Efficient Type-Checking for Amortised Heap-Space Analysis

Martin Hofmann and Dulma Rodriguez

Department of Computer Science, University of Munich
Oettingenstr. 67, D-80538 München, Germany
{martin.hofmann,dulma.rodriguez}@ifi.lmu.de

Abstract. The prediction of resource consumption in programs has gained interest in the last years. It is important for a number of areas, notably embedded systems and safety critical systems. Different approaches to achieve bounded resource consumption have been analysed. One of them, based on an amortised complexity analysis, has been studied by Hofmann and Jost in 2006 for a Java-like language.

In this paper we present an extension of this type system consisting of more general subtyping and sharing relations that allows us to type more examples. Moreover we describe efficient automated type-checking for a finite, annotated version of the system. We prove soundness and completeness of the type checking algorithm and show its efficiency.

Keywords: Type systems, Resource analysis, Semantics, OOP.

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 must be costed whose size is proportional to parts of the input. 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 similar to C's `free()`. The evaluation of such programs is carried out by maintaining a set of free memory units called freelist. When an object is created, a number of heap units required to store it is taken from the freelist if it contains enough units, otherwise the program execution is aborted. Finally, each deallocated heap unit is returned to the freelist.

E. Grädel and R. Kahle (Eds.): CSL 2009, LNCS 5771, pp. 317–331, 2009.

The goal of the analysis is to predict a bound on the initial size that the freelist must have so that a given program may be executed without causing unsuccessful abortion due to insufficient memory. This has been achieved by combining amortized analysis [Tar85, Oka98] with type-based techniques in order to define potentials.

Essentially each object is ascribed an abstracted portion of the freelist, referred to as *potential*, which 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 represents an upper bound on the total heap consumption.

While type inference and automated type checking have already been developed for a functional language within the EmBounded Project ([HDF$^+$05], [HBH$^+$07]), most of the properties of the type system for the Java-like language (called Resource Aware JAva – RAJA) are still unknown.

This paper provides algorithmic typing rules for that system. We prove soundness and completeness of algorithmic typing with respect to the declarative typing from [HJ06]. This allows for automatic type checking under relatively mild annotations. In particular, we automatically construct types arising from sharing and conditionals which had to be provided manually beforehand. This enables a realistic implementation of the system which we also provide.

The notion of subtyping we use is slightly more flexible than the one from [HJ06] and thus allows more examples to be typed. Semantic soundness of the improved system is a direct extension of the soundness proof in [HJ06] and can be found in the following manuscript: [HJR].

Contents. Section 2 describes briefly the system RAJA and motivates it with some examples. In Section 3 we define the type-checking algorithm and show its soundness and completeness w.r.t. the declarative system. We then argue that typechecking can be performed efficiently, i.e. in small-degree polynomial time. Finally, in Sections 4 and 5, we discuss future and related work.

2 FJEU and RAJA

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 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. The syntax of FJEU is given in Fig. 1. The let-normal form of terms was merely chosen to eliminate boring redundancies from our proofs. In our implementation we transform nested expressions into let-normal form and infer a type for the let expressions by a simple preprocessing.

We will use a couple of shorthand notations: We write $\mathsf{S}(C)$ to denote the *super-class* D of a class 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. $\mathsf{A}(C) := \{a_1, \ldots, a_k\} \dot\cup \mathsf{A}(D)$. We write $C.\boldsymbol{a_i}$ to denote the class type of each attribute a_i of class C. Similarly we write $\mathsf{M}(C)$ to denote the set of all defined method names

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

Fig. 1. The syntax of FJEU

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 $C.m$ to denote the *method type* of m in class C. We base our statical resource analysis on the standard operational semantics that can be found in [HJR].

Example 1 (Copy of 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. Most programmers would use this style only for more complex tree-like data structures relying on "null" to model the empty list. We use it here in order to have a simple enough running example.

```
class List { List copy(){return null;} }
class Nil  extends List { List copy() { return this; }}
class Cons extends List { int elem; List next;
     List copy(){ let List res = new Cons in
          let List res1 = res.elem <- this.elem in
          let List res2 = res1.next <- this.next.copy() in return res2;}}
```

2.1 The System RAJA

Definition 1. *A* RAJA program *is an annotation of an FJEU class table* $\mathscr{C}$ *in the form of a sextuple* $\mathscr{R} = (\mathscr{C}, \mathscr{V}, \Diamond(\cdot), \mathsf{A}^{\mathrm{get}}(\cdot\,,\cdot)\,, \mathsf{A}^{\mathrm{set}}(\cdot\,,\cdot)\,, \mathsf{M}(\cdot\,,\cdot))$ *specified as follows:*

$\mathscr{V}$ *is a possibly infinite set of* views*. A* RAJA class *or* refined type *consists of a class* C *and a view* r *and is written* C^r*. We use the letters* r, s, p, q *to denote views. The meaning of views is given by the maps:*

1. $\Diamond(\cdot)$ *assigns to each RAJA class* C^r *a number* $\Diamond(C^r) \in \mathbb{D}$*, where* $\mathbb{D} = \mathbb{Q}^+ \cup \infty$*.*
2. $\mathsf{A}^{\mathrm{get}}(\cdot\,,\cdot)$ *and* $\mathsf{A}^{\mathrm{set}}(\cdot\,,\cdot)$ *assign to each RAJA class* C^r *and attribute* $a \in \mathsf{A}(C)$ *two views* $q = \mathsf{A}^{\mathrm{get}}(C^r, a)$ *and* $s = \mathsf{A}^{\mathrm{set}}(C^r, a)$*.*
3. $\mathsf{M}(\cdot\,,\cdot)$ *assigns to each RAJA class* C^r *and method* $m \in \mathsf{M}(C)$ *having method type* $E_1, \dots, E_j \to E_0$ *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* $r_1, \ldots, r_j \xrightarrow{p/q} r_0$. *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 introduce views and RAJA classes because we want to be able to assign objects of the same class different potentials.The number $\Diamond(\cdot)$ will be used to define the potential of a heap configuration under a given static RAJA typing. The exact definition is omitted here for lack of space and can be found in [HJR]. Essentially, the potential of a program state is the sum of the annotations of all its objects determined by their RAJA-type. Each access path (alias) to an object makes a separate contribution to that sum. In reasonable typings of circular data structures one arranges that all but finitely many paths make a nonzero contribution.

