf{M}, \Xi, \Gamma, p, p', e, C^v) = \mathcal{C} \text{ if } \mathsf{M}; \Xi; \Gamma \vdash^{p}_{p'} e : C^v \ \&\ \mathcal{C}$$

The subtyping constraints are of the form $C^v <: D^u$ where C and D are classes and v and u are view variables. We also create constraints of the form $u \sqsubseteq v \oplus w$ in the rule ($\vdash Let$), where $v \oplus w$ is a view expression.

$$\boxed{\mathsf{M};\Xi;\Gamma \overset{p}{\underset{p'}{\vdash}} e : C^v \;\&\; \mathcal{C}}$$

$$\frac{\mathcal{C} = (E^v <: C^u \;\wedge\; p' \leq p)}{\mathsf{M};\Xi;\Gamma, x : E^v \overset{p}{\underset{p'}{\vdash}} x : C^u \;\&\; \mathcal{C}} \qquad \frac{\mathcal{C} = (p' \leq p)}{\mathsf{M};\Xi;\Gamma \overset{p}{\underset{p'}{\vdash}} \mathsf{null} : C^v \;\&\; \mathcal{C}} \qquad \frac{\mathcal{C} = \bigwedge_{D<:C} p' \leq p + \Diamond(D^v) + 1}{\mathsf{M};\Xi;\Gamma, x : C^v \overset{p}{\underset{p'}{\vdash}} \mathsf{free}\,(x) : E^u \;\&\; \mathcal{C}}$$

$$\frac{\mathcal{E} = D^v <: C^v \qquad \mathcal{AC} = p \geq \Diamond(D^v) + 1 \;\wedge\; p' \leq p - \Diamond(D^v) - 1}{\mathsf{M};\Xi;\Gamma \overset{p}{\underset{p'}{\vdash}} \mathsf{new}\; D : C^v \;\&\; \mathcal{E} \;\wedge\; \mathcal{AC} \;\wedge\; \bigwedge_{a \in \mathcal{A}(D)} \mathsf{A}^{\mathrm{set}}(D^v, a) \sqsubseteq \mathsf{A}^{\mathrm{get}}(D^v, a)} \;(\vdash New)$$

$$\frac{\mathcal{C} = (D^v <: E^v \;\wedge\; D^v <: C^u \;\wedge\; p' \leq p)}{\mathsf{M};\Xi;\Gamma, x : E^v \overset{p}{\underset{p'}{\vdash}} (D)\, x : C^u \;\&\; \mathcal{C}} \;(\vdash Cast) \qquad \frac{\mathcal{C} = \bigwedge_{E<:C} (C.a)^{\mathsf{A}^{\mathrm{set}}(E^v, a)} <: D^u \;\wedge\; p' \leq p)}{\mathsf{M};\Xi;\Gamma, x : C^v \overset{p}{\underset{p'}{\vdash}} x.a : D^u \;\&\; \mathcal{C}}$$

$$\frac{\mathcal{C} = (\bigwedge_{E<:C} F^w <: (C.a)^{\mathsf{A}^{\mathrm{set}}(E^v, a)} \;\wedge\; C^v <: D^u)}{\mathsf{M};\Xi;\Gamma, x : C^v, y : F^w \overset{p}{\underset{p'}{\vdash}} x.a \leftarrow y : D^u \;\&\; \mathcal{C} \;\wedge\; p' \leq p} \;(\vdash Update)$$

$$\frac{\mathsf{M};\Xi;\Gamma \overset{p}{\underset{p'}{\vdash}} e_1 : C^v \;\&\; \mathcal{C}_1 \qquad \mathsf{M};\Xi;\Gamma \overset{p}{\underset{p'}{\vdash}} e_2 : C^v \;\&\; \mathcal{C}_2 \qquad \mathcal{C} = (\mathcal{C}_1 \;\wedge\; \mathcal{C}_2)}{\mathsf{M};\Xi;\Gamma \overset{p}{\underset{p'}{\vdash}} \mathsf{if}\; x \;\mathsf{instanceof}\; E \;\mathsf{then}\; e_1 \;\mathsf{else}\; e_2 : C^v \;\&\; \mathcal{C}} \;(\vdash Cond.)$$

