Memory Usage Verification for OO Programs[⋆]

Wei-Ngan Chin[1,2], Huu Hai Nguyen[1], Shengchao Qin[3], and Martin Rinard[4]

[1] Computer Science Programme, Singapore-MIT Alliance
[2] Department of Computer Science, National University of Singapore
[3] Department of Computer Science, University of Durham
[4] Laboratory for Computer Science, Massachusetts Institute of Technology
`{chinwn,nguyenh2}@comp.nus.edu.sg`
`shengchao.qin@durham.ac.uk, rinard@lcs.mit.edu`

Abstract. We present a new type system for an object-oriented (OO) language that characterizes the sizes of data structures and the amount of heap memory required to successfully execute methods that operate on these data structures. Key components of this type system include type assertions that use symbolic Presburger arithmetic expressions to capture data structure sizes, the effect of methods on the data structures that they manipulate, and the amount of memory that methods allocate and deallocate. For each method, we conservatively capture the amount of memory required to execute the method as a function of the sizes of the method's inputs. The safety guarantee is that the method will never attempt to use more memory than its type expressions specify. We have implemented a type checker to verify memory usages of OO programs. Our experience is that the type system can precisely and effectively capture memory bounds for a wide range of programs.

1 Introduction

Memory management is a key concern for many applications. Over the years researchers have developed a range of memory management approaches; examples include explicit allocation and deallocation, copying garbage collection, and region-based memory allocation. However, an important aspect that has been largely ignored in past work is the safe estimation of memory space required for program execution. Overallocation of memory may cause inefficiency, while underallocation may cause software failure. In this paper, we attempt to make memory usage more predictable by static verification on the memory usage of each program.

We present a new type system, based on dependent type[21], that characterizes the amount of memory required to execute each program component. The key components of this type system include:

- **Data Structure Sizes and Size Constraints:** The type of each data structure includes index parameters to characterize its size properties, which are expressed in terms of the sizes of data structures that it contains. In many cases the sizes of these data structures are correlated; our approach uses size constraints expressed using symbolic Presburger arithmetic terms to precisely capture these correlations.
- **Heap Recovery:** Our type system captures the distinction between shared and unaliased objects and supports explicit deallocation of unaliased objects.

[⋆] slightly revised from SAS'05 version

– **Preconditions and Postconditions:** Each method comes with a precondition that captures both the expected sizes of the data structures on which it operates and any correlations between these sizes. The method's postcondition expresses the new size and correlations of these data structures after the method executes as a function of the original sizes when the method was invoked.
– **Heap Usage Effects:** Each method comes with two memory effects. These effects use symbolic values (present in method precondition) to capture (i) *memory requirement* which specify the maximum heap space that the method *may* consume, (ii) *memory release* which specify the minimum heap space that the method *will* recover. Heap effects are expressed at the granularity of classes and can capture the net change in the number of instances of each class.

Our paper makes several new technical contributions. Firstly, we design a formal verification system in the form of a type system, that can *formally* and *statically* capture memory usage for the object-oriented (OO) paradigm. We believe that ours is the first such formal type system for OO paradigm. Secondly, we advocate for *explicit heap recovery* to provide more timely reclamation of dead objects in support of tighter bounds on memory usage. We show how such recovery commands may be automatically inserted. Thirdly, we have proven the soundness of our type checking rules. Each well-typed program is guaranteed to meet its memory usage specification, and will *never fail due to insufficient memory* whenever its memory precondition is met. Lastly, we have implemented a type checker and have shown that it is fairly precise and can handle a reasonably large class of programs. Runtime **stack space** to hold methods' parameters and local variables is another aspect of memory needed. For simplicity, we omit its consideration in this paper.

2 Overview

Memory usage occurs primarily in the heap to hold dynamically created objects. In our model, heap space is consumed via the `new` operation for newly created objects, while unused objects may be recovered via an explicit deallocation primitive, called `dispose`. Memory usage (based on consumption and recovery) should be calculated over the entire computation of each program. This calculation is done in a safe manner to help identify the high watermark on memory space needed. We achieve this through the use of a conservative upper bound on memory consumed, and a conservative lower bound on memory recovered for each expression (and method).