If $D = C.a$ is the FJEU type of attribute a in C then the RAJA class $D^{\mathsf{A}^{\mathsf{get}}(C^r a)}$ will be the type used when reading a, whereas the (intendedly stronger) type $D^{\mathsf{A}^{\mathsf{set}}(C^r a)}$ must be used when updating a. The stronger typing is needed since an update will possibly affect several aliases.

If a method m has a RAJA method type $E_1^{r_1}, \ldots, E_j^{r_j} \xrightarrow{p/q} E_0^{r_0}$ 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.

Example 2 (RAJA annotation of copy of singly-linked lists).
We aim at analysing the heap-space requirements of the program of Example 1. It is clear that the memory consumption of a call `l.copy()` will equal the length of the list `l`. To calculate this formally we use 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. In the following we show the RAJA annotation of Example 1 in the syntax of our implementation.

```
class List { rich, poor : pot = 0;
      rich : List<poor>,0 copy(0) { return null; }
}
class Nil extends List { rich, poor : pot = 0;
      rich : List<poor>,0 copy(0) { return this; }
}
class Cons extends List { rich : pot = 1; poor : pot = 0;
      rich : List<rich,rich> next;
      poor : List<poor,poor> next;
      rich, poor: int elem;

      rich : List<poor>,0 copy(0) { let List res = new Cons in
        let List res1 = res.elem <- this.elem in
        let List res2 = res1.next <- this.next.copy() in return res2; }
}
```

The RAJA type of the method copy states that it is only defined in $\mathsf{List}^{\mathsf{rich}}$, $\mathsf{Nil}^{\mathsf{rich}}$ and $\mathsf{Cons}^{\mathsf{rich}}$, but not in, e.g., $\mathsf{List}^{\mathsf{poor}}$. It will consume the potential of this and no additional potential. Upon successful completion the return value will be of type $\mathsf{List}^{\mathsf{poor}}$ hence carry potential 0. In addition to this the method will return no more potential. Thus, the typing amounts to saying that the memory consumption of every call to copy is bounded by the potential of this, that in case of $\mathsf{Cons}^{\mathsf{rich}}$ is equal 1 and in case of $\mathsf{Nil}^{\mathsf{rich}}$ is equal 0. If a list of length n is to be copied, the method will be called $n+1$ times, and the potential consumed will be bounded by n. More examples can be found in the RAJA web page [raj].

RAJA Subtyping Relation. RAJA subtyping is an extension of FJEU subtyping ($<:$), which is based on inheritance. We provide here a new definition of subtyping w.r.t. [HJ06]. There, a subtyping relation $r \sqsubseteq s$ on views was defined, based on all classes of the class table. Then, subtyping of RAJA classes $C^r <: D^s$ was defined as $C <: D$ and $r \sqsubseteq s$. This made subtyping unnecessarily rigid. For example $\mathsf{Nil}^{\mathsf{rich}} <: \mathsf{Nil}^{\mathsf{poor}}$ did not hold because $\mathsf{rich} \sqsubseteq \mathsf{poor}$ did not hold due to the class Cons. The new subtyping relation is defined directly on refined types. However, the straightforward definition where $C^r <: D^s$ only depends on C and D is unsound. It is necessary to analyse the subclasses of C and D as well because resource usage is determined by the dynamic type of the expressions.

Definition 2 (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 $C^r <: D^s \iff$ for each $E <: C$, $F <: D$ with $E <: F$:*

$$\Diamond(E^r) \geq \Diamond(F^s) \tag{2.1}$$

$$\forall a \in \mathsf{A}(F)\ .\ (F.a)^{\mathsf{A}^{\mathsf{get}}(E^r,a)} <: (F.a)^{\mathsf{A}^{\mathsf{get}}(F^s,a)} \tag{2.2}$$

$$\forall a \in \mathsf{A}(F)\ .\ (F.a)^{\mathsf{A}^{\mathsf{set}}(F^s,a)} <: (F.a)^{\mathsf{A}^{\mathsf{set}}(E^r,a)} \tag{2.3}$$

$$\forall m \in \mathsf{M}(F)\ .\ \forall \beta \in \mathsf{M}(F^s, m)\ .\ \exists \alpha \in \mathsf{M}(E^r, m)\ .\ (F.m)^{\alpha} <: (F.m)^{\beta} \tag{2.4}$$

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

Definition 3 (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 $i = 1, \ldots, j$.*

Sharing Relation. The sharing relation $\curlyvee(\cdot\,|\,\cdot)$ is important for correctly using variables more than once. In a RAJA program, if a variable is to be used more than once, 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 for that variable. For example if we have $l : \mathsf{List}^{\mathsf{rich}}$ we can use the variable l with the types $\mathsf{List}^{\mathsf{s}_1}$ and $\mathsf{List}^{\mathsf{s}_2}$ if $\curlyvee\big(\mathsf{List}^{\mathsf{rich}}\,|\,\mathsf{List}^{\mathsf{s}_1}, \mathsf{List}^{\mathsf{s}_2}\big)$ holds. In [HJ06] sharing was defined on views, i.e. $\curlyvee(r\,|\,s_1, \ldots, s_n)$ which is less flexible and precludes several examples.

