Type-Based Amortised Heap-Space Analysis

Martin Hofmann[1] and Steffen Jost[2]

[1] LMU München, Institut für Informatik
[2] University of St Andrews, School of Computer Science

Abstract. We present a type system for a compile-time analysis of heap-space requirements of Java style object-oriented programs with explicit deallocation.

Our system is based on an amortised complexity analysis: the data is arbitrarily assigned a potential related to its size and layout; allocations must be "payed for" from this potential. The potential of each input then furnishes an upper bound on the heap space usage for the computation on this input.

We successfully treat inheritance, downcast, update and aliasing. Example applications for the analysis include destination-passing style and doubly-linked lists.

Type inference is explicitly not included; the contribution lies in the system itself and the nontrivial soundness theorem. This *extended abstract* elides most technical lemmas and proofs, even nontrivial ones, due to space limitations. A full version is available at the authors' web pages.

1 Introduction

Consider a Java-like class-based object-oriented language without garbage collection, but with explicit deallocation in the style of C's `free()`. Such programs may be evaluated by maintaining a set of free memory units, the freelist. Upon object creation a number of heap units required to store the object is taken from the 3 provided it contains enough units; each deallocated heap unit is returned to the freelist. An attempt to create a new object with an insufficient freelist causes unsuccessful abortion of the program. This also happens upon attempts to access a deallocated object via a stale pointer.

It is now natural to ask what initial size the freelist must have so that a given program may be executed without causing unsuccessful abortion due to penury of memory. If we know such a bound on the initial freelist size we can then execute our program within a fixed amount of memory which can be useful in situations where memory is a scarce resource like embedded controllers or SIM cards. It may also be useful to calculate such bounds for individual parts of a program so that several applications can be run simultaneously even if their maximum memory needs exceed the available capacity [6].

Typically, the required initial freelist size will depend on the data, for example the size of some initial data structure, e.g., a phone book or an HTML document.

P. Sestoft (Ed.): ESOP 2006, LNCS 3924, pp. 22–37, 2006.

We therefore seek to determine an upper bound on the required freelist size as a function of the input size. We propose to approach this input dependency by a type-based version of amortised analysis in the sense of Tarjan [17].

Amortised Analysis. In amortised analysis data structure(s) are assigned an arbitrary nonnegative number, the *potential*. The amortised cost of an operation is its total cost (time or space) plus the difference in potential before and after the operation. The sum of the amortised costs plus the potential of the initial data structure then bounds (from above) the actual cost of a sequence of operations. If the potential is cleverly chosen then the amortised cost of individual operations is zero or a constant even when their actual cost is difficult to determine. The simplest example is an implementation of a queue using two stacks A and B. Enqueuing is performed on A, dequeuing is performed on B unless B is empty in which case the whole contents of A are moved to B prior to dequeuing. Thus, dequeuing sometimes takes time proportional to the size of A. If we decree that the size of A is the potential of the data then enqueuing has an amortised cost of 2 (one for the actual cost, one for the increase in potential). Dequeuing on the other hand has an amortised cost of 1 since the cost of moving A over to (the empty stack) B cancels out against the decrease in potential. Thus, the actual cost of a sequence of operations is bounded by the initial size of A plus twice the number of enqueues plus the number of dequeues. In this case, one can also see this directly by observing that each element is moved exactly three times: once into A, once from A to B, once out of B.

Type-Based Potential. In the above queue example, both stacks have the same type, but each element of A contributes 1 to the overall potential, whereas each element of B contributes a potential of 0. We recorded this information within the type by adding a number to each type constructor in our previous work [9]. However, object-oriented languages require a more complex approach due to aliasing and inheritance: Where in a purely functional setting a *refined type* might consist of a simple type together with a number a refined (class) type will consist of a number together with refined types for the attributes and methods. In order to break the recursive nature of this requirement we resort to explicit names for refined types as is common practice in Java (though not in OCaml): we introduce a set of names, the *views*. A *view* on an object shall determine its contribution to the potential. This is formally described in Section 3, but we shall convey a good intuition here. A refined type then consists of a class C and a view r and is written C^r. We sometimes conveniently use a refined type where only a class or a view is needed.

The fact that views are in some sense orthogonal to class types caters for typecasting. If, e.g., x has refined type C^r, then $(D)x$ will have refined type D^r.[1]

[1] Peter Thiemann, Freiburg, independently and simultaneously used a similar approach in as yet unpublished work on a generic type-based analysis for Java. His main application is conformance of XML-document generators to standards and his treatment of aliasing is different from ours.

The meaning of views is given by three maps $\Diamond$ defining potentials, A defining views of attributes, and M defining refined method types. More precisely, $\Diamond : \text{Class} \times \text{View} \to \mathbb{Q}^+$ assigns each class its potential according to the employed view. Next, $\mathsf{A} : \text{Class} \times \text{View} \times \text{Field} \to \text{View} \times \text{View}$ determines the refined types of the fields. A different view may apply according to whether a field is read from (get-view) or written to (set-view), hence the codomain $\text{View} \times \text{View}$. Subtyping of refined types is behavioural and covariant in the get-view and contravariant in the set-view.

Finally, $\mathsf{M} : \text{Class} \times \text{View} \times \text{Method} \to \mathcal{P}(\text{Views of Arguments} \to \text{Effect} \times \text{View of Result})$ assigns refined types and effects to methods. The effect is a pair of numbers representing the potential consumed before and released after method invocation. We allow polymorphism in the sense that, as implied by the powerset symbol $\mathcal{P}$ one method may have more than one (or no) refined typing.

One and the same runtime object can have several refined types at once, since it can be regarded through different views at the same time. In fact, each *access path* leading from the current scope via field access to an object will determine its individual view on the object by the repeated application of A. The overall potential of a runtime 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 typesystem has an explicit contraction rule: If a variable is used more often, the associated potential is split by assigning different views to each use. The potential also depends on the dynamic class types of each object. However, our runtime model is the standard one which does not include any view/potential related information.