$$\frac{\mathsf{M};\Xi;\boldsymbol{y} : \boldsymbol{F^v} \overset{p}{\underset{p'}{\vdash}} e_1 : D^u \;\&\; \mathcal{C}_1 \qquad \mathsf{M};\Xi;\boldsymbol{y} : \boldsymbol{F^w}, x : D^u \overset{p'}{\underset{p''}{\vdash}} e_2 : C^v \;\&\; \mathcal{C}_2}{\mathsf{M};\Xi;\boldsymbol{y} : \boldsymbol{F^u} \overset{p}{\underset{p''}{\vdash}} \mathsf{let}\; D\, x = e_1 \;\mathsf{in}\; e_2 : C^v \;\&\; (\mathcal{C}_1 \;\wedge\; \mathcal{C}_2 \;\wedge\; \bigwedge_i u_i \sqsubseteq v_i \oplus w_i)} \;(\vdash Let)$$

$$\frac{\begin{array}{c}\Xi(G, m) = G^{v_0}; \boldsymbol{E^v} \xrightarrow{q_1/q_2} H^{v_{n+1}} \\ \mathcal{TC} = G^u <: G^{v_0} \;\wedge\; F_i^{u_i} <: E_i^{v_i} \;\wedge\; H^{v_{n+1}} <: C^{u'} \;\wedge\; p \geq q_1 \;\wedge\; p' \leq q_2 + p - q_1\end{array}}{\mathsf{M};\Xi;\Gamma, x : G^u, \boldsymbol{y} : \boldsymbol{F^u} \overset{p}{\underset{p'}{\vdash}} x.m\,(y_1, \ldots, y_j) : C^{u'} \;\&\; \mathcal{TC}} \;(\vdash MInv)$$

$$\frac{\begin{array}{c}\mathsf{M}(G, m) = \forall \boldsymbol{v}, \boldsymbol{q}\, \exists \boldsymbol{v'}, \boldsymbol{q'}\, .\, G^{v_0}; \boldsymbol{E^v} \xrightarrow{q_1/q_2} H^{v_{n+1}} \;\&\; \mathcal{D} \quad \mathcal{D}' = \mathcal{D}[\boldsymbol{w}/\boldsymbol{v}, \boldsymbol{w'}/\boldsymbol{v'}, \boldsymbol{t}/\boldsymbol{q}, \boldsymbol{t'}/\boldsymbol{q'}] \\ \mathcal{TC} = G^u <: G^{w_0} \;\wedge\; F_i^{u_i} <: E_i^{w_i} \;\wedge\; H^{w_{n+1}} <: C^{u'} \;\wedge\; p \geq t_1 \;\wedge\; p' \leq t_2 + p - t_1\end{array}}{\mathsf{M};\Xi;\Gamma, x : G^u, \boldsymbol{y} : \boldsymbol{F^u} \overset{p}{\underset{p'}{\vdash}} x.m\,(y_1, \ldots, y_j) : C^{u'} \;\&\; \mathcal{TC} \;\wedge\; \mathcal{D}'} \;(\vdash PInv)$$

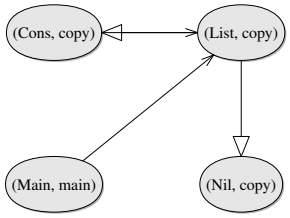
$$\boxed{\vdash_{\mathsf{mc}} \mathsf{M}\ \mathsf{ok}}$$