Definition 4 (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 $E <: C$, $F <: D$ with $E <: F$:*

$$\Diamond(E^r) \geq \sum_i \Diamond(F^{s_i}) \tag{2.5}$$

$$\forall i \,.\, E^r <: F^{s_i} \tag{2.6}$$

$$\forall a \in \mathrm{dom}(\mathsf{A}(F)) \,.\, \curlyvee\Big((F.a)^{\mathsf{A}^{\mathrm{get}}(E^r,a)} \,\Big|\, (F.a)^{\mathsf{A}^{\mathrm{get}}(F^{s_1},a)}, \ldots, (F.a)^{\mathsf{A}^{\mathrm{get}}(F^{s_n},a)}\Big) \tag{2.7}$$

We define sharing similarly to subtyping, so that the following can be proved: subtyping and sharing coincide when the multiset of RAJA types consists of only one element.

Lemma 1. $C^r <: C^s \iff \curlyvee(C^r \,|C^s)$

Typing RAJA. The RAJA-typing judgment is formally defined by the rules in Figure 2. 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 an FJEU 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 judgment 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 extend the typing rules of FJEU. 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 we come to implement the system in the next section. $(\Diamond Waste)$ corresponds to the rule of subsumption of subtyping systems and weakens context, type, and effect. Herein, $\Gamma <: \Theta$ means $\forall x \in \Theta \,.\, \Gamma_x <: \Theta_x$.

The purpose of the $(\Diamond Share)$ rule is to ensure that a variable can be used twice without duplication of potential. Suppose we have the following expression:

$$\Gamma, l : \mathsf{List}^{\mathsf{rich}} \vdash^{n}_{n'} \mathsf{let}\ nl = l.\mathsf{copy}()\ \mathsf{in}\ l.\mathsf{copy}() : \mathsf{List}^{\mathsf{poor}} \tag{2.8}$$

If we allow the second call to the copy method we would be creating objects without "paying" for it, which would be unsound. Since the method copy is only defined for the view rich, the only possibility of typing (2.8) would be that $\curlyvee\big(\mathsf{List}^{\mathsf{rich}} \,\big|\, \mathsf{List}^{\mathsf{rich}}, \mathsf{List}^{\mathsf{rich}}\big)$ would hold, but it does not because $\Diamond\big(\mathsf{Cons}^{\mathsf{rich}}\big) < \Diamond\big(\mathsf{Cons}^{\mathsf{rich}}\big) + \Diamond\big(\mathsf{Cons}^{\mathsf{rich}}\big)$. Notice that the declarative type system as it is gives no procedure to find those intermediate views. To actually find them in order to implement the system is not trivial and will be discussed in the next section.

The judgment $\vdash m : \alpha\ \mathsf{ok}$ means that α is a valid RAJA type for a method m if the method body of m can be typed with the arguments, return type and effects as specified in α. Programs, then, are well-typed if all method bodies admit the announced type and, moreover, view and potential annotations are compatible with subtyping. Formally,

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

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

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

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

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

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

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

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

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

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

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

Fig. 2. Typing RAJA

Definition 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. $\forall m \in \mathsf{M}(C)\ .\ \forall \alpha \in \mathsf{M}(C^r, m)\ .\ \vdash m : \alpha\ \mathsf{ok}$

2.2 Algorithmic Views and Complete RAJA Programs

In this section we define algorithmic views which provide least upper and greatest lower bounds for subtyping restricted to refinements of a fixed FJEU type and also a formal addition operation on these refinements allowing us to infer the necessary type of a variable from the types of its (multiple) occurrences. Finally, they include operations to construct the intermediate views in method typings.

Recall the copy method of Example 2. We need one item of potential in order to create a $\mathsf{Cons}^{\mathsf{poor}}$ object . We said before that this object creation will be payed with the potential of this, but how exactly? In order to use the potential of the variable this of RAJA-class $\mathsf{Cons}^{\mathsf{rich}}$, we put it in the context with a modified type, for example, $\mathsf{Cons}^{\mathsf{rich}\dot{-}1}$, which is a view defined just like rich but with potential 0 everywhere. Moreover we find another view with potential 1, which we call $1(\mathsf{rich})$, such that $\curlyvee\Big(\mathsf{Cons}^{\mathsf{rich}} \,\Big|\, \mathsf{Cons}^{\mathsf{rich}\dot{-}1}, \mathsf{Cons}^{1(\mathsf{rich})}\Big)$ holds. Then we can derive: $\mathsf{this}:\mathsf{Cons}^{\mathsf{rich}\dot{-}1} \vdash^{1}_{0} \mathsf{let}\,\mathsf{List}\ \mathsf{res} = \mathsf{new}\ \mathsf{Cons}^{\mathsf{poor}}\ \mathsf{in}\ \ldots\ \mathsf{in}\ \mathsf{return}\ \mathsf{res};$

The declarative rule gives no information about how to find the views q and s. In order to find them algorithmically, we will introduce special algorithmic views like $\mathsf{rich} \dot{-} 1$ and $1(\mathsf{rich})$.

Definition 6 (Algorithmic views). *Let $\mathscr{R}$ be a RAJA-program. We extend the given set of views $\mathscr{V}$ by algorithmic views*

$$\delta, \gamma ::= \ s_1 \vee s_2 \ \mid\ s_1 \wedge s_2 \ \mid\ s_1 + s_2 \ \mid\ s \dot{-} \mathsf{n} \ \mid\ \mathsf{n}(s) \qquad s, s_1, s_2 \in \mathscr{V}, n \in \mathbb{D}$$

by extending the given maps $\Diamond(\cdot), \mathsf{A}^{\mathrm{get}}(\cdot,\cdot), \mathsf{A}^{\mathrm{set}}(\cdot,\cdot), \mathsf{M}(\cdot,\cdot)$ according to Fig. 3.