To safely predict the memory usage of each program, we propose a *size-polymorphic type system* for object-oriented programs with support for interprocedural size analysis. In this type system, size properties of both user-defined types and primitive types are captured. In the case of primitive integer type $\texttt{int}\langle v\rangle$, the size variable v captures its integer value, while for boolean type $\texttt{bool}\langle b\rangle$, the size variable b is either 0 or 1 denoting `false` or `true`, respectively. (Note that size variables capture some integer-based properties of the data structure. For simple types, the values are directly captured.) For user-defined class types, we use $c\langle n_1,\ldots,n_p\rangle$ $\texttt{where}$ ϕ ; ϕ_I with size variables $n_1,\ldots,n_p$ to denote size properties that are defined in size relation ϕ, and invariant constraint ϕ_I. As an example, consider a user-defined stack class, that is implemented with a linked list, and a binary tree class as shown below.

```
class List⟨n⟩ where n=m+1 ; n≥0 { Object⟨⟩@S val; List⟨m⟩@U next; ··· }
class Stack⟨n⟩ where n=m ; n≥0 { List⟨m⟩@U head; ··· }
class BTree⟨s, d⟩ where s=1+s1+s2∧d=1+max(d1, d2) ; s≥0∧d≥0 {
  Object⟨⟩@S val; BTree⟨s1, d1⟩@U left; BTree⟨s2, d2⟩@U right; ··· }
```

List$\langle n\rangle$ denotes a linked-list data structure of size n, and similarly for Stack$\langle n\rangle$. The size relations $n{=}m{+}1$ and $n{=}m$ define some size properties of the objects in terms of the sizes of their components, while the constraint $n{\geq}0$ signifies an invariant associated with the class type. Class BTree$\langle s, d\rangle$ represents a binary tree with size variables s and d denoting the total number of nodes and the depth of the tree, respectively. Due to the need to track the states of mutable objects, our type system requires the support of alias controls of the form A=U | S | R | L. We use U and S to mark each reference that is (definitely) *unaliased* and (possibly) *shared*, respectively. We use R to mark read-only fields which must never be updated after object initialization. We use L to mark unique references that are temporarily borrowed by a parameter for the duration of its method's execution. Our alias annotation mechanism are adapted from [5, 8, 1] and reported in [9]. Briefly, they allow us to track unique objects from mutable fields, as well as shareable objects from read-only fields.

To specify memory usage, we decorate each method with the following declaration:

$$t\ mn\ (t_1 v_1, \ldots, t_n v_n)\ \texttt{where}\ \ \phi_{pr}; \phi_{po}; \epsilon_c; \epsilon_r\ \{e\}$$

where ϕ_{pr} and ϕ_{po} denote the precondition and postcondition of the method, expressed in terms of constraints/formulae on the size variables of the method's parameters and result. Precondition ϕ_{pr} denotes an applicability condition of the method in terms of the sizes of its parameters. Postcondition ϕ_{po} can provide a precise size relation for the parameters and result of the declared method. The memory effect is captured by ϵ_c and ϵ_r. Note that ϵ_c denotes *memory requirement*, i.e., the maximum memory space that *may be consumed*, while ϵ_r denotes *net release*, i.e., the minimum memory space that *will be recovered* at the end of method invocation. Memory effects (consumption and recovery) are expressed using a bag notation of the form $\{(c_i, \alpha_i)\}_{i=1}^{m}$, where c_i denotes a class type, while α_i denotes its symbolic count.

```
class Stack⟨n⟩ where n=m ; n≥0 { List⟨m⟩@U head;
L ∥ void⟨⟩@S push(Object⟨⟩@S o) where true; n′=n+1; {(List, 1)}; {}
  { List⟨⟩@U tmp=this.head; this.head=new List(o, tmp)}
L ∥ void⟨⟩@S pop() where n>0; n′=n−1; {}; {(List, 1)}
  { List⟨⟩@U t1 = this.head; List⟨⟩@U t2 = t1.next; t1.dispose(); this.head = t2}
L ∥ bool⟨b⟩@S isEmpty() where n≥0; n′=n ∧ (n=0∧b=1 ∨ n>0∧b=0); {}; {}
  { List⟨⟩@U t = this.head; bool⟨⟩@S v = isNull(t); this.head = t; v}
 L ∥ void⟨⟩@S emptyStack() where n≥0∧d=n; n′=0; {}; {(List, d)}
  { bool⟨⟩@S v = this.isEmpty(); if v then () else {this.pop(); this.emptyStack()}}
L ∥ void⟨⟩@S push3pop2(Object⟨⟩@S o) where true; n′=n+1; {(List, 2)}; {(List, 1)}
  { this.push(o); this.push(o); this.pop(); this.push(o); this.pop()}}
```

Fig. 1. Methods for the Stack Class

Examples of method declarations for the `Stack` class are given in Fig 1. The notation (A ‖) prior to each method captures the alias annotation of the current `this` parameter. Note our use of the primed notation, advocated in [13, 16], to capture imperative changes on size properties. For the `push` method, $n'{=}n{+}1$ captures the fact that the size of the stack object has increased by 1; similarly, the postcondition for the `pop` method, $n'{=}n{-}1$, denotes that the size of the stack is decreased by 1 after the operation. The memory requirement for the `push` method, $\epsilon_r{=}\{(\texttt{List}, 1)\}$, captures the fact that one `List` node will be consumed. For the `pop` method, $\epsilon_r{=}\{(\texttt{List}, 1)\}$ indicates that one `List` node will be recovered.