$$\frac{\begin{array}{c}\vdash_{\mathsf{mc}} \mathsf{M}'\ \mathsf{ok} \qquad \forall i = 1\,..\,k \quad (C_i, m_i) \in \mathrm{dom}(\mathsf{M}'') \qquad \Xi(C_i, m_i) = C_i^{v_0}; \boldsymbol{E^v} \xrightarrow{p_1/p_2} H^{v_{n+1}} \\ \mathsf{M}';\Xi;\mathtt{this} : C_i^{\bar{v}_0},\ \boldsymbol{x} : \boldsymbol{E^v} \overset{\bar{p}_1}{\underset{p_2}{\vdash}} \mathsf{M}_{\mathsf{body}}(C, m) : H^{v_{n+1}} \;\&\; \mathcal{C}^{(i)} \\ \psi^{(i)} = \forall \boldsymbol{v}, \boldsymbol{p}\,.\, C^{v_0}; \boldsymbol{E^v} \xrightarrow{p_1/p_2} H^{v_{n+1}} \;\&\; (\mathcal{C}^{(i)} \;\wedge\; v_0 \sqsubseteq \bar{v}_0 \;\wedge\; \Diamond\big(C_i^{v_0}\big) + p_1 \geq \Diamond\Big(C_i^{\bar{v}_0}\Big) + \bar{p}_1) \\ \mathsf{S}(D_j) = C_i \quad \lambda_j = \begin{cases} \mathsf{M}'(D_j, m_i) & \text{if } (D_j, m_i) \in \mathrm{dom}(\mathsf{M}') \\ \top^{(D_j, m_i)} & \text{if } (D_j, m_i) \in \mathrm{dom}(\mathsf{M}'') \end{cases} \qquad \phi^{(i)} = \psi^{(i)} \vee \bigvee_j \lambda_j \\ \mathsf{M}''(C_i, m_i) = \forall \boldsymbol{v}, \boldsymbol{p}\,.\, C_i^{v_0}; \boldsymbol{E^v} \xrightarrow{p_1/p_2} H^{v_{n+1}} \;\&\; \mathcal{D}^{(i)} \qquad \mathcal{D}^{(i)} = \bigwedge_{l \in \{1, \ldots, k\}} \mathsf{constr}(\phi^{(l)})\end{array}}{\vdash_{\mathsf{mc}} \mathsf{M}' \uplus \mathsf{M}''\ \mathsf{ok}}$$

Fig. 4. Generation of RAJA polymorphic types.

There are two rules for method invocation: ($\vdash PInv$) for polymorphic method invocation and ($\vdash MInv$) for monomorphic method invocation. In ($\vdash PInv$) we assume that the called method has already been analysed and so its polymorphic RAJA method type is available. The constraints generated by this rule consist of the method's constraints, where we substitute the view and arithmetic variables with fresh ones, in conjunction with standard subtyping and arithmetic constraints. We apply the rule ($\Diamond MInv.$) when the called method appears in the map Ξ, which means, as we discussed earlier, that the method and the method whose body we are analysing are mutually recursive. In that case the constraints for the method are not yet available. Thus, we only generate the standard subtyping and arithmetic constraints.

The judgement $\vdash_{\mathsf{mc}} \mathsf{M}\ \mathsf{ok}$ returns RAJA polymorphic method types for the methods in M by generating the constraints for the methods' bodies. We perform the analysis on the basis of the call graph of the program, which we modify slightly by adding the inheritance relations to it. For example, the graph corresponding to the program for copying lists defined in Fig. 1, can be represented as follows:



After we have built the graph, we decompose it in its strongly connected components to obtain the acyclic component graph G^{SCC}. Afterwards, we sort the obtained dag G^{SCC} topologically and call the constraint generation algorithm in that order, with the strongly connected components being analysed together. When applied to the graph above, we obtain the following order where $(\,\mathsf{Cons}, \mathsf{copy})$ and $(\mathsf{List}, \mathsf{copy})$ are analysed together.

$$(\,\mathsf{Nil}, \mathsf{copy}), [(\,\mathsf{Cons}, \mathsf{copy}), (\mathsf{List}, \mathsf{copy})], (\mathsf{Main}, \mathsf{main})$$

Now, why do we need to extend the call graph with inheritance relations? The reason for this is that, before we analyse a method m in a class C, we would like to analyse the same method m in each subclass D of C. For proving soundness of the constraint generation algorithm we need to show $\mathsf{M}(D, m) <: \mathsf{M}(C, m)$, and this follows trivially when we add the constraints of $D.m$ to the polymorphic type of $C.m$. For example, the method $\mathsf{List.copy}$ should contain the constraints of the methods $\mathsf{Cons.copy}$ and $\mathsf{Nil.copy}$, as explained earlier.