Definition 7 (Complete 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* complete *if the following conditions are satisfied. Let $* \in \{\wedge, \vee, +\}$.*

1. *$s_1 * s_2 \in \mathscr{V}$, for all $s_1, s_2 \in \mathscr{V}$.*
2. *$s \dot{-} \mathsf{n}$, $\mathsf{n}(s) \in \mathscr{V}$, for all $s \in \mathscr{V}, n \in \mathbb{D}$.*
3. *The annotation table of $\mathscr{R}$ satisfies the equations from Def. 6.*

Given a RAJA program $\mathscr{R}$ we can complete it with algorithmic views. $C^{s_1 \vee s_2}$ is the least upper bound of C^{s_1} and C^{s_2} and $C^{s_1 \wedge s_2}$ is the greatest lower bound of C^{s_1} and C^{s_2}. $C^{s_1+s_2}$ is defined such that $\curlyvee(C^s \,|C^{s_1}, C^{s_2})$ is equivalent to $C^s <: C^{s_1+s_2}$. This way we can deal only with subtyping instead of sharing, which is simpler and more intuitive. Finally, the views $\mathsf{n}(s)$ are neutral views of potential n and set-views like s. They are intended to be used together with the views $s \dot{-} \mathsf{n}$, which are nothing but the view s, with n units of potential stripped-off. This way, we get $\curlyvee\Big(C^s \,\Big|\, C^{s\dot{-}\mathsf{n}}, C^{\mathsf{n}(s)}\Big)$. These algorithmic views are useful for implementing $\vdash m : \alpha\ \mathsf{ok}$. If we need to use n units of potential of the type C^s of this in the method body of a given method, we give this the type $C^{s\dot{-}\mathsf{n}}$ and use the potential of $C^{\mathsf{n}(s)}$ in the method.

Of course, we are free to use the algorithmic views from the beginning and in particular in the provided class and method typings. They may be seen as a

Let $C \in \mathscr{C}$, $a \in \mathsf{A}(C)$ and $m \in \mathsf{M}(C)$. We set:

$$
\begin{array}{ll}
\Diamond(C^{s_1 \wedge s_2}) & = \mathsf{max}(\Diamond(C^{s_1}), \Diamond(C^{s_2})) \\
\Diamond(C^{s_1 \vee s_2}) & = \mathsf{min}(\Diamond(C^{s_1}), \Diamond(C^{s_2})) \\
\Diamond\left(C^{s_1 + s_2}\right) & = \Diamond(C^{s_1}) + \Diamond(C^{s_2}) \\
\Diamond\left(C^{\mathsf{n}(s)}\right) & = n \\
\Diamond\left(C^{s \dot{-} \mathsf{n}}\right) & = \begin{cases} \Diamond(C^s) \dot{-} n & \Diamond(C^s) \geq n \\ 0 & \text{otherwise} \end{cases}
\end{array}
$$

$$
\begin{array}{ll}
\mathsf{A}^{\mathrm{get}}(C^{s_1 \wedge s_2}; a) & = \mathsf{A}^{\mathrm{get}}(C^{s_1}; a) \wedge \mathsf{A}^{\mathrm{get}}(C^{s_2}; a) \\
\mathsf{A}^{\mathrm{get}}(C^{s_1 \vee s_2}; a) & = \mathsf{A}^{\mathrm{get}}(C^{s_1}; a) \vee \mathsf{A}^{\mathrm{get}}(C^{s_2}; a) \\
\mathsf{A}^{\mathrm{get}}(C^{s_1 + s_2}; a) & = \mathsf{A}^{\mathrm{get}}(C^{s_1}; a) + \mathsf{A}^{\mathrm{get}}(C^{s_2}; a) \\
\mathsf{A}^{\mathrm{get}}\left(C^{\mathsf{n}(s)}; a\right) & = \mathsf{0}(s) \\
\mathsf{A}^{\mathrm{get}}\left(C^{s \dot{-} \mathsf{n}}; a\right) & = \mathsf{A}^{\mathrm{get}}(C^s; a)
\end{array}
$$

$$
\begin{array}{ll}
\mathsf{M}(C^{s_1 \wedge s_2}; m) & = \mathsf{M}(C^{s_1}; m) \cap \mathsf{M}(C^{s_2}; m) \\
\mathsf{M}(C^{s_1 \vee s_2}; m) & = \mathsf{M}(C^{s_1}; m) \cup \mathsf{M}(C^{s_2}; m) \\
\mathsf{M}\left(C^{s_1 + s_2}; m\right) & = \mathsf{M}(C^{s_1}; m) \cap \mathsf{M}(C^{s_2}; m) \\
\mathsf{M}\left(C^{\mathsf{n}(s)}; m\right) & = \emptyset \\
\mathsf{M}\left(C^{s \dot{-} \mathsf{n}}; m\right) & = \mathsf{M}(C^s; m)
\end{array}
$$

$$
\begin{array}{ll}
\mathsf{A}^{\mathrm{set}}(C^{s_1 \wedge s_2}; a) & = \mathsf{A}^{\mathrm{set}}(C^{s_1}; a) \vee \mathsf{A}^{\mathrm{set}}(C^{s_2}; a) \\
\mathsf{A}^{\mathrm{set}}(C^{s_1 \vee s_2}; a) & = \mathsf{A}^{\mathrm{set}}(C^{s_1}; a) \wedge \mathsf{A}^{\mathrm{set}}(C^{s_2}; a) \\
\mathsf{A}^{\mathrm{set}}(C^{s_1 + s_2}; a) & = \mathsf{A}^{\mathrm{set}}(C^{s_1}; a) \vee \mathsf{A}^{\mathrm{set}}(C^{s_2}; a) \\
\mathsf{A}^{\mathrm{set}}\left(C^{\mathsf{n}(s)}; a\right) & = \mathsf{A}^{\mathrm{set}}(C^s; a) \\
\mathsf{A}^{\mathrm{set}}\left(C^{s \dot{-} \mathsf{n}}; a\right) & = \mathsf{A}^{\mathrm{set}}(C^s; a)
\end{array}
$$

Fig. 3. Definition of $\Diamond(\cdot), \mathsf{A}^{\mathrm{get}}(\cdot\,,\cdot), \mathsf{A}^{\mathrm{set}}(\cdot\,,\cdot), \mathsf{M}(\cdot\,,\cdot)$ of algorithmic views

shorthand for a longer table which includes them explicitly. We stress, though, that efficient type checking for *incomplete* programs is not possible with the techniques from this paper.