Our main contribution is the proof that the total potential plus the heap used never increases during execution. In other words, any object creation must be paid for from the potential in scope and the potential of the initial configuration furnishes an upper bound on the total heap consumption.

In this way, we can model data-dependent memory usage without manipulating functions and recurrences, as is the case in approaches based on sized types and also without any alteration to the runtime model.

We will now describe our approach in more detail using three suggestive examples: a copying function for lists, imperative append in destination passing style, and doubly-linked lists. These examples show many of the salient features of our methods: heap usage proportional to input size (the copying example), correct accounting for aliasing (destination passing style), circular data (doubly-linked lists). Especially the last example seems to go beyond the scope of current methods based on sized types or similar.

Example: Copying singly-inked lists in an object-oriented style:

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

It is clear that the memory consumption of a call `x.copy()` will equal the length of the list `x`. To calculate this formally we construct a view `a` 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 `b` that assigns potential 0 to all classes. The complete specification of the two views is shown here, together with other views used later:

$\Diamond(\cdot)$	a	b	c	d	n
List	0	0	0	0	0
Nil	0	0	0	0	0
Cons	1	0	0	1	0

	$\texttt{Cons}^{\texttt{a}}$	$\texttt{Cons}^{\texttt{b}}$	$\texttt{Cons}^{\texttt{c}}$	$\texttt{Cons}^{\texttt{d}}$	$\texttt{Cons}^{\texttt{n}}$
$\mathsf{A}^{\mathrm{get}}(\cdot, \texttt{next})$	a	b	a	n	n
$\mathsf{A}^{\mathrm{set}}(\cdot, \texttt{next})$	a	b	a	a	a

$$\mathsf{M}\big(\{\texttt{List}^{\texttt{a}}, \texttt{Cons}^{\texttt{a}}, \texttt{Nil}^{\texttt{a}}\}, \texttt{copy}\big) = () \xrightarrow{0/0} \texttt{b} \tag{1.1}$$

The call `x.copy()` is well-typed and of type $\texttt{List}^{\texttt{b}}$ if `x` has refined type $\texttt{List}^{\texttt{a}}$, $\texttt{Nil}^{\texttt{a}}$ or $\texttt{Cons}^{\texttt{a}}$. It is ill-typed if `x` has refined type, e.g., $\texttt{List}^{\texttt{b}}$. Its effect 0/0 will not decrement the freelist beyond the amount implicit in the potential of `x` (which equals in this case the length of the list pointed to by `x`) and will not return anything to the freelist beyond the amount implicit in the potential of the result (which equals zero due to its refined type $\texttt{List}^{\texttt{b}}$). Thus, the typing amounts to saying that the memory consumption of this call is equal to the length of `x`.

Let us explain why the potential of `x` indeed equals its length. Suppose for the sake of the example that `x` points to a list of length 2. The potential is worked out as the sum over all access paths emanating from `x` and not leading to `null` or being undefined. In this case, these access paths are $\boldsymbol{p}_1 = \texttt{x}, \boldsymbol{p}_2 = \texttt{x.next}$, $\boldsymbol{p}_3 = \texttt{x.next.next}$. Each of these has a dynamic type: `Cons` for $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$; `Nil` for $\boldsymbol{p}_3$. Each of them also has a view worked out by chaining the view of `x` along the get-views. Here it is view is `a` in each case. For each access paths we now look up the potential annotation of its dynamic type under its view. It equals 1 in case of $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$ given $\Diamond(\texttt{List}^{\texttt{a}}) = 1$ and 0 for $\boldsymbol{p}_3$ by $\Diamond(\texttt{Nil}^{\texttt{a}}) = 0$, yielding a sum of 2. Notice that if the very same list had been accessed via a variable `y` of type $\texttt{List}^{\texttt{b}}$ the potential ascribed would have been 0.

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

For the typing of `copy()` to be accepted we must derive the judgements

$$\texttt{this:Nil}^{\texttt{c}} \vdash^{0}_{0} e_{\texttt{Nil}} : \texttt{List}^{\texttt{b}} \qquad \texttt{this:Cons}^{\texttt{c}} \vdash^{1}_{0} e_{\texttt{Cons}} : \texttt{List}^{\texttt{b}} \tag{1.2}$$

where $e_{\texttt{Nil}}$ and $e_{\texttt{Cons}}$ are the bodies of `copy` in classes `Nil` and `Cons`, respectively. View `c` is basically the view `a` with the toplevel potential 1 stripped off. In exchange we get access to this toplevel potential in the form of the superscript 1 of the "turnstile". This weaker typing of `this` allows us to read attributes from `this` as if its typing were $\texttt{List}^{\texttt{a}}$ but it precludes the use of any toplevel potential which is right for otherwise there would be an unjustified duplication of potential.

Formally, this "stripping-off" is achieved through the coinductive definition of a relation $\curlyvee(r\,|\mathcal{D})$ between views r and multisets of views $\mathcal{D}$ which asserts that a variable of view r may be used multiple times provided the different occurrences are given the views in $\mathcal{D}$ and it is this relation that appears as a side condition to the typing rule for methods. Entry of a method body is the only point were potential becomes available for use, but it can be used anywhere inside the method's body and even passed on to further method calls.

In particular for the views listed in (1.1), we have $\curlyvee(\texttt{a}\,|\{\texttt{d},\texttt{c}\})$, $\curlyvee(\texttt{b}\,|\{\texttt{b},\texttt{b},\dots\})$, and $\curlyvee(\texttt{a}\,|\{\texttt{a},\texttt{n},\texttt{n}\dots\})$, but neither $\curlyvee(\texttt{a}\,|\{\texttt{a},\texttt{a}\})$ (because $1+1\neq 1$) nor $\curlyvee(\texttt{c}\,|\{\texttt{c},\texttt{c}\})$ (because the get-view of `next` in `c` is `a`), nor $\curlyvee(\texttt{a}\,|\{\texttt{a},\texttt{b}\})$ (because the set-view of `next` in `b` is not `a`, but the set-view has to be preserved upon sharing).