In the following we prove that, if the constraints generated for the expression e are satisfiable, then the expression is typeable in the RAJA system with the result type, context and effect given by the solution to the constraints.

Lemma 1 (Soundness of constraint generation).
If $\mathcal{E} :: \mathsf{M}; \Xi; \Gamma \vdash^{q_1}_{q_2} e : C^v \ \&\ \mathcal{C}$ *and* $\pi \models \mathcal{C}$ *then* $\mathsf{M}; \pi(\Xi); \pi(\Gamma) \vdash^{\pi(q_1)}_{\pi(q_2)} e : C^{\pi(v)}$.

Proof. By induction on $\mathcal{E}$.

Lemma 2 (Soundness of constraint generation for methods).
Let $\mathcal{P} = (\mathscr{C}, \mathsf{main})$ *be an FJEU program and let* $\mathcal{E} :: \vdash_{\mathsf{mc}} \mathsf{M}\ \mathsf{ok}$ *and let* $\mathsf{M}(C, m)$ *be non-empty for each* $(C, m) \in \mathrm{dom}(\mathsf{M})$. *Then:*

1. $\vdash_{\mathsf{m}} \mathsf{M}\ \mathsf{ok}$.
2. $\mathsf{S}(D) = C$ *implies* $\mathsf{M}(D, m) <: \mathsf{M}(C, m)$.

Proof. 1. By induction on $\mathcal{E}$.
2. Follows by the design of the judgement $\vdash_{\mathsf{mc}} \mathsf{M}\ \mathsf{ok}$, as discussed earlier.

Next, we show that, when applied to a typeable expression, the constraint generation rules emit a satisfiable constraint set.

Lemma 3 (Completeness of constraint generation).
If $\mathcal{E} :: \mathsf{M}; \Xi; \Gamma \vdash^{n_1}_{n_2} e : C^r$ *and* $\mathsf{M}; |\Xi|; |\Gamma| \vdash^{p_1}_{p_2} e : C^v$ *&* $\mathcal{C}$ *then there exists* π *with* $\pi(p_i) = n_i$, $\pi(v) = r$, $\pi(|\Gamma|) = \Gamma$, $\pi(|\Xi|) = \Xi$ *such that* $\pi \models \mathcal{C}$.

Proof. By induction on $\mathcal{E}$.

Lemma 4 (Completeness of constraint generation for methods).
Let $\mathscr{R} = (\mathscr{C}, \mathsf{main}, \mathsf{M})$ *be a well-typed RAJA program and let* $\mathcal{N}$ *be a map with* $\mathrm{dom}(\mathcal{N}) = \mathrm{dom}(\mathsf{M})$ *and* $\mathcal{E} :: \vdash_{\mathsf{mc}} \mathcal{N}\ \mathsf{ok}$. *Then for all* $(C, m) \in \mathsf{M}$ *holds* $\mathcal{N}(C, m) <: \mathsf{M}(C, m)$.

Proof. By induction on $\mathcal{E}$.

4.2 Constraint solving

In this section we shall see how to apply RAJA types to the analysis of the heap-space requirements of methods. Currently, our tool is not capable of computing bounds for arbitrary methods, but only for the method main. This is because translating refined types to closed-form upper bounds is challenging and requires further research. We wish to compute an upper bound on the number of heap cells needed for executing main as a function of main's arguments, which follows from the potential given to its arguments by its RAJA type. We have seen in the previous section how to obtain a polymorphic RAJA type for main, but, for being able to read off the potential from that type, we need a concrete instance of the type, which we can obtain by solving the type's constraints.

Whereas solving linear arithmetic constraints is easily achieved by an LP-Solver, solving subtyping constraints is more challenging. The task of solving a constraint $C^u <: D^v$ can be reduced to the tasks of solving $C <: D$ and $u \sqsubseteq v$, by the definition of subtyping. $C <: D$ can be checked easily by analysing the inheritance relations in the program. Thus, the real challenge is solving $u \sqsubseteq v$. Solving these kind of constraints is difficult for various reasons.