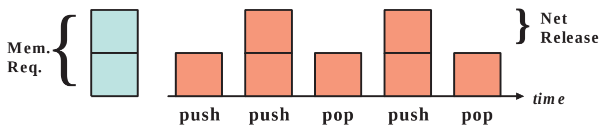

Fig. 2. `push3pop2`: Heap Consumption and Recovery

For the `isEmpty` method, $n'{=}n$ captures the fact that the size of the receiver object (`this`) is not changed by the method. Furthermore, its output of type `bool`$\langle b\rangle$`@S` is related to the object's size through a disjunctive constraint $n{=}0{\wedge}b{=}1{\vee}n{>}0{\wedge}b{=}0$. Primitive types are annotated with alias `S` because their values are immutable and can be freely shared and yet remain trackable. The `emptyStack` method releases all `List` nodes of the `Stack` object. For `push3pop2` method, the memory consumed (or required) from the heap is $\{(\texttt{List}, 2)\}$, while the net release is $\{(\texttt{List}, 1)\}$, as illustrated in Fig. 2.

Size variables and their constraints are specified at method boundary, and need not be specified for local variables. Hence, we may use `bool`$\langle\rangle$`@S` instead of `bool`$\langle v\rangle$`@S` for the type of a local variable.

3 Language and Annotations

We focus on a core object-oriented language, called MEMJ, with size, alias, and memory annotations in Fig 3. MEMJ is designed to be an intermediate language for Java with either supplied or inferred annotations. A suffix notation y^* denotes a list of zero or more distinct syntactic terms that are suitably separated. For example, $(t\ v)^*$ denotes $(t_1\ v_1, \ldots, t_n\ v_n)$ where $n{\geq}0$. Local variable declarations are supported by block structure of the form: $(t\ v = e_1; e_2)$ with e_2 denoting the result. We assume a call-by-value semantics for MEMJ, where values (primitives or references) are passed as arguments to parameters of methods. For simplicity, we do not allow the parameters to be updated (or re-assigned) with different values. There is no loss of generality, as we can always copy such parameters to local variables for updating.

The MEMJ language is deliberately kept simple to facilitate the formulation of static and dynamic semantics. Typical language constructs, such as multi-declaration block, sequence, calls with complex arguments, *etc.* can be automatically translated to constructs in MEMJ. Also, loops can be viewed as syntactic abbreviations for tail-recursive methods, and are supported by our analysis. Several other language features, including downcast and a field-binding construct are also supported in our implementation. For simplicity, we omit them in this paper, as they play supporting roles and are not

$$
\begin{array}{rl}
P & ::= \mathit{def}^* \ \mathit{meth}^* \\
\mathit{def} & ::= \mathtt{class}\ c_1\langle n_{1..p}\rangle\ [\,\mathtt{extends}\ c_2\langle n_{1..q}\rangle\,]\ \mathtt{where}\ \phi\ ;\ \phi_I\ \{\ \mathit{fd}^*\ (A \parallel \mathit{meth})^*\ \} \\
\mathit{meth} & ::= t\ \mathit{mn}\ ((t\ v)^*)\ \mathtt{where}\ \ \phi_{pr}; \phi_{po}; \epsilon_c; \epsilon_r\ \{e\}
\end{array}
$$

$$
\begin{array}{lll}
\mathit{fd} ::= t \quad f & t ::= \tau\ \langle n^*\rangle @A & A ::= \mathtt{U}\ |\ \mathtt{L}\ |\ \mathtt{S}\ |\ \mathtt{R} \\
\tau ::= c\ |\ \mathit{pr} & w ::= v\ |\ v.f & \mathit{pr} ::= \mathtt{int}\ |\ \mathtt{bool}\ |\ \mathtt{void}
\end{array}
$$

$$
\begin{array}{rl}
e ::= & (c)\,\mathtt{null}\ |\ k\ |\ w\ |\ w = e\ |\ t\ v = e_1\ ;\ e_2\ |\ \mathtt{new}\ c(v^*) \\
& |\ v.\mathit{mn}\,(v^*)\ |\ \mathit{mn}\,(v^*)\ |\ \mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2\ |\ v.\mathtt{dispose}()
\end{array}
$$