Typing "Copy". Let us now see how we can derive the required typings in (1.2). The typing of $e_{\texttt{Cons}}$ works as follows. The creation of a new object of class $\texttt{Cons}^{\texttt{b}}$ incurs a cost of 1 (zero potential plus one physical heap unit – note that the physical cost of object creation can be chosen arbitrarily for each class to suit the applicable memory model). Thus, it remains to justify

$$\texttt{this:Cons}^{\texttt{c}}, \texttt{res:Cons}^{\texttt{b}} \vdash^{0}_{0}$$

```
res.elem=this.elem; res.next=this.next.copy(); return res;
```

The threefold use of `res` in this term relies on the sharing relation $\curlyvee(\texttt{b}\,|\{\texttt{b},\texttt{b},\texttt{b}\})$. Let us now consider the assignments in order. The first assignment being of scalar type is trivial. The set-view of `res.next` is `b` but so is the view of `this.next.copy()` thus justifying the second assignment. The view of `res` equals the announced return view `b` so we are done.

The body $e_{\texttt{Nil}}$ of `copy()` in `Nil` is simply `return this;`. The type of `this` is $\texttt{Nil}^{\texttt{c}}$ which is unfortunately not a subtype of the required type $\texttt{List}^{\texttt{b}}$, which we ignore here for the sake of simplicity. A proper way to avoid this problem would be to have non-unique `Nil` objects for each list, which makes sense if the `Nil`-node has a function. Otherwise one could abandon the `Nil` class and use a null pointer instead, which would somehow defeat our example. A third solution would be to include mechanisms for static objects in our theory.

Example: Destination Passing Style. We augment `List` by two methods:

```
abstract void appAux(List y, Cons dest);
List append(List y) { Cons dest = new Cons(); this.appAux(y,dest);
                      List result = dest.next; free(dest); return result; }
```

The call `this.appAux(y,dest)` to be implemented in the subclasses `Nil` and `Cons` should imperatively append `y` to `this` and place the result into the `dest.next`. The point is that `appAux` admits a tail-recursive implementation which corresponds to a while-loop.

```
/* In Nil  */      void appAux(List y, Cons dest){ dest.next <- y; }
/* In Cons */      void appAux(List y, Cons dest){ dest.next <- this;
                                                   this.next.appAux(y, this); }
```

We propose the following refined typings for these newly introduced methods:

$$\mathsf{M}(\texttt{List}^{\texttt{a}}, \texttt{append}) = \texttt{List}^{\texttt{a}} \xrightarrow{1/1} \texttt{List}^{\texttt{a}}$$
$$\mathsf{M}(\texttt{List}^{\texttt{a}}, \texttt{appAux}) = (\texttt{List}^{\texttt{a}}, \texttt{Cons}^{\texttt{n}}) \xrightarrow{0/0} \texttt{void}$$

We focus here on the most interesting judgement

$$\texttt{this:List}^{\texttt{a}}, \texttt{dest:List}^{\texttt{n}} \vdash^{\frac{0}{0}} \texttt{dest.next<-this;this.next.appAux(y,this):void}$$

Here we have decided not to glean any potential from `this` in the method body so that `this` is available as of type $\texttt{List}^{\texttt{a}}$. We split $\texttt{this:List}^{\texttt{a}}$ using $\curlyvee(\texttt{a}\,|\{\texttt{a},\texttt{n}\})$ and $\texttt{dest:List}^{\texttt{n}}$. The set-view of `dest.next` is `a` coinciding with the view of `this` thus the assignment is justified. This example shows that the potential is correctly chained through the `appAux` method despite of heavy aliasing.

Example: Doubly-Linked Lists. Our final example illustrates doubly-linked lists which brings more aliasing and even circular data.

```
abstract class DList { }
class DNil  extends DList{ }
class DCons extends DList{ Object elem; DList next;  DList previous;
                                        int getNext() { return this.next;}}
```

We would like to be able to implement methods `toList()` and `toDList()` which non-destructively transform singly-linked lists into doubly-linked ones and vice versa. To make this possible we need views on doubly-linked lists defined in such a way that the potential of a doubly-linked list is proportional to its length. This can be achieved as follows with two views `q` and `r`.

$\Diamond(\cdot)$	q	r
DList	0	0
DNil	0	0
DCons	1	0

	$\texttt{DCons}^{\texttt{q}}$	$\texttt{DCons}^{\texttt{r}}$
$\mathsf{A}^{\text{get}}(\cdot, \texttt{next})$	q	r
$\mathsf{A}^{\text{set}}(\cdot, \texttt{next})$	q	q
$\mathsf{A}^{\text{get}}(\cdot, \texttt{previous})$	r	r
$\mathsf{A}^{\text{set}}(\cdot, \texttt{previous})$	r	r

(1.3)

It is irrelevant what these views are at the other classes `Nil, Cons, List`.

The potential of a $\texttt{DList}^{\texttt{q}}$ equals its length, whereas the potential of a $\texttt{DList}^{\texttt{r}}$ is zero. The potential is defined as an infinite sum ranging over all access paths, i.e. $\boldsymbol{p} \in \{\texttt{next}, \texttt{previous}\}^*$. However, due to the fact that field `previous` has view `r` and that in `r` even the `next` attribute has view `r`, only access paths of the form $\texttt{next}^i$ for $i <$ the length of the list make a nonzero contribution.