First, views are infinite objects, and the inequality relation over views is defined coinductively. Thus, unfolding the definition of inequality; that is, trying to solve the constraints $\Diamond(C^u) \geq \Diamond(C^v)$ for each $C \in \mathcal{C}$ and $\mathsf{A}^{\mathrm{get}}(C^u, a) \sqsubseteq \mathsf{A}^{\mathrm{get}}(C^v, a)$ and $\mathsf{A}^{\mathrm{set}}(C^v, a) \sqsubseteq \mathsf{A}^{\mathrm{set}}(C^u, a)$ for each $a \in \mathsf{A}(C)$ would lead to more unfolding steps and this process would not terminate.

Second, subtyping over views is covariant in the get views and contravariant in the set views. The contravariance also brings difficulties. For this reason, we studied in previous work [14] a simpler type of infinite trees than views. We fix a finite set $\mathcal{L} = \{l_1, \ldots, l_n\}$ of labels to address the children of a node, e.g. $\mathcal{L} = \{L, R\}$ for infinite binary trees and $\mathcal{L} = \{tl\}$ for infinite lists. Such trees can be added, scaled, and compared componentwise; furthermore, we have an operation $\Diamond(.)$ that extracts the root label of a tree, thus if t is a tree then $\Diamond(t)$ is an element of $\mathbb{D}$. Finally, if t is a tree and $l \in \mathcal{L}$ then $l(t)$ is the l-labelled immediate subtree of t. We define a preorder $\sqsubseteq$ between trees as follows:

Definition 11. *Let $t, t' \in \mathsf{T}^{\mathcal{L}}_{\mathbb{D}}$. We define $t \sqsubseteq t'$ coinductively by $t \sqsubseteq t' \iff \Diamond(t) \leq \Diamond(t')$ and $l_i(t) \sqsubseteq l_i(t')$ for all $l_i \in \mathcal{L}$.*

This inequality relation is covariant in all cases. Because these trees are simpler objects, solving constraints over them is simpler than solving constraints over views. Thus, we solve the inequalities over views by reducing them to inequalities over infinite trees. For solving constraints over infinite trees, we still have the problem that unfolding the inequality relation would not terminate. This is why, to ensure termination of unfolding, we developed a heuristic algorithm for solving these constraints that assumes that the solutions to the constraints are regular infinite trees. This implies, however, that the algorithm is not able to solve all the constraints but only a subset of them that admit regular solutions. Therefore, when using this algorithm, we can solve only subtyping constraints that admit regular views as a solution, which correspond to programs whose heap-space consumption is a linear function of its input. Hence, the algorithm that we present in this paper can compute only linear bounds on the heap-space requirements of programs. However, we remark that this is because no better algorithm for solving the constraints over infinite trees is known at the moment.

We present here a reduction from views to infinite trees. The idea of the reduction is to separate the "positive parts" and "negative parts" of a view, to build infinite trees. To reduce a view $r \in \mathscr{V}$, we define, for each class $C_i \in \mathscr{C}$, the infinite trees $r_i^+, r_i^- \in \mathsf{T}^{\mathcal{L}}_{\mathbb{D}}$, where $\mathcal{L} = \mathcal{L}^+ \cup \mathcal{L}^-$ and $L^+ = \{l^+_{kj} \mid C_k \in \mathscr{C}, a_j \in \mathsf{A}(C_k)\}$ and $L^- = \{l^-_{kj} \mid C_k \in \mathscr{C}, a_j \in \mathsf{A}(C_k)\}$ such that we can reduce inequalities between views to inequalities between infinite trees. More exactly, we want to prove: $r \sqsubseteq s \Rightarrow \bigwedge_{C_i \in \mathscr{C}} s_i^+ \sqsubseteq r_i^+ \ \wedge \ r_i^- \sqsubseteq s_i^-$. Fig. 5 shows a representation of the reduction applied to the view rich.