$$
\begin{array}{ll}
\epsilon = \{(c,\alpha)^*\} & \textit{(Memory Space Abstraction)} \\
\phi \in \ \mathbf{F} & \textit{(Presburger Size Constraint)} \\
\quad ::= \ b \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi \mid \exists n \cdot \phi \mid \forall n \cdot \phi & \\
b \in \ \mathbf{BExp} & \textit{(Boolean Expression)} \\
\quad ::= \mathtt{true} \mid \mathtt{false} \mid \alpha_1 = \alpha_2 \ \mid \alpha_1 < \alpha_2 \mid \alpha_1 \leq \alpha_2 & \\
\alpha \in \ \mathbf{AExp} & \textit{(Arithmetic Expression)} \\
\quad ::= k^{\mathtt{int}} \mid n \mid k^{\mathtt{int}} * \alpha \mid \alpha_1 + \alpha_2 \mid -\alpha \mid \mathit{max}(\alpha_1,\alpha_2) \mid \mathit{min}(\alpha_1,\alpha_2) &
\end{array}
$$

$$
\textit{where } k^{\mathtt{int}} \in Z \textit{ is an integer constant};\ \ n \in \mathit{SV} \textit{ is a size variable}
$$

$$
f \in \mathit{Fd} \textit{ is a field name};\ \ v \in \mathit{Var} \textit{ is an object variable}
$$

Fig. 3. Syntax for the MEMJ Language

core to the main ideas proposed here. The interested reader may refer to our companion technical report[10] for more information.

To support sized typing, our programs are augmented with size variables and constraints. For size constraints, we restrict to Presburger form, as decidable (and practical) constraint solvers exist, e.g. [19]. We are primarily interested in tracking size properties of objects. We therefore restrict the relation ϕ in each class declaration of $c_1\langle n_1, .., n_p\rangle$ which extends $c_2\langle n_1, .., n_q\rangle$ to the form $\bigwedge_{i=q+1}^{p} n_i{=}\alpha_i$ whereby $V(\alpha_i) \cap \{n_1, .., n_p\} = \emptyset$. Note that $V(\alpha_i)$ returns the set of size variables that appeared in α_i. This restricts size properties to depend solely on the components of their objects.

Note that each class declaration has a set of instance methods whose main purpose is to manipulate objects of the declared class. For convenience, we also provide a set of static methods with the same syntax as instance methods, except for its access to the `this` object. One important feature of MEMJ is that memory recovery is done safely (without creating dangling references) through a $v.\mathtt{dispose}()$ primitive.

4 Heap Usage Specification

To allow memory usage to be precisely specified, we propose a bag abstraction of the form $\{(c_i, \alpha_i)\}_{i=1}^{n}$ where c_i denotes its classification, while α_i is its cardinality. In this paper, we shall use $c_i \in \mathit{CN}$ where CN denotes all class types. For instance, $\Upsilon_1 = \{(c_1, 2), (c_2, 4), (c_3, \mathtt{x} + 3)\}$ denotes a bag with c_1 occurring twice, c_2 four times and c_3 $\mathtt{x} + 3$ times. We provide the following two basic operations for bag abstraction to capture both the domain and the count of its element, as follows:

$$
\mathit{dom}(\Upsilon) =_{df} \{c \ \mid\ (c, n) \in \Upsilon\} \qquad\qquad \Upsilon(c) =_{df} \begin{cases} n, & \mathit{if}\ (c, n) \in \Upsilon \\ 0, & \mathit{otherwise} \end{cases}
$$

We define union, difference, exclusion over bags as:

$$\begin{array}{lll} \Upsilon_1 \uplus \Upsilon_2 & =_{df} & \{(c, \Upsilon_1(c){+}\Upsilon_2(c)) \mid c \in \mathit{dom}(\Upsilon_1) \cup \mathit{dom}(\Upsilon_2)\} \\ \Upsilon_1 - \Upsilon_2 & =_{df} & \{(c, \Upsilon_1(c){-}\Upsilon_2(c)) \mid c \in \mathit{dom}(\Upsilon_1) \cup \mathit{dom}(\Upsilon_2)\} \\ \Upsilon \setminus X & =_{df} & \{(c, \Upsilon(c)) \mid c \in \mathit{dom}(\Upsilon) - X\} \end{array}$$

To check for adequacy of memory, we provide a bag comparator operation under a size constraint Δ, as follows:

$$\Delta \vdash \Upsilon_1 \sqsupseteq \Upsilon_2 =_{df} (\Delta \Rightarrow (\forall c \in Z \cdot \Upsilon_1(c) \geq \Upsilon_2(c))) \ \mathtt{where}\ Z = \mathit{dom}(\Upsilon_1) \cup \mathit{dom}(\Upsilon_2)$$

The bag abstraction notation for memory is quite general and can be made more precise by refining its operations. For example, some class types are of the same size and could replace each other to increase memory reuse. To achieve this we can use a bag abstraction that is grouped by $\mathit{size}(c_i)$ instead of class type c_i.