It is now possible to include and justify in $\texttt{DList}^{\texttt{q}}$ a method that computes a singly-linked copy $\mathsf{M}(\texttt{DList}^{\texttt{q}}, \texttt{toList}) = () \xrightarrow{1/0} \texttt{List}^{\texttt{b}}$. The effect shows the cost of the additional object of type $\texttt{Nil}^{\texttt{b}}$ that is required. Similarly, a method `toDList()` can be defined.

We remark that a circular singly-linked list can be constructed, with any fixed potential unrelated to its length, e.g. of type $\texttt{List}^{\texttt{b}}$ with an overall potential 0.

Related Work. A commonly found approach to bound memory usage is the use of sized types as initially proposed by Hughes and Pareto [10]. However, as pointed out by Vasconcelos [21], these systems have difficulties, e.g. with algorithms that divide and merge their input, such as the list splitting found in the popular quick-sort algorithm: the chosen pivot could be already minimal/maximal, hence each list originating from the splitting has its size bounded by $n-1$. Merging these lists then results in an overall size of $2n-1$ instead of n and thus to an exponential size for the resulting list of the quick-sort algorithm. Our amortised analysis does not suffer from this flaw, as the potential can be properly split and merged without this kind of loss of information.

A system employing sized types for an object-oriented language is presented by Rinard et al. [3]. Their system also depends upon a deallocation primitive like ours and in addition incorporates an alias control via usage aspects. We think it is fair to say that [3] bundles together known techniques into a single system to form an actual implementation that can deal with sizeable examples. Due to the lack of worked out examples in the paper it is difficult to compare exactly the strengths and weaknesses of loc. cit. and our approach. In any case, we feel that the topic is important and new enough to justify several competing approaches for some time until it will eventually be found out which one is better.

Another widespread approach is the use of a region based memory management as initially proposed by Tofte and Talpin [19] and realized in the ML Kit Compiler [18], which primarily aims at efficient memory usage rather than obtaining provable bounds. However, Berger et al. suggest in [1] that region based approaches suffer from increased memory consumption due to retarded deallocation if the programmer is unwilling to adjust his or her programming style to suit the region approach and they propose a more generalised version of regions.

Yet another way to obtain quantity bounds on memory usage is abstract interpretation and symbolic evaluation [20, 7, 8], which aim at identifying code portions which do not affect the overall memory usage of a program. An exhaustive search of all paths of computation is then performed on the remaining abstracted code parts. However, this exhaustive search might still lead to performance problems as reported in [20], which then leads to further abstraction jeopardising provable bounds in favour of estimates.

Finally, approaches based on formal specification and theorem proving are beginning to emerge [14]. From our own experience the current state of theorem provers does not suffice to automatically prove space assertions of the kind of examples we are interested in and able to treat. However, it may be that future progress in theorem proving will eventually make analyses like ours and indeed most other program analyses redundant.

2 Featherweight Java with Update

Our formal model of Java, FJEU, is an extension of Featherweight Java (FJ) [11] with attribute update, conditional and explicit deallocation. It is thus similar to Flatt et al. Classic Java [5].

We refer to our full paper for a formal definition of its syntax and semantics and content ourselves with an informal description here.

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. Throughout the following sections we will consider a fixed (but arbitrary) class table $\mathscr{C}$ for the ease of notation.

Each class consists of a super-class, a set of attributes (or fields) with their types, and a set of methods with their types and bodies. A method body is an expression in let normal form (nested expressions flattened out using a sequence of let-definitions). Classes have only one implicit constructor that initialises all attributes to a nil-value.

An *access path* is a list of attribute names, written $a.b.\cdots.c = \boldsymbol{p}$. It is convenient to write $\mathsf{A}(C, \boldsymbol{p})$ for the class type reached by following the access path $\boldsymbol{p}$, i.e. $\mathsf{A}(C, \boldsymbol{p}.b) = \mathsf{A}(\mathsf{A}(C, \boldsymbol{p}), a)$. The typing judgement of FJEU takes the form $\Gamma \vdash e : C$ where Γ is a finite partial mapping from identifiers to class names. It is defined as a standard extension of the FJ typing rules and is omitted for lack of space.

For reasons of convenience, field update differs slightly and interdefinably from Java: the term $x.a$`<-`y evaluates to the value of x after the update rather than y as in Java.

One new feature of FJEU is the presence of an explicit deallocation construct `free`(x) that deallocates the object pointed to by x.

The typing judgement of FJEU takes the form $\Gamma \vdash e : C$ where Γ is a finite partial mapping from identifiers to class names. It is defined as a standard extension of the FJ typing rules. To illustrate FJEU we give here the corresponding version of the list copy example from Sect. 1:

```
List copy(){ let re1 = new Cons in
             let re2 = let elem = this.elem in re1.elem <- elem in
             let re3 = let next = this.next in let nres = next.copy() in
                       re2.next <- nres in return re3; }
```

The dynamic semantics of FJEU is based on a global store ("heap") mapping locations to object records as usual. We use the judgement $\eta, \sigma \vdash e \rightsquigarrow v, \tau$ shall mean that the expression e evaluates successfully to the value v, beginning with stack η, heap σ and ending with heap τ.

We also use the judgement $\eta, \sigma \overset{m}{\underset{m'}{\vdash}} e \rightsquigarrow v, \tau$ to mean that the evaluation succeeds with an initial freelist of size at least m and leaves a freelist of size at least m' upon completion.

Both judgements are given as an inductive definition which increase and decrease resource counters m, m' as expected.

Unsuccessful evaluations such as null pointer access are not modelled explicitly in the semantics. For example, when e is $\texttt{free}(\texttt{null})$ then $\eta, \sigma \overset{m}{\underset{m'}{\vdash}} e \rightsquigarrow v, \tau$ never holds. We assume that object creation `new` always returns a fresh location never seen before (and increments m by the size of the allocated object). Deallocation, on the other hand, overwrites an object record with a special value (invalid). In addition, the counter m' will be increased by the size of the deallocated object. We allow pointers to such disposed objects ("stale pointers"), however, any attempt to access a deallocated object via such a pointer leads to unsuccessful termination just as a null pointer access.