We do not consider typechecking of incomplete programs to be of any practical relevance. The following lemma summarizes the desirable order- and proof-theoretic properties of algorithmic views:

Lemma 2. *Let $C, D \in \mathscr{C}$ and $s, s_1, s_2, \ldots, s_n, q_1, q_2, \ldots, q_n \in \mathscr{V}$.*

1. *$C^{s_1 \vee s_2}$ is the least upper bound of C^{s_1} and C^{s_2}.*
2. *$C^{s_1 \wedge s_2}$ is the greatest lower bound of C^{s_i}.*
3. *$\curlyvee(C^s \,|C^{s_1}, \ldots, C^{s_n}) \iff C^s <: C^{s_1 + \ldots + s_n}$.*
4. *If $C^s <: C^{s_1 + \ldots + s_n}$ and $C^{s_i} <: C^{q_i}$ for all i, then $C^s <: C^{q_1 + \ldots + q_n}$.*
5. *$C^{s + \mathsf{0}(s)} = C^s$.*
6. *If $n \leq \Diamond(C^s)$ then $\curlyvee\left(C^s \,\middle|\, C^{s \dot{-} \mathsf{n}}, C^{\mathsf{n}(s)}\right)$. Moreover, $\curlyvee(C^s \,|C^{s_1}, C^{s_2})$ and $\Diamond(C^{s_2}) \geq n$ imply $C^{s \dot{-} \mathsf{n}} <: C^{s_1}$.*

Algorithmic typechecking now faces one more obstacle. Officially, one method can have infinitely many RAJA types. This does not compromise semantic type soundness, but must of course be restricted to finitely many to enable algorithmic type checking.

Moreover, the rule ($\Diamond$*Invocation*) chooses non-deterministically one monomorphic RAJA method type according to the given method call. In order for algorithmic typing to be efficient (not NP-complete) we need to make sure that there is an optimal such choice in any situation.

Definition 8. *If $\alpha = r_1, \ldots, r_j \xrightarrow{p/p'} r_0$ and $\beta = s_1, \ldots, s_j \xrightarrow{n/n'} s_0$ then $\alpha \sqsubseteq \beta$ iff $p \leq n$ and $p - p' \leq n - n'$.*

Definition 9. *A RAJA-program is* algorithmic *if it is finite, complete, and for all C, r, m the set $\mathsf{M}(C^r, m)$ is totally ordered by the ordering in Def. 8.*

From now on we assume that all RAJA-programs are algorithmic without explicit notice.

3 Algorithmic Typing of RAJA Programs

In this section we present an algorithm for typechecking RAJA programs. Algorithmic type-checking must consist of syntax directed rules, thus, the rules $(\Diamond Share)$ and $(\Diamond Waste)$ must be integrated in other rules. Instead of using $(\Diamond Waste)$, we integrate subtyping in the rules.

The purpose of the $(\Diamond Share)$ rule is to ensure that a variable can be used more than once without unsound incrementation of potential. The main challenge for implementing it is that it contains no information about how to find the views q_1 to q_n for the different occurrences. The current implementation does not include inference of these views. Instead, every variable occurrence has been annotated with the corresponding view, which can be an algorithmic view. The task of the type checker is then to check the correctness of the given sharing, or, more exactly, since, as we saw in last section, using algorithmic views a sharing task can be reduced into a subtyping task, the algorithm checks only subtyping. The inference of these intermediate views remains under investigation.

The computed resource annotations in rules ($\vdash Let$) and ($\vdash Cond.$) are a bit intricate. Ultimately, they are justified by soundness and completeness. Rule ($\vdash Let$) may be easier to understand if broken down into the two cases $m \geq n'$ and $m < n'$. In the latter case the output of the first computation suffices to satisfy the second one. In the former case extra input potential must be provided for the second computation. In rule ($\vdash Cond.$) we must cater for both computations, hence the max and the min. The adaptations $u - n$ and $u - m$ cater for the case where, say, $m \geq n$ units were provided due to the max, yet the first branch of the conditional was taken hence only n units "used" and vice versa.

In the rule ($\vdash Inv.$) we choose the minimal RAJA monomorphic type that satisfies the subtyping conditions. Since the algorithmic system considers only finite programs and the set of RAJA monomorphic types is totally ordered according to $\sqsubseteq$, every nonempty subset of $M(G^r, m)$ has a minimal element.

We define the judgment $\Delta^{\Psi} \overset{n}{\underset{n'}{\vdash}} e^{\circ} \leftleftarrows C^{\gamma}$ inductively by the rules in Figure 4, where Δ, e° and C^{γ} are inputs and Ψ, n and n' are outputs. Δ is an FJEU context, i.e. a map from variable names to FJEU types. Ψ is a map from variable names to algorithmic views. C^{γ} is an algorithmic RAJA type, which is an FJEU class refined with an algorithmic view and e° is an annotated FJEU expression.

The notation Δ^{Ψ} means that for every variable $x \in \Delta$, if $\Delta_x = C$ and $\Psi_x = \delta$ then $\Delta_x^{\Psi} = C^{\delta}$. We also use the notation $\Delta^{\Psi_1 + \Psi_2}$ for meaning that if $\Delta_x^{\Psi_1} = C^{\delta_1}$ and $\Delta_x^{\Psi_2} = C^{\delta_2}$ then $\Delta_x^{\Psi_1 + \Psi_2} = C^{\delta_1 + \delta_2}$. The meaning of $\Delta^{\Psi_1 \wedge \Psi_2}$ is similar. We write $x{:}C^r, {}^+ y{:}D^s$ for the following two cases. The usual case is $x \neq y$ and then it means nothing but $x{:}C^r, y{:}D^s$. On the other hand, if $x = y$, then $C = D$ too, and the notation means $x{:}C^{r+s}$. We write $\Delta^{\Psi_\emptyset}$ for meaning $\Delta_x^{\Psi_\emptyset} = C^{\mathbf{0}(s)}$ where

$\Delta_x = C$ and s is one of the view annotations of x or any view if x is not used in the program. The idea is to return neutral views for variables that are not used in the given expression. Finally, let e° denote an annotated RAJA expression. In summary, we define the partial function $\mathsf{typecheck}(\Delta, e^\circ, C^\gamma)$ (Fig. 4).