4.1 Heap Consumption

Heap space is consumed when objects are created by the `new` primitive, and also by method calls, except that the latter is aggregated to include recovery prior to consumption. Our aggregation (of recovery prior to consumption) is designed to identify a high watermark of maximum memory needed for safe program execution. For each expression, we predict a conservative upper bound on the memory that the expression *may* consume, and also a conservative lower bound on the memory that the expression *will* release. If the expression releases some memory before consumption, we will use the released memory to obtain a lower memory requirement. Such aggregated calculations on both consumption and recovery can help capture both a net change in the level of memory, as well as the high watermark of memory needed for safe execution.

For example, consider a recursive function which does p pops from one stack object, followed by the same number of pushes on another stack.

```
void⟨⟩@S moverec(Stack⟨a⟩@L s, Stack⟨b⟩@L t, int⟨p⟩@S i)
   where a≥p≥0; a′=a−p∧b′=b+p; {} ; {}
 { if i<1 then ()
   else {Object⟨⟩@S o = s.top(); s.pop(); moverec(s, t, i−1); t.push(o)} }
```

Due to aggregation (involving recovery before consumption), the heap space that may be consumed is zero. For each recursive call, the space for a `List` node is released by `s.pop()` before it is reused by `t.push(o)`. Aggregated over the recursive calls, we will have p number of `List` nodes that have been released before the same number of nodes are consumed. Hence, no new heap space is needed. Such aggregation is sensitive to the order of the operations.

Consider now a different function which performs p pushes on `t`, followed by the same number of pops from `s`.

```
void⟨⟩@S moverec2(Stack⟨a⟩@L s, Stack⟨b⟩@L t, int⟨p⟩@S i)
   where a≥p≥0; a′=a−p∧b′=b+p; {(List, p)}; {(List, p)}
 { if i<1 then ()
   else {Object⟨⟩@S o = s.top(); t.push(o); moverec2(s, t, i−1); s.pop()} }
```

Though the net change in memory usage is also zero, the memory effect for this function is different as we require p number of `List` nodes to be consumed on entry, *before* the same number of `List` nodes are recovered. This new memory effect has the potential to push up the high watermark of memory needed by p `List` nodes.

4.2 Heap Recovery

Explicit heap space recovery via `dispose` has several advantages. It facilitates the timely recovery of dead objects, which allows memory usage to be predicted more accurately (with tighter bounds). It also permits the use of more efficient custom allocators[4], where desired. Moreover, we shall provide an automatic technique to insert `dispose` primitives with the help of alias annotation. With such a technique, we only need to ensure that objects that are being disposed are non-null. This non-nullness property can be captured by a non-nullness analyser, such as [12]. This property is required as we always recover memory space for each `dispose` primitive.

Memory recovery via `dispose` should occur when unique references that are still alive (not in dead-set) are being discarded. This could occur at four places[1] : (i) end of local block, (ii) end of method block, (iii) prior to assignment operation, and (iv) at conditional expression. We would like to recover the memory space for each non-null reference that is about to become dead. For example, consider the `pop` method's definition:

$\texttt{L} \mid \texttt{void}\langle\rangle\texttt{@S pop}()\ \texttt{where}\ \cdots\ \{\texttt{List}\langle\rangle\texttt{@U t1} = \texttt{this.head};\ \texttt{head} = \texttt{t1.next}\}$

The object pointed to by `head` is about to become dead prior to the operation, $\texttt{head} = \texttt{t1.next}$. To recover this dead object, we insert a `dispose` command to obtain $\texttt{head} = (\texttt{t1.next} <; \texttt{head.dispose}())$ where $e_1 {<;} e_2 \equiv (t\ v = e_1; e_2; v)$. Consider the definition of the `destroy` method which calls `emptyStack` with an L-mode parameter.

$\texttt{void}\langle\rangle\texttt{@S destroy}(\texttt{Stack}\langle n\rangle\texttt{@U s})\ \texttt{where}\ \cdots\ \{\texttt{emptyStack}(\texttt{s})\}$

A unique `s` object is about to become dead at the end of the `destroy` method. To recover this space, we can insert $\texttt{s.dispose}()$ prior to the method's exit.