This abstract and essentially storeless [15, 2, 4, 13] semantics abstracts away from two important aspects of freelist based memory management: a) accidental "reanimation" of stale pointers through recycling of previously issued locations and b) fragmentation. Our strategy is to deal with those separately using known or orthogonal approaches.

To counter the problem with recycled locations, we can employ indirect pointers (symbolic handles) used by earlier implementations of the Sun JVM for the compacting garbage collector. Alternatively, we can statically reject programs that might access stale pointers using the alias types by Walker and Morrisett [22], or the bunched implication logic as practised by Ishtiaq and O'Hearn [12]. For those programs, our abstract semantics coincides with a concrete implementation using a freelist.

In order to deal with fragmentation in the freelist model one has several known possibilities that interact smoothly with the resource counting in the abstract semantics: allocating all objects with the same size or as linked lists of such blocks; maintaining several independent freelists for objects of various sizes (a slight change to the typing rules is then required to prevent trading objects of different sizes against each other), and, finally, compacting garbage collection. In the last case, we would simply run a compacting garbage collection as soon as the freelist does no longer contain a contiguous portion of the required size. Of course, the total memory requirement would be twice the one predicted by our analysis as usual with compacting garbage collection.

We find that the abstract operational semantics used here provides an adequate modular interface between the resource analysis and concrete implementation issues that have been and are being treated elsewhere.

3 Definition of RAJA

We now extend FJEU to an annotated version, RAJA, (Resource Aware JAva) as announced in the Introduction. A *RAJA program* is an annotation of an FJEU class table $\mathscr{C}$ or more precisely 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:

1. $\mathscr{V}$ is a possibly infinite set of views.
   For each class $C \in \mathrm{dom}(\mathscr{C})$ and for each view $r \in \mathscr{V}$ the pair C^r is called a RAJA class (or *refined type*). If C^r is a RAJA type then we denote by $|C^r| = C$ the underlying FJEU type C and by $\langle\!\langle C^r \rangle\!\rangle = r$ its view. However, we allow ourselves to omit these projections if it is clear from the context whether the view or the FJEU class is required.
2. $\Diamond(\cdot)$ assigns to each RAJA class C^r a number $\Diamond(C^r) \in \mathbb{D}$.
   This number will be used to define the potential of a heap configuration under a given static RAJA typing. For convenience, we extend the notation $\Diamond(\cdot)$ to possibly undefined meta-expressions by putting $\Diamond(\langle expr \rangle) = 0$ if $\langle expr \rangle$ is undefined.
3. $\mathsf{A}^{\mathrm{get}}(\cdot,\cdot)$ 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 $s = \mathsf{A}^{\mathrm{set}}(C^r, a)$.
   The intention is that if $D = \mathsf{A}(C, a)$ is the FJEU type of attribute a in C then the RAJA type D^q will be the type of an access to a, whereas the (intendedly stronger) type D^s must be used when updating a. The stronger typing is needed since an update will possibly affect several aliases.
4. $\mathsf{M}(\cdot,\cdot)$ assigns to each RAJA class C^r and method $m \in \mathsf{M}(C)$ having method type $E_1, \ldots, E_j \to E_0$ of arity j a j-ary polymorphic RAJA method type $\mathsf{M}(C^r, m)$.
   A *j-ary polymorphic RAJA method type* is a (possibly empty or infinite) set of j-ary monomorphic RAJA method types. A *j-ary monomorphic RAJA method type* consists of $j+1$ views and two numbers $p, q \in \mathbb{D}$, written $r_1, \ldots, r_j \xrightarrow{p/q} r_0$.
   The idea is that if m (of FJEU-type $E_1, \ldots, E_j \to E_0$) has (among others) the monomorphic RAJA method type $r_1, \ldots, r_j \xrightarrow{p/q} r_0$ then it may be called with arguments $v_1{:}E_1^{r_1}, \ldots, v_j{:}E_j^{r_j}$, whose associated potential will be consumed, as well as an additional potential of p. Upon successful completion the return value will be of type $E_0^{r_0}$ hence carry an according potential. In addition to this a potential of another q units will be gained.
   We note at this point that if a variable is to be used more than once, e.g., as an argument to a method, then the different occurrences must be given different types which are chosen such that the individual potentials assigned to each occurrence add up to the total potential available.
   We sometimes write $E_1^{r_1}, \ldots, E_j^{r_j} \xrightarrow{p/q} E_0^{r_0}$ to denote an FJEU method type combined with a corresponding monomorphic RAJA method type.

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

RAJA Subtyping Relation. We intend to define a preorder $r \sqsubseteq s$ on views as a largest fixpoint. If $\sqsubseteq_{var} \subseteq \mathscr{V} \times \mathscr{V}$ and $C <: D$ in $\mathscr{C}$ and $r, s \in \mathscr{V}$ we define

$$\mathit{Compat}(\sqsubseteq_{var}, C, D, r, s) \iff$$

$$\Diamond(C^r) \geq \Diamond(D^s) \tag{3.1}$$

$$\forall a \in \mathsf{A}(D)\,.\,\mathsf{A}^{\mathrm{get}}(C^r, a) \sqsubseteq_{var} \mathsf{A}^{\mathrm{get}}(D^s, a) \tag{3.2}$$

$$\forall a \in \mathsf{A}(D)\,.\,\mathsf{A}^{\mathrm{set}}(D^s, a) \sqsubseteq_{var} \mathsf{A}^{\mathrm{set}}(C^r, a) \tag{3.3}$$

$$\forall m \in \mathsf{M}(D)\,.\,\forall \beta \in \mathsf{M}(D^s, m)\,.\,\exists \alpha \in \mathsf{M}(C^r, m)\,.\,\alpha \sqsubseteq_{var} \beta \tag{3.4}$$

where we extend $\sqsubseteq_{var}$ to monomorphic RAJA method types as follows: if $\alpha = r_1, \ldots, r_j \xrightarrow{p/q} r_0$ and $\beta = s_1, \ldots, s_j \xrightarrow{t/u} s_0$ then $\alpha \sqsubseteq_{var} \beta$ is defined as $p \leq t$ and $q \geq u$ and $r_0 \sqsubseteq_{var} s_0$ and $s_i \sqsubseteq_{var} r_i$ for $i = 1, \ldots j$.