Next, we define the algorithmic judgment $\vdash_{\mathsf{a}} m : \alpha\ \mathsf{ok}$ based on algorithmic typing. (Fig. 4). The typechecking algorithm returns a greater context than the declared one. This has to be checked. Moreover, it calculates the space consumption u of the method body. If $u \leq n$ then n items are enough and we do not need any potential from this. Otherwise, we calculate how many items of potential we need from this, i.e. $p = u \dot{-} n$, and we of course have to check whether the potential of this is at least p. Finally, the amount of freelist units u' released by the expression should be at least $n' + u - (n + p)$.

In the following we show that the algorithmic typing system we just defined is correct w.r.t. the declarative typing system of RAJA. If Γ is a RAJA context, we write $|\Gamma|$ for meaning its underlying FJEU context.

Lemma 3 (Soundness of algorithmic RAJA typing)
If $\Delta^\Psi \overset{n}{\underset{n'}{\vdash}} e^\circ \leftleftarrows C^\gamma$ *then* $\Delta^\Psi \overset{n}{\underset{n'}{\vdash}} e : C^\gamma$.

Proof By induction on algorithmic typing derivations, using the $(\Diamond Waste)$ rule and Lemma 2.

Lemma 4 (Soundness of algorithmic RAJA method typing). *Given a RAJA type* C^r*, a method* $m \in \mathsf{M}(C)$ *and a RAJA method type* $\alpha \in \mathsf{M}(C^r, m)$*, if* $\vdash_{\mathsf{a}} m : \alpha\ \mathsf{ok}$ *then* $\vdash m : \alpha\ \mathsf{ok}$.

Proof. Follows by Lemma 3.

The completeness proof is a bit more complicated than the soundness proof. The reason for this is that we have eliminated the rules $(\Diamond Share)$ and $(\Diamond Waste)$ and we have to show that typing derivations that use these rules are still admissible in the algorithmic system. The following lemma states the admissibility of sharing in the algorithmic system.

Lemma 5 (Share). *Let* $\Delta^\Psi, y_1 : D^{\delta_1}, \ldots, y_n : D^{\delta_n} \overset{n}{\underset{n'}{\vdash}} e^\circ \leftleftarrows C^\gamma$*. Then* $\Delta^\Psi, x : D^\delta \overset{n}{\underset{n'}{\vdash}} e[x/y_1, \ldots, x/y_n]^\circ \leftleftarrows C^\gamma$ *where either* $\delta = \delta_1 + \ldots + \delta_n$ *or* $\delta = \delta_1 \wedge \ldots \wedge \delta_n$.

Proof. By induction on algorithmic typing derivations.

Lemma 6 (Waste). *Let* $\Lambda^\Psi \overset{u}{\underset{u'}{\vdash}} e^\circ \leftleftarrows D^\gamma$*,* $D^\gamma <: C^\delta$ *and* $\Delta <: \Lambda$ *then* $\Delta^\Psi \overset{w}{\underset{w'}{\vdash}} e^\circ \leftleftarrows C^\delta$ *for some* $w \leq u$ *and* $w' \geq u' + w - u$.

Proof. By induction on algorithmic typing derivations.

Lemma 7 (Completeness of algorithmic RAJA typing).
If $\Gamma \overset{n}{\underset{n'}{\vdash}} e : C^r$ *then there is an annotated version* e° *of the expression* e *with* $|\Gamma|^\Psi \overset{u}{\underset{u'}{\vdash}} e^\circ \leftleftarrows C^r$ *for some* $u \leq n$ *and* $u' \geq n' + u - n$ *so that* $\Gamma <: |\Gamma|^\Psi$.

Proof. By induction on typing derivations, using Lemma 5 and 6.

Algorithmic RAJA Typing $\boxed{\Delta^{\Psi} \mid\!\!\frac{n}{n'}\; e^{\circ} \leftleftarrows C^{r}}$

$$\frac{D^{\gamma} <: C^{\gamma}}{\Delta^{\Psi_\emptyset} \mid\!\!\frac{\Diamond(D^{\gamma})+1}{0}\; \mathsf{new}\; D \leftleftarrows C^{\gamma}}\;(\vdash New) \qquad \frac{E^{q} <: C^{\gamma}}{\Delta^{\Psi_\emptyset}, x{:}E^{q} \mid\!\!\frac{0}{0}\; x^{q} \leftleftarrows C^{\gamma}}\;(\vdash Var)$$

$$\frac{}{\Delta^{\Psi_\emptyset}, x{:}C^{q} \mid\!\!\frac{0}{\Diamond(C^{q})+1}\; \mathsf{free}\,(x^{q}) \leftleftarrows E^{\gamma}}\;(\vdash Free)$$

$$\frac{D <: E \;\;(\text{or}\;\; E <: D) \qquad D^{q} <: C^{\gamma}}{\Delta^{\Psi_\emptyset}, x{:}E^{q} \mid\!\!\frac{0}{0}\; (D)x^{q} \leftleftarrows C^{\gamma}}\;(\vdash Cast) \qquad \frac{}{\Delta^{\Psi_\emptyset} \mid\!\!\frac{0}{0}\; \mathsf{null} \leftleftarrows C^{\gamma}}\;(\vdash Null)$$

$$\frac{\mathsf{A}^{\mathsf{get}}(C^{r}, a) = q \qquad C.a = E \qquad E^{q} <: D^{\gamma}}{\Delta^{\Psi_\emptyset}, x{:}C^{r} \mid\!\!\frac{0}{0}\; x^{r}.a \leftleftarrows D^{\gamma}}\;(\vdash Access)$$