Let us formalise an automatic technique for the explicit recovery of dead objects that are known at compile-time. Given an expression e, we utilize the alias annotation to obtain a new expression e_1 where suitable explicit heap `dispose` operations have been safely inserted. This is achieved by a translation below with Γ to denote a type environment mapping program variables to their annotated types, and $\Theta(\Theta_1)$ to denote the set of dead references (of the form v or $v.f$) before (after) the evaluation of expression e.

$$\Gamma; \Theta \vdash e \hookrightarrow_H e_1 :: t, \Theta_1$$

Most rules are structure-preserving (or identity) rewritings, except for four rules given in Fig 4. A sequence of disposals can be effected through *dispose*(D), with D containing a set of variable/field references that are about to be dead at the end of expression e.

For the assignment rule [H:ASSIGN], we add w to the disposal set if it is unique and is not yet in dead-set using $D = \{w \mid \mathit{ann}(t){=}\texttt{U}\} {-} \Theta_1$. The function *isParam*$(w)$ returns `true` if w is a parameter variable, otherwise it returns `false` (for fields and local variables). The function *ann* extracts the alias of an annotated type, $\mathit{ann}(\tau\langle v^*\rangle@A) = A$. A

[1] Note that unique reference cannot escape through e_1 in $e_1; e_2$ as we require e_1 to be of the `void` type.

$$\frac{\begin{array}{c}[\textsc{h:assign}]\\ \neg\ \mathit{isParam}(w)\quad \Gamma(w)=t\\ D=\{w \mid \mathit{ann}(t)=\mathtt{U}\}-\Theta_1\\ \Gamma;\Theta\vdash e\hookrightarrow_H e_1::t_1,\Theta_1\\ \vdash t_1<:t\\ e_2=(e_1\lhd D{=}\emptyset\rhd e_1<;\mathit{dispose}(D))\end{array}}{\begin{array}{l}\Gamma;\Theta\vdash w=e\hookrightarrow_H\\ \qquad w=e_2::\mathtt{void@S},\Theta_1\backslash w\end{array}}$$

$$\frac{\begin{array}{c}[\textsc{h:if}]\\ \Gamma(v)=\mathtt{bool}\langle b\rangle\mathtt{@S}\\ \Gamma;\Theta\vdash e_i\hookrightarrow_H \hat{e}_i::t_i,\Theta_i\ \ i=1,2\\ t=\mathit{msst}(t_1,t_2)\quad \Theta_3=\Theta_1\cup\Theta_2\\ D_i=\Theta_3{-}\Theta_i\ \ i=1,2\\ E_i=(\hat{e}_i\lhd D_i{=}\emptyset\rhd \hat{e}_i<;\mathit{dispose}(D))\ \ i=1,2\end{array}}{\begin{array}{l}\Gamma;\Theta\vdash \mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2\hookrightarrow_H\\ \qquad \mathtt{if}\ v\ \mathtt{then}\ E_1\ \mathtt{else}\ E_2::t,\Theta_3\end{array}}$$

$$\frac{\begin{array}{c}[\textsc{h:meth}]\\ \Gamma_1=\Gamma+\{v_1::t_1,..,v_p::t_p\}\\ \Gamma_1;\emptyset\vdash e\hookrightarrow_H e_1::t,\Theta\\ \vdash t<:t_0\quad \mathit{ann}(t_0)\neq\mathtt{L}\\ \forall i{\in}1..p\cdot(\mathit{ann}(t_i)=\mathtt{L}){\Rightarrow}(\forall f\cdot v_i.f{\notin}\Theta)\\ D=\{w\mid(w::t)\in\Gamma_1,\mathit{ann}(t){=}\mathtt{U}\}{-}\Theta\\ e_2=(e_1\lhd D{=}\emptyset\rhd e_1<;\mathit{dispose}(D))\end{array}}{\begin{array}{l}\Gamma\ \vdash_{\mathit{meth}} t_0\ mn((t_i\ v_i)_{i:1..p})\{e\}\\ \qquad\hookrightarrow_H t_0\ mn((t_i\ v_i)_{i:1..p})\ \{e_2\}\end{array}}$$

$$\frac{\begin{array}{c}[\textsc{h:local}]\\ \Gamma;\Theta\vdash e_1\hookrightarrow_H e_3::t_1,\Theta_1\\ \vdash t_1<:t\\ \mathit{ann}(t)\notin\{\mathtt{L},\mathtt{R}\}\\ \Gamma{+}\{v::t\};\Theta_1\vdash e_2\hookrightarrow_H e_4::t_2,\Theta_2\\ D=\{v\mid\mathit{ann}(t)=\mathtt{U}\}-\Theta_2\\ e_5=(e_4\lhd D{=}\emptyset\rhd e_4<;\mathit{dispose}(D))\end{array}}{\begin{array}{l}\Gamma;\Theta\vdash(t\ v=e_1\,\mathtt{;}\,e_2)\hookrightarrow_H\\ \qquad(t\ v=e_3\,\mathtt{;}\,e_5)::t_2,\Theta_2\backslash v\end{array}}$$