The subtyping relation $r \sqsubseteq s$ between views is now defined as the largest relation $\sqsubseteq$ such that

$$r \sqsubseteq s \implies \mathit{Compat}(\sqsubseteq, C, C, r, s) \text{ for all } C$$

It is easy to see that $\sqsubseteq$ is a preorder because if $\sqsubseteq_{var}$ is a preorder, so is $\forall C.\mathit{Compat}(\sqsubseteq, C, C, \cdot, \cdot)$. We extend subtyping to RAJA-classes by

$$C^r <: D^s \iff C <: D \text{ and } r \sqsubseteq s \tag{3.5}$$

It is possible to define a more fine-grained subtyping relation directly on RAJA-classes, which would in particular give the subtyping `Nil<c> <: Nil<b>` required in the copying example. We choose not to do this here because it unduly clutters notation and clarity. A practical implementation should, however, include this feature. Note that since both $\sqsubseteq$ and $<:$ on FJEU are reflexive and transitive so is $<:$ on RAJA.

Definition 1 (Sharing Relation). *We define the* sharing relation *between a single view r and a multiset of views $\mathcal{D}$ written $\curlyvee(r \,|\mathcal{D})$ as the largest relation $\curlyvee$, such that if $\curlyvee(r \,|\mathcal{D})$ then for all $C \in \mathscr{C}$:*

$$\Diamond(C^r) \geq \sum_{s \in \mathcal{D}} \Diamond(C^s) \tag{3.6}$$

$$\forall s \in \mathcal{D}\,.\,r \sqsubseteq s \tag{3.7}$$

$$\forall a \in \mathrm{dom}(\mathsf{A}(C))\,.\,\curlyvee\big(\mathsf{A}^{\mathrm{get}}(C^r, a)\,\big|\,\mathsf{A}^{\mathrm{get}}\big(C^{\mathcal{D}}, a\big)\big) \tag{3.8}$$

where $\mathsf{A}^{\mathrm{get}}\big(C^{\mathcal{D}}, a\big) = \{\mathsf{A}^{\mathrm{get}}(C^s, a) \mid s \in \mathcal{D}\}$. When $\mathcal{D} = \{s_1, \ldots, s_i\}$ is a finite multiset, we also write $\curlyvee(r \,|s_1, \ldots, s_i)$ for $\curlyvee(r \,|\mathcal{D})$. We remark that, it would be possible to define $\curlyvee(\cdot \,|\cdot)$ on the level of RAJA-classes rather than views.

Lemma 1.

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

$$\curlyvee(r \,|\{r\}) \tag{3.10}$$

$$\curlyvee(r \,|\mathcal{D}) \iff \forall \textit{ finite } \mathcal{E} \subset \mathcal{D}\,.\,\curlyvee(r \,|\mathcal{E}) \tag{3.11}$$

$$\curlyvee(r \,|\mathcal{D} \cup \{s\}) \wedge \curlyvee(s \,|\mathcal{E}) \implies \curlyvee(r \,|\mathcal{D} \cup \mathcal{E}) \tag{3.12}$$

$$r' \sqsubseteq r \wedge s' \sqsubseteq s \wedge \curlyvee(r \,|\mathcal{D} \cup \{s'\}) \implies \curlyvee(r' \,|\mathcal{D} \cup \{s\}) \tag{3.13}$$

Typing RAJA. We now give the formal definition of the RAJA-typing judgement. RAJA-typing is defined in Curry style, i.e., the terms being typed contain no RAJA-type annotations whatsoever. The intuitive meaning of the typing judgement $\Gamma \vdash^{n}_{n'} e : C^r$ has already been given in the introduction.

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

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

$$\frac{C <: E}{x{:}E^r \vdash^{0}_{0} (C)x : C^r}\ (\Diamond\textsc{Cast}) \qquad \frac{\curlyvee(s\,|q_1, q_2) \quad \Gamma, y{:}D^{q_1}, z{:}D^{q_2} \vdash^{n}_{n'} e : C^r}{\Gamma, x{:}D^s \vdash^{n}_{n'} e[x/y, x/z] : C^r}\ (\Diamond\textsc{Share})$$

$$\frac{}{\varnothing \vdash^{0}_{0} \texttt{null} : C^r}\ (\Diamond\textsc{Null}) \qquad \frac{}{x{:}C^r \vdash^{0}_{0} x : C^r}\ (\Diamond\textsc{Var})$$

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

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

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

$$\frac{n \geq u \quad n + u' \geq n' + u \quad \Theta \vdash^{u}_{u'} e : D^s \quad \forall x \in \Theta\,.\,\Gamma_x <: \Theta_x \quad D^s <: C^r}{\Gamma \vdash^{n}_{n'} e : C^r}\ (\Diamond\textsc{Waste})$$