Definition 12. *Let $r \in \mathscr{V}$. We define $\mathsf{expand}(r) = (\boldsymbol{r}^+, \boldsymbol{r}^-)$, where r_i^+ and r_i^- are defined coinductively as follows. Let $C_i \in \mathscr{C}$ and $C_k \in \mathscr{C}$ and $a_j \in \mathsf{A}(C_k)$.*

$$\begin{array}{ll} \Diamond(r_i^+) = \Diamond(C_i^r) & \Diamond(r_i^-) = 0 \\ l^+_{kj}(r_i^+) = \mathsf{A}^{\mathrm{get}}(C_k^r, a_j)_i^+ & l^+_{kj}(r_i^-) = \mathsf{A}^{\mathrm{get}}(C_k^r, a_j)_i^- \\ l^-_{kj}(r_i^+) = \mathsf{A}^{\mathrm{set}}(C_k^r, a_j)_i^- & l^-_{kj}(r_i^-) = \mathsf{A}^{\mathrm{set}}(C_k^r, a_j)_i^+ \end{array}$$

Lemma 5. *Let $r \sqsubseteq s$ and let $(\boldsymbol{r}^+, \boldsymbol{r}^-) = \mathsf{expand}(r)$ and $(\boldsymbol{s}^+, \boldsymbol{s}^-) = \mathsf{expand}(s)$.*

1. *$s_i^+ \sqsubseteq r_i^+$ for all i.*

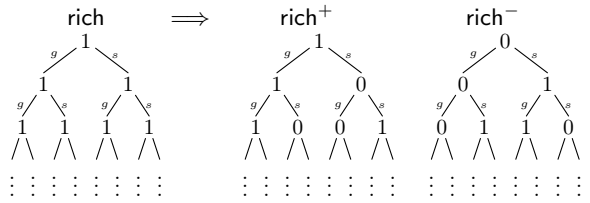


Fig. 5. View rich reduced to rich^+ and rich^-, assuming that $\mathcal{C} = \{\,\mathsf{Cons}\}$.

2. $r_i^- \sqsubseteq s_i^-$ *for all* i.

Proof. Simultaneously by coinduction.

Next, we build a view from two vectors of trees $\boldsymbol{t}$ and $\boldsymbol{t'}$, with $|\boldsymbol{t}| = |\boldsymbol{t'}| = |\mathscr{C}|$.

Definition 13. *We define the function* $\mathsf{reduce}(\boldsymbol{t}, \boldsymbol{t'}) = r$*, where* $r \in \mathscr{V}$*, coinductively as follows.*

$$\begin{aligned} \langle\!\langle C_i^r \rangle\!\rangle &= \langle\!\langle t_i \rangle\!\rangle \dot{-} \langle\!\langle t_i' \rangle\!\rangle \\ \mathsf{A}^{\mathrm{get}}(C_k^r, a_j) &= \mathsf{reduce}(\overrightarrow{l_{kj}^+(t_i)}, \overrightarrow{l_{kj}^+(t_i')}) \\ \mathsf{A}^{\mathrm{set}}(C_k^r, a_j) &= \mathsf{reduce}(\overrightarrow{l_{kj}^-(t_i')}, \overrightarrow{l_{kj}^-(t_i)}) \end{aligned}$$

First, we notice that reduce is the left inverse of expand.

Lemma 6. *Let* $r \in \mathscr{V}$*. Then* $\mathsf{reduce}(\mathsf{expand}(r)) = r$.

Proof. By coinduction.

Lemma 7. *Let* $\boldsymbol{t}$*,* $\boldsymbol{t'}$*,* $\boldsymbol{p}$*,* $\boldsymbol{p'}$ *be vectors of infinite trees. Then, if* $t_i' \sqsubseteq p_i'$ *and* $p_i \sqsubseteq t_i$ *for each* i*, then* $\mathsf{reduce}(\boldsymbol{t}, \boldsymbol{t'}) \sqsubseteq \mathsf{reduce}(\boldsymbol{p}, \boldsymbol{p'})$.

Proof. By coinduction.

Now we can reduce inequalities over views to inequalities of infinite trees, based on the reduction from views to infinite trees. When we obtain a solution for the set of constraints over infinite trees, we can build a solution for the original set of inequalities over views, based on the reduction from infinite trees to views. The details are omitted for lack of space, and can be found in [16].