Fig. 4. Automatic Insertion of `dispose` operation

conditional is expressed as $\xi_1 \lhd b \rhd \xi_2 =_{df} \begin{cases}\xi_1, & \mathit{if}\ b;\\ \xi_2, & \mathit{otherwise}.\end{cases}$ Furthermore, we have:

$$\Theta\backslash v =_{df} \Theta-\{v, v.f^*\} \qquad \Theta\backslash v.f =_{df} \Theta-\{v.f\}$$

For the method declaration rule [h:meth], we add to the disposal set those parameters which are unique but not yet dead using $\{w \mid (w::t)\in\Gamma_1, \mathit{ann}(t)=\mathtt{U}\}-\Theta$. For the local declaration rule [h:local], we add v to the disposal set if it is unique but not yet dead using $\{v \mid \mathit{ann}(t)=\mathtt{U}\}-\Theta_2$. For the [h:if] rule, the uniqueness that are consumed in one branch may have their heap spaces recovered in the other branch. This is captured by $D_i=\Theta_3{-}\Theta_i$, $i=1,2$. Notice that $\mathit{msst}(t_1,t_2)$ returns the minimal supertype of both t_1 and t_2, as follows:

$$\frac{\begin{array}{c}\tau_1<:\tau\quad \tau_2<:\tau\quad \forall\tau_3\cdot(\tau_1,\tau_2<:\tau_3{\Rightarrow}\tau<:\tau_3)\\ A_1{\leq_a}A\quad A_2{\leq_a}A\quad \forall A_3\cdot(A_1,A_2{\leq_a}A_3{\Rightarrow}A{\leq_a}A_3)\end{array}}{\mathit{msst}(\tau_1\mathtt{@}A_1,\tau_2\mathtt{@}A_2)=_{df}\ \tau\mathtt{@}A}$$

Note that $\tau_1<:\tau_2$ denotes the subtype relation for underlying types (without annotations). Alias subtyping rules (shown below) allow unique references to be passed to shared and lent-once locations (in addition to other unique locations), but not vice-versa.

$$A\leq_a A \qquad \mathtt{U}\leq_a\mathtt{L} \qquad \mathtt{U}\leq_a\mathtt{S}$$

In the rest of this paper, we shall present a new static type system for verifying memory heap usage, followed by a set of safety theorems on the type rules.

5 Rules for Memory Checking

We present type judgements for *expressions*, *method declarations*, *class declarations* and *programs* to check for adequacy of memory, using relations of the form:

$$\Gamma;\Delta;\Upsilon \vdash e :: t, \Delta_1, \Upsilon_1 \qquad \Gamma \vdash_{\mathit{meth}} \mathit{meth} \qquad \vdash_{\mathit{class}} \mathit{def} \qquad \vdash P$$

Note that Γ is the type environment as explained earlier; $\Delta(\Delta_1)$ denotes the size constraint, which holds for the size variables associated with Γ (Γ and t) for expression e before (after) its evaluation; t is an annotated type. Also, $\Upsilon(\Upsilon_1)$ is used to denote the available memory space in terms of bag abstraction before (after) the evaluation.

We present a few key syntax-directed type rules in Fig 5, with the rest of the rules in the technical report. Before that, let us describe some notations used by the type rules.

[ASSIGN]

$$\frac{\begin{array}{c}\Gamma;\Delta;\Upsilon \vdash e :: t_1, \Delta_1, \Upsilon_1 \quad \Gamma \vdash w :: t, \phi, Y \\ \vdash t_1 {<:} t, \rho \quad X{=}V(t_1){\cup}V(t) \quad \Delta_2{=}\exists X\cdot(\Delta_1 \circ_Y \rho\phi)\end{array}}{\Gamma;\Delta;\Upsilon \vdash w = e :: \mathtt{void}\langle\rangle\mathtt{@S}, \Delta_2, \Upsilon_1}$$

[DISPOSE]

$$\frac{\Gamma(v) = c\langle n^*\rangle\mathtt{@U} \quad \Upsilon_1 = \Upsilon \uplus \{(c,1)\}}{\Gamma;\Delta;\Upsilon \vdash v.\mathtt{dispose}() :: \mathtt{void}\langle\rangle\mathtt{@S}, \Delta, \Upsilon_1}$$