Class Table. 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:

$$\mathsf{S}(C) = D \implies \mathit{Compat}(\sqsubseteq, C, D, r, r) \tag{3.14}$$

$$\forall a \in \mathsf{A}(C)\,.\,\mathsf{A}^{\mathrm{set}}(C^r, a) \sqsubseteq \mathsf{A}^{\mathrm{get}}(C^r, a) \tag{3.15}$$

$$\forall m \in \mathsf{M}(C)\,.\,\forall \alpha \in \mathsf{M}(C^r, m)\,.\,\exists q, s \in \mathscr{V}\,.\,\curlyvee(r\,|q, s) \quad \wedge$$
$$\texttt{this}{:}C^q, x_1{:}E_1^{r_1}, \ldots, x_j{:}E_j^{r_j} \vdash^{n + \Diamond(C^s)}_{n'} \mathsf{M}_{\mathsf{body}}(C, m) : E_0^{r_0} \tag{3.16}$$

where $C.m = E_1, \ldots, E_j \to E_0$ and $\alpha = r_1, \ldots, r_j \xrightarrow{n/n'} r_0$

4 Main Result

Our main result involves the following concepts which we explain informally here; the full version contains formal definitions and motivations. We write $\llbracket (v{:}r).\boldsymbol{p} \rrbracket^{\mathrm{stat}}_{\sigma}$ for the view on the object reached from v (of view r) via access path $\boldsymbol{p}$ when accessed in this way. For example, if v points to a doubly-linked list of length 2 in σ then $\llbracket (v{:}\mathtt{q}).\mathtt{next.next} \rrbracket^{\mathrm{stat}}_{\sigma} = \mathtt{q}$.

We write $\Phi_\sigma(v : r)$ and $\Phi_\sigma(\eta : \Gamma)$ for the potentials of the data structures reachable from v, resp. η when viewed through r, resp., Γ. For example, $\Phi_\sigma(v : r) = \sum_{\boldsymbol{p}} \Diamond(D^s)$, where D is the dynamic type of the record reached from v in σ via $\boldsymbol{p}$, whereas $s = \mathbb{\|}(v{:}r).\boldsymbol{p}\mathbb{\|}^{\mathrm{stat}}_\sigma$.

Finally, if Γ is a RAJA typing context with underlying FJEU context $|\Gamma|$, we write $\sigma \vDash \eta : \Gamma$ to mean that $\sigma \models \eta : |\Gamma|$ and, moreover, for each location ℓ reachable from η there exists a view r (its *proto-view*) such that $\curlyvee(r \,|\mathscr{V}_{\sigma,\eta,\Gamma}(\ell))$ where $\mathscr{V}_{\sigma,\eta,\Gamma}(\ell)$ is the multiset consisting of all assumable views on location ℓ by σ, η, Γ, formally, $\mathscr{V}_{\sigma,\eta,\Gamma}(\ell) = \{\, \mathbb{\|}(\eta_x{:}\langle\!\langle \Gamma_x \rangle\!\rangle).\boldsymbol{p}\mathbb{\|}^{\mathrm{stat}}_\sigma \mid x \in \Gamma, [\![\eta_x.\boldsymbol{p}]\!]_\sigma = \ell\}$.

This definition is a crucial invariant needed in the proof of the main result; it does not appear in the corollary intended for end users.

Theorem 1. *Fix a well-typed RAJA program $\mathscr{R}$. If*

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

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

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

then

$$\eta, \sigma \overset{\circ}{\underset{n' + \Phi_\tau(v:r) + \Phi_\tau(\eta:\Delta)}{\vdash^{n + \Phi_\sigma(\eta:\Gamma) + \Phi_\sigma(\eta:\Delta)}}} e \rightsquigarrow v, \tau \tag{4.1}$$

$$\tau \vDash \eta[x_{res} \mapsto v] : (\Delta, x_{res}{:}C^r) \tag{4.2}$$

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

Proof. (Sketch) The proof is by induction on the operational semantics and a subordinate induction on typing derivations. Several of the cases present interesting difficulties. To give a flavour of the proof we sketch the case of field update here where we essentially have to show that a field update leaves the total potential unchanged and that newly created aliases admit a "proto-view". We describe the crucial observation to give a flavour of the proof. Suppose that at runtime the update is $\ell.a := v$, ℓ being a location, a a field, v a value. For locations ℓ_1, ℓ_2 let $P(\ell_1, \ell_2)$ stand for access paths from ℓ_1 to ℓ_2 and $Q(\ell_1, \ell_2$ stand for the subset of $P(\ell_1, \ell_2)$ consisting of those paths that do not go through $\ell.a$ and are thus unaffected by the update. After the update we have

$$P(\ell_1, \ell_2) = Q(\ell_1, \ell_2) + Q(\ell_1, \ell)(aQ(v, \ell))^* aQ(v, \ell_2)$$

since any access path either does not meet the updated location at all or goes through it a finite number of times. Since the right hand side of this identity comprises paths that are not affected by the update, information about it can be obtained from the assumptions that describe the situation before the update. A number of technical lemmas about the sharing relation and potentials are of course needed to flesh this out.

The following corollary is a direct consequence of the main result and it is in this form that we intend to use it. The apparently clumsy form of the main result is needed in order to enable an inductive proof.

Corollary 1. *Suppose that $\mathscr{C}$ is an FJEU program containing (in Java notation) a class* `List` *of singly-linked lists with boolean entries, a class* `C` *containing a method* `void C.main(List args)`*, and arbitrary other classes and methods.*

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

5 Conclusion

We have presented a generic method for using potentials in the sense of amortised complexity to count memory allocations and deallocations. Our method allows for input dependent analysis without explicitly manipulating size expressions. This sets it apart against more direct methods based on sized types. We have stated and proved a nontrivial soundness property which shows that our typing rules for sharing correctly account for aliased and even circular data.

Inference. We have not studied the problem of view inference and not even algorithmic type checking since these two tasks are independent of soundness which was our main concern here. But of course inference and automatic type checking are of paramount importance for the viability of our method. We therefore briefly comment on how we plan to attack these issues.