$$\frac{\mathsf{A}^{\mathsf{set}}(E^{q}, a) = s \qquad E.a = D \qquad F^{p} <: D^{s} \qquad E^{q} <: C^{\gamma}}{\Delta^{\Psi_\emptyset}, x{:}E^{q}, {}^{+}\, y{:}F^{p} \mid\!\!\frac{0}{0}\; x^{q}.a \leftarrow y^{p} \leftleftarrows C^{\gamma}}\;(\vdash Update)$$

$$\frac{\Delta^{\Psi'} \mid\!\!\frac{n}{n'}\; e_1^{\circ} \leftleftarrows D^{\gamma_1} \qquad \Delta^{\Psi''}, x{:}D^{\gamma_1} \mid\!\!\frac{m}{m'}\; e_2^{\circ} \leftleftarrows C^{\gamma_2}}{\Delta^{\Psi'+\Psi''} \mid\!\!\frac{\mathsf{max}(\,n,\, n+m-n'\,)}{\mathsf{max}(\,m',\, m'+n'-m\,)}\; \mathsf{let}\; D\, x = e_1^{\circ}\; \mathsf{in}\; e_2^{\circ} \leftleftarrows C^{\gamma_2}}\;(\vdash Let)$$

$$\frac{x \in \Delta \quad \Delta^{\Psi'} \mid\!\!\frac{n}{n'}\; e_1^{\circ} \leftleftarrows C^{\gamma} \quad \Delta^{\Psi''} \mid\!\!\frac{m}{m'}\; e_2^{\circ} \leftleftarrows C^{\gamma} \quad u = \mathsf{max}(\,m, n)}{\Delta^{\Psi' \wedge \Psi''} \mid\!\!\frac{u}{\mathsf{min}(\,n'+u-n,\, m'+u-m\,)}\; \mathsf{if}\; x\; \mathsf{instanceof}\; E\; \mathsf{then}\; e_1^{\circ}\; \mathsf{else}\; e_2^{\circ} \leftleftarrows C^{\gamma}}\;(\vdash Cond.)$$

$$\frac{p/p' = \mathsf{arg\ min}\{\, (E_1^{q_1}, \ldots, E_j^{q_j} \xrightarrow{p/p'} E_0^{q_0}) \in M(G^{r}, m) \mid \forall i\,.\, F_i^{t_i} <: E_i^{q_i}, E_0^{q_0} <: C^{\gamma} \,\}}{\Delta^{\Psi_\emptyset}, x{:}G^{r}, {}^{+}\, y_1{:}F_1^{t_1}, {}^{+}\, \ldots, {}^{+}\, y_j{:}F_j^{t_j} \mid\!\!\frac{p}{p'}\; x^{r}.m\,(y_1^{t_1}, \ldots, y_j^{t_j}) \leftleftarrows C^{\gamma}}\;(\vdash Inv.)$$

Typecheck function

$$\mathsf{typecheck}(\Delta, e^{\circ}, C^{\gamma}) = \begin{cases} (\Psi, n, n') & \text{if } \Delta^{\Psi} \mid\!\!\frac{n}{n'}\; e^{\circ} \leftleftarrows C^{\gamma} \\ \mathsf{fail} & \text{otherwise} \end{cases}$$

Algorithmic RAJA Method Typing $\boxed{\vdash_{\mathsf{a}} m : \alpha\ \mathsf{ok}}$

$$\frac{\begin{array}{c}\alpha = E_1^{r_1}, \ldots, E_j^{r_j} \xrightarrow{n/n'} E_0^{r_0} \in \mathsf{M}(C^{r}, m) \\ \mathtt{this}{:}C^{\beta},\, x_1{:}E_1^{\beta_1}, \ldots, x_j{:}E_j^{\beta_j} \mid\!\!\frac{u}{u'}\; \mathsf{M}_{\mathsf{body}}(C, m)^{\circ} \leftleftarrows E_0^{r_0} \\ E_i^{r_i} <: E_i^{\beta_i} \quad p = u \mathbin{\dot{-}} n \quad u' \geq n' + u - (n+p) \quad \Diamond(C^{r}) \geq p \quad C^{r \mathbin{\dot{-}} \mathsf{p}} <: C^{\beta}\end{array}}{\vdash_{\mathsf{a}} m : \alpha\ \mathsf{ok}}$$

Fig. 4. Algorithmic RAJA Typing

Lemma 8 (Completeness of algorithmic RAJA method typing). *Given a RAJA type C^r, a method $m \in \mathsf{M}(C)$ and a RAJA method type $\alpha \in \mathsf{M}(C^r, m)$, if $\vdash m : \alpha$ ok then $\vdash_{\mathsf{a}} m : \alpha$ ok.*

Proof. Follows by Lemma 7.

Lemma 9 (Efficiency of algorithmic RAJA typing). *$\Delta^{\Psi} \overset{n}{\underset{n'}{\vdash}} e^{\circ} \leftleftarrows C^r$ is decidable in polynomial time.*

Proof (sketch). The syntax-directed backwards application of the algorithmic typing rules produces a linear number of subtyping and sharing constraints. Furthermore, the algorithmic view expressions occurring in these constraints are themselves of linear size. It then suffices to restrict attention to the views that occur as subexpressions of the ones appearing in the constraints. Their number is therefore polynomial in the size of the program. A complete table of the subtyping and sharing judgments for this relevant subset can then be computed iteratively in polynomial time. In practice, a goal-directed implementation performs even better.

Lemma 10. *Given a RAJA class C, a view r, a method $m \in \mathsf{M}(C)$ and a RAJA method type $\alpha \in \mathsf{M}(C^r, m)$, $\vdash m : \alpha$ ok is decidable.*

Proof. Follows by Lemmas 4, 8 and 9.

Theorem 1 (Decidability of RAJA typing). *Given a RAJA-Program $\mathscr{R}$, its well-typedness is decidable.*

Proof. Follows by Lemma 10.