5 Experimental Results

We have implemented a tool in OCaml for type checking, evaluating and analysing the heap-space requirements of FJEU programs, based on the algorithm presented in this paper. The tool uses the result of the analysis for building an optimised heap for evaluating the programs; that is, it creates a heap with a size equal to the size predicted by the analysis. The tool assumes that each FJEU program contains a main method which has one parameter of type List. Further, the tool assumes that it is given an input file for the program execution. The

Program	LoC	Heap space	Run time
Copy	37	n	0.2s
CircList	56	1 + n	1.6s
InsSort	66	2 + n	1.9s
DList	70	3 + n	1.2s
Append	80	2 + n	3s

Program	LoC	Heap space	Run time
CAppend	60	2 + 2n	0.7s
MergeSort	127	1	10.3s
BankAcc	200	2 + 8n	6.3s
Bank	908	11 + 6n	9.8 min

Table 1. Experimental results. The column **Heap space** shows the prediction of the required size of the free-list which in each case was equal to the actual heap-space requirements of the program and **Run time** represents the run time of the analysis.

interpreter then creates a singly linked list (one node for each row of the input file) and saves it in the heap before it starts executing the program.

The analyser component of the tool can analyse methods whose heap-space consumption is a linear function on the size of its arguments. When it analyses the main method of a program, it delivers two non-negative real numbers a and b, which shall mean that the program can be evaluated with no memory errors with a heap of size at least $a \cdot |\text{input file}| + b$.

Table 1 shows some programs that we could analyse with our tool. For each example, we could solve the constraints and resultantly provide a (linear) upper bound for its heap-space requirements. The experiments were performed on a 2.20GHz Intel(R) Core(TM)2 Duo CPU laptop with 2GB RAM. The run-time of the analysis varied from 0.2s to about 10 minutes on a program of 908 LoC. Being able to analyse 908 LoC may seem rather modest, but one should note that nearly all these LoC contribute to the analysis. A typical real program will contain large portions that look like white space to the analysis, e.g. numerical computations, I/O, etc. and resource-wise independent parts of a larger program could be analysed separately.

All bounds are exact in the above experiments, although our soundness result only ensures an upper bound. There is a demo website where the examples can be analysed and downloaded, and the user can perform the analysis on their own programs[3].

6 Conclusions and related work

We have presented a type-based analysis of the heap-space requirements of object-oriented programs. The soundness of each step of the analysis has been rigorously proved. Moreover, the analysis was modular, enabled by the use of polymorphic types. Thus, in principle, the analysis is capable of scaling to large programs, although there is plenty of room for improvement. Polymorphic types also enable an incremental analysis because they can be saved after the analysis, and in most cases they do not need to be re-generated when more classes and methods are added to the programs.

[3] http://raja.tcs.ifi.lmu.de

Related work. Constraint-based type reconstruction for numerically refinement types has been introduced in [17] and been further developed in [18] under the name "liquid types" which further introduces techniques from predicate abstraction and model checking.

Our method for the generation of view constraints is directly inspired by those works, in particular [17]; however, the view constraints thus gleaned are not directly amenable to algorithmic solution. The main conceptual contribution of the present work is thus not so much the RAJA type system which has already been presented elsewhere, albeit in slightly different and less general form, but rather the solution of the generated typing or view constraints by translation to tree constraints and iterated elimination.

Looking at the black-box front end of our contribution—the fully automatic inference of symbolic resource bounds for object-oriented programs, we can place our contribution in the following perspective. The topic has been researched intensively in the past years and many different approaches to it have been proposed.

In a series of papers [2, 13, 19] Chin and his collaborators have used sized types and separation logic for the generation of symbolic resource bounds for object-oriented programs. Presently, the method is not fully automatic because the user must provide aliasing and shape information. Furthermore, in the functional realm, the amortised approach has proved superior to related methods in the case of algorithms that heavily use intermediate data structures whose size is difficult to describe [10]. However, it might very well be possible to combine our approach with the generic system "HIP and SLEEK" [19] that maintains and propagates separation logic assertions for an object-oriented language.