First, we remark that if the structure of the views, i.e., the views without their potential annotations are known, the latter quantities can be efficiently found by LP-solving as was done in the precursor of this work [9]. Indeed, the system presented there can be faithfully embedded into RAJA and for this fragment automatic inference is unproblematic.

Likewise, the intermediate views that do not appear in class tables but only within method bodies typically take the form of fragments of already declared views in the sense that some fields are set to a zero view like `n` in Sect. 1. We are confident that these views can be generated automatically and on the fly during algorithmic type checking for example by reformulating the sharing rule in an algorithmic fashion.

A simple kind of view polymorphism should also be within reach if one applies the generic type and effect discipline [16] to our system.

Going beyond these low-hanging fruit will probably require the isolation of several fragments or high-level systems built on top of RAJA, supporting for example particular styles or patterns of programming.

Lastly, we mention that the possible access paths emanating from each class define an infinite regular tree, e.g., the tree consisting of the paths in $\texttt{next}^*(\texttt{elem} \cup \{\epsilon\})$ in the case of `Cons`. The set of views on a class in a (finite!) RAJA program defines a regular tiling of that tree and can perhaps be found using automata- or language-theoretic methods.

Extensions. Our focusing on heap space usage was a rather arbitrary choice. We believe that by slight modifications we can use the amortised method for other quantitative resources such as stack size, multiple freelists, number of open files, etc., in a similar fashion.

Limitations. Rule $\Diamond$Update contains a source of possible over-approximation because it does not restitute the potential contained in the overwritten data. This can lead to sound but not typable examples the simplest of which is as follows: if a is an object with a field f whose get- and set-types differ then $a.f$<-$a.f$ is not typable yet obviously sound since its effect is zero.

Another limitation of the system stems from the fact that object types do not change after a method invocation. This is mediated by the linear formulation of the type system: after a call $x.m()$ the reference x with its type is "used up"; a further invocation of a method on the object referenced by x can only happen through a prior invocation of $\Diamond$Share and hence in general with a different type. Nevertheless, exploring type change after method invocation could be worthwhile.

We also note that our method estimates resource usage as a function of the input. Thus, programs whose resource usage depends on other parameters cannot be analysed. A concrete example is the numerical solution of a boundary value problem by solving successively larger and larger linear systems of equations.

Other limitations stem from the type inference problem. While it is in many cases possible to find a typing it might be difficult to come up with an inference scheme that encompasses those. On a positive side we note that the earlier system by the authors [9] can be faithfully mapped into the present system.

Acknowledgements. We thank Olha Shkaravska for pointing out the relationship of our previous work [9] with amortised complexity. We had been teaching the latter for years but failed to see the connection. We also thank Peter O'Hearn for a long and lively phone conversation on the topic of operational semantics of "free". The authors acknowledge financial support by the GKLI Munich and the EU FET-IST projects IST-510255 (EmBounded), IST-15905 (Mobius), IST-2001-33149 (MRG).

References

1. E. D. Berger, B. G. Zorn, and K. S. McKinley. Reconsidering custom memory allocation. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, 2002.
2. M. Bozga, R. Iosif, and Y. Laknech. Storeless semantics and alias logic. In *Proceedings of the 2003 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation (PEPM)*, pages 55–65. ACM, 2003.
3. W.-N. Chin, H. H. Nguyen, S. Qin, and M. Rinard. Memory usage verification for oo programs. In *The 12th International Static Analysis Symposium (SAS)*. LNCS, Sep 2005.
4. A. Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. *ACM SIGPLAN Notices*, 29(6):230–241, 1994.

5. M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 171–183, 1998.
6. A. Galland and M. Baudet. Controlling and Optimizing the Usage of One Resource. In A. Ohori, editor, *1st Asian Symposium on Programming Languages and Systems (APLAS)*, volume 2895 of *Lecture Notes in Computer Science*, pages 195–211, Beijing, China, Nov. 27-29, 2003. Springer-Verlag.
7. G. Gómez and Y. A. Liu. Automatic time-bound analysis for a higher-order language. In *Proceedings of the 2002 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, pages 75–86. ACM Press, 2002.
8. B. Grobauer. *Topics in Semantics-based Program Manipulation.* PhD thesis, BRICS Aarhus, 2001.
9. M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 185–197. ACM, 2003.
10. J. Hughes and L. Pareto. Recursion and dynamic data structures in bounded space: towards embedded ML programming. In *Proc. International Conference on Functional Programming (ICFP). Paris, September 1999.*, pages 70–81, 1999.
11. A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In L. Meissner, editor, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, volume 34(10), pages 132–146, N.Y., 1999.
12. S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 14–26. ACM, 2001.
13. H. Jonkers. Abstract storage structures. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 321–343. IFIP, North Holland, 1981.
14. J. Krone, W. F. Ogden, and M. Sitaraman. Modular verification of performance constraints. Technical report, Dep. of Comp. Sci., Clemson Univeristy, May 2003.
15. N. Rinetzky, J. Bauer, T. Reps, M. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 296–309, New York, NY, USA, 2005. ACM Press.
16. J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *J. Funct. Program.*, 2(3):245–271, 1992.
17. R. E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1985.
18. M. Tofte, L. Birkedal, M. Elsman, N. Hallenberg, T. Olesen, and P. Sestoft. Programming with regions in the ml kit, April 2002. IT University of Copenhagen `http://www.itu.dk/research/mlkit/`.
19. M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.
20. L. Unnikrishnan, S. D. Stoller, and Y. A. Liu. Automatic accurate live memory analysis for garbage-collected languages. In *Proceedings of The Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES)*. ACM, 2001.
21. P. Vasconcelos. *Space Cost Modeling for Concurrent Resource Sensitive Systems.* PhD thesis, School of Comp. Sci., University of St Andrews, Scotland, to appear.
22. D. Walker and G. Morrisett. Alias types for recursive data structures. *Lecture Notes in Computer Science*, 2071:177+, 2001.