4 Related Work

Since [HJ06] several authors have made contributions towards costing heap consumption of object-oriented programs. [MP07] uses methods from abstract interpretation and term rewriting (quasi interpretations) to estimate the size of data structures and thus indirectly heap consumption. The approach is promising, but aliasing does not seem to have been taken into account properly and not many examples are given. The interpretation of methods must be provided manually.

COSTA [AAG+07] is similar in that it assigns cost functions to methods and program parts. These refer directly to heap consumption and are given as solutions of automatically constructed recurrence systems. The main contribution of COSTA is an improved solver for these recurrences. COSTA is not as general as RAJA which, however, is not fully automatic.

Another promising fully automatic system is [GMC09] which works by instrumenting code with resource-counting, integer-valued "ghost"-variables and using modern tools from static analysis for estimating their range of values. The examples given are stunning, but do not involve dynamically allocated data

structures. With further progress with automatic analysis of arithmetic relationships between integer variables systems like SPEED may eventually render type-based analyses obsolete. More likely, however, is a combination of the two.

Finally, Java(X) [DTW07] is a type system quite similar to RAJA and developed independently which has, however, a different purpose, namely ensuring the correct usage of resources like files etc. according to a specified protocol. The paper [DTW07] does not present algorithmic type checking, let alone automatic type inference; it is likely that the algorithmic system presented here could be adapted to Java(X).

5 Conclusions

We have provided a type checking algorithm for RAJA programs and proved its correctness and efficiency in the sense of polynomial-time computability. In order to do this, we introduced algorithmic views which render the subtyping lattice more well behaved and could also be a useful addition to the declarative system which is exposed to the programmer. In this way, we were able to get rid of most type annotations in method bodies although we still have to indicate the types of multiple occurrences of a variable, i.e., how the potential belonging to the variable is to be split among the different occurrences.

The algorithmic typechecking and the implementation allow us to investigate larger examples which might prompt further extensions to RAJA. In particular, we would like to investigate the typability of the Iterator pattern and more challengingly patterns involving callbacks like Observer. From a pragmatic viewpoint, polymorphic quantification over views could be a useful extension, too.

Of course, full-blown type inference is also on our agenda, thus potentially rendering RAJA into a push-button analysis.

Acknowledgment. We acknowledge support by the EU integrated project MOBIUS IST 15905. We thank Andreas Abel, Lennart Beringer and Steffen Jost for valuable comments.

References

[AAG+07] Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: COSTA: Design and implementation of a cost and termination analyzer for java bytecode. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2007. LNCS, vol. 5382, pp. 113–132. Springer, Heidelberg (2008)

[Cam08] Campbell, B.: Type-based amortized stack memory prediction. PhD thesis, University of Edinburgh (2008)

[DTW07] Degen, M., Thiemann, P., Wehr, S.: Tracking linear and affine resources with JAVA(X). In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 550–574. Springer, Heidelberg (2007)

[FKF98] Flatt, M., Krishnamurthi, S., Felleisen, M.: Classes and mixins. In: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1998), New York, January 1998, pp. 171–183. Association for Computing Machinery (1998)

[GL98] Gómez, G., Liu, Y.A.: Automatic accurate time-bound analysis for high-level languages. In: Müller, F., Bestavros, A. (eds.) LCTES 1998. LNCS, vol. 1474, p. 31. Springer, Heidelberg (1998)

[GMC09] Gulwani, S., Mehra, K.K., Chilimbi, T.M.: SPEED: precise and efficient static estimation of program computational complexity. In: Shao, Z., Pierce, B.C. (eds.) POPL, pp. 127–139. ACM Press, New York (2009)

[Gro01] Grobauer, B.: Topics in Semantics-based Program Manipulation. PhD thesis, BRICS Aarhus (2001)

[HBH+07] Herrmann, C.A., Bonenfant, A., Hammond, K., Jost, S., Loidl, H.-W., Pointon, R.: Automatic amortised worst-case execution time analysis. In: 7th Int'l Workshop on Worst-Case Execution Time (WCET) Analysis, Proceedings, pp. 13–18 (2007)

[HDF+05] Hammond, K., Dyckhoff, R., Ferdinand, C., Heckmann, R., Hofmann, M., Jost, S., Loidl, H.-W., Michaelson, G., Pointon, R.F., Scaife, N., Srot, J., Wallace, A.: The embounded project (project start paper). In: van Eekelen, M.C.J.D. (ed.) Trends in Functional Programming. Trends in Functional Programming, vol. 6, pp. 195–210. Intellect (2005)

[HJ03] Hofmann, M., Jost, S.: 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] Hofmann, M.O., Jost, S.: Type-based amortised heap-space analysis. In: Sestoft, P. (ed.) ESOP 2006. LNCS, vol. 3924, pp. 22–37. Springer, Heidelberg (2006)

[HJR] Hofmann, M., Jost, S., Rodriguez, D.: Type-based amortised heap space analysis (complete soundness proof), `http://raja.tcs.ifi.lmu.de/download/files/rajaSoundProof.pdf`

[HP99] Hughes, J., Pareto, L.: Recursion and dynamic data-structures in bounded space, June 21 (1999)

[IPW99] Igarashi, A., Pierce, B., Wadler, P.: Featherweight Java: A minimal core calculus for Java and GJ. In: Meissner, L. (ed.) Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA 1999), N.Y., vol. 34(10), pp. 132–146 (1999)

[MP07] Marion, J.-Y., Péchoux, R.: Resource control of object-oriented programs. CoRR, abs/0706.2293, informal publication (2007)

[NCQR05] Nguyen, H.H., Chin, W.N., Qin, S., Rinard, M.C.: Memory usage inference for object-oriented programs (January 2005)

[Oka98] Okasaki, C.: Purely Functional Data Structures. Cambridge University Press, Cambridge (1998)

[raj] `http://raja.tcs.ifi.lmu.de`

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