In the recurrence-based approach COSTA [4, 1], one introduces an unknown resource bounding function for each method and then derives recurrence constraints for those by going over the control-flow graph. The main methodological innovation lies not so much in the analysis which uses mainly standard techniques, but in the development of improved solvers for these recurrences. As discussed above the amortised approach is superior when resource usage is intertwined with size and layout of intermediate data structures. The path-length analysis performed by COSTA to infer size-relations is sound with the condition that there is no aliasing and cyclic data, whereas the analysis performed by RAJA is sound for all programs and we do not require an extra cyclicity analysis. Nevertheless, we hope that the advanced recurrence-solving technology developed by the COSTA team could allow us to go beyond linear arithmetic constraints and bounds.

For imperative non-object-oriented programs several other fully- or semi-automatic analyses have been developed, notably SPEED [3] which is based on the inference of linear arithmetic relationships between manually added counter variables. Here, the performance in the presence of dynamically allocated data structures strongly depends on the instrumentation. SPEED also uses user-defined quantitative functions that are associated with abstract data-structures. In contrary, RAJA is fully automatic and does not require any user-input.

Atkey [20] combined amortised analysis and separation logic to analyse imperative programs. Like RAJA, Atkey's system can compute only linear bounds. On the other hand, the user needs to provide complex annotations.

Acknowledgements. We acknowledge support by the DFG Graduiertenkolleg 1480 Programm- und Modell-Analyse (PUMA).

References

1. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Closed-Form Upper Bounds in Static Cost Analysis. Journal of Automated Reasoning (2010)
2. Chin, W.N., Nguyen, H.H., Qin, S., Rinard, M.: Memory usage verification for oo programs. In: SAS. (2005)
3. Gulwani, S., Mehra, K.K., Chilimbi, T.M.: SPEED: precise and efficient static estimation of program computational complexity. In: POPL, ACM (2009)
4. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: COSTA: Design and implementation of a cost and termination analyzer for java bytecode. In: FMCO. (2007)
5. Grobauer, B.: Topics in Semantics-based Program Manipulation. PhD thesis, BRICS Aarhus (2001)
6. Gomez, G., Liu, Y.A.: Automatic time-bound analysis for a higher-order language. In: PEPM. (2002)
7. Hofmann, M., Jost, S.: Type-based amortised heap-space analysis. In: ESOP. (2006)
8. Hofmann, M., Rodriguez, D.: Efficient type-checking for amortised heap-space analysis. In: CSL. (2009)
9. Jost, S., Loid, H.W., Hammond, K., Hofmann, M.: Static determination of quantitative resource usage for higher-order programs. In: POPL. (January 2010)
10. Hoffmann, J., Aehlig, K., Hofmann, M.: Multivariate amortized resource analysis. In: POPL. (2011)
11. Tarjan, R.E.: Amortized computational complexity. SIAM Journal on Algebraic and Discrete Methods **6**(2) (April 1985) 306–318
12. Okasaki, C.: Purely Functional Data Structures. Cambridge University Press (1998)
13. He, G., Qin, S., Luo, C., Chin, W.N.: Memory usage verification using hip/sleek. In: ATVA. (2009)
14. Hofmann, M., Rodriguez, D.: Linear constraints over infinite trees. In: LPAR. (2012)
15. Igarashi, A., Pierce, B., Wadler, P.: Featherweight Java: A minimal core calculus for Java and GJ. In: OOPSLA. (1999)
16. Rodriguez, D.: Amortised Resource Analysis for Object Oriented Programs. PhD thesis, Ludwig-Maximilians-Universität München (2012)
17. Knowles, K.L., Flanagan, C.: Type reconstruction for general refinement types. In: ESOP. Volume 4421 of LNCS., Springer (2007)
18. Rondon, P.M., Kawaguci, M., Jhala, R.: Liquid types. ACM SIGPLAN Notices **43**(6) (June 2008) 159–169
19. Chin, W.N., David, C., Gherghina, C.: A hip and sleek verification system. In: OOPSLA Companion. (2011)
20. Atkey, R.: Amortised resource analysis with separation logic. Logical Methods in Computer Science **7**(2) (2011)