[NEW]

$$\frac{\begin{array}{c}\mathit{fdList}(c\langle n^*\rangle) = ([(\hat{t}_i\ f_i)]_{i=1}^{p},\ \phi') \\ r^* = \mathit{fresh}() \quad t_i = \mathit{prime}(\Gamma(v_i)) \\ \vdash t_i <: [\mathtt{R} \mapsto \mathtt{S}]\hat{t}_i, \rho_i\ i{\in}1..p \\ \rho = [n^* \mapsto r^*] \cup \bigcup_{i=1}^{p}\rho_i \\ \Delta \vdash \Upsilon \sqsupseteq \{(c,1)\} \quad X = \bigcup_{i=1}^{p} V(\hat{t}_i) \\ \Delta_1 = \Delta{\wedge}(\exists X{\cdot}\rho\phi') \quad \Upsilon_1 = \Upsilon{-}\{(c,1)\}\end{array}}{\Gamma;\Delta;\Upsilon \vdash \mathtt{new}\ c(v_{1..p}) :: c\langle r^*\rangle\mathtt{@U}, \Delta_1, \Upsilon_1}$$

[IF]

$$\frac{\begin{array}{c}\Gamma(v) = \mathtt{bool}\langle b\rangle\mathtt{@S} \\ \Gamma;\Delta \wedge b' = 1;\Upsilon \vdash e_1 :: t_1, \Delta_1, \Upsilon_1 \\ \Gamma;\Delta \wedge b' = 0;\Upsilon \vdash e_2 :: t_2, \Delta_2, \Upsilon_2 \\ (t, \Upsilon_3, \Delta_3) = \ \mathit{unify}(t_1, t_2, \Upsilon_1, \Upsilon_2, \Delta_1, \Delta_2)\end{array}}{\Gamma;\Delta;\Upsilon \vdash \mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2\ :: t, \Delta_3, \Upsilon_3}$$

[OVERRIDE]

$$\frac{\begin{array}{c}\mathit{meth}_k = t\ \mathit{mn}((t_i\ v_i)_{i:1..p})\ \mathtt{where} \\ \phi_{\mathit{pr}_k}; \phi_{\mathit{po}_k}; \epsilon_{km}; \epsilon_{kn}\ \{\cdots\},\ k = 1,2 \\ \phi_{\mathit{pr}_1} {\Rightarrow} \phi_{\mathit{pr}_2} \quad \phi_{\mathit{po}_2} {\Rightarrow} \phi_{\mathit{po}_1} \\ \phi_{\mathit{pr}_1} \vdash \epsilon_{1m} {\sqsupseteq} \epsilon_{2m} \quad \phi_{\mathit{pr}_1} \vdash \epsilon_{2n} {\sqsupseteq} \epsilon_{1n}\end{array}}{\vdash \mathit{OverridesOK}(\mathit{meth}_1, \mathit{meth}_2)}$$

[IMI]

$$\frac{\begin{array}{c}\vdash (A \,\|\, \hat{t}\ \mathit{mn}\boldsymbol{(}(\hat{t}_i\ \hat{v}_i)_{i:1..p}\boldsymbol{)}\ \mathtt{where}\ \phi_{\mathit{pr}}; \phi_{\mathit{po}}; \epsilon_c; \epsilon_r \{e\}) {\in} c\langle n^*\rangle \\ t = \mathit{fresh}(\hat{t}) \quad t_0 = c\langle n^*\rangle\mathtt{@}A \quad \Gamma(v_i) = t_i\ \ i{\in}0..p \quad \vdash t_i <: \hat{t}_i,\ \rho_i\ \ i{\in}1..p \\ \rho_p = \bigcup_{i=1}^{p}\ \rho_i \quad \Delta_1 \vdash \Upsilon {\sqsupseteq} \epsilon_c \quad \rho{=}\mathit{rename}(\hat{t}, t){\cup}\rho_p{\cup}\mathit{prime}(\rho_p) \\ \Delta \mathrel{\approx\!\!>}_{V(\Gamma)} \exists V(\epsilon_c){\cup}V(\epsilon_r){\cdot}\rho\ \phi_{\mathit{pr}} \quad \Delta_1 = \Delta \circ_L \exists Y \cdot \rho(\phi_{\mathit{pr}}{\wedge}\phi_{\mathit{po}}) \\ \Upsilon_1 = \Upsilon{-}\epsilon_c{\uplus}\epsilon_r \quad X = \bigcup_{i=1}^{p} V(\hat{t}_i) \quad Y = X \cup \mathit{prime}(X) \quad L = \bigcup_{i=0}^{p} V(t_i)\end{array}}{\Gamma;\Delta;\Upsilon \vdash v_0.\mathit{mn}(v_{1..p}) :: t, \Delta_1, \Upsilon_1}$$

[METH]

$$\frac{\begin{array}{c}\Gamma_1 = \Gamma \cup \{v_1 :: \hat{t}_1, .., v_p :: \hat{t}_p\} \quad \Delta = \mathit{no}\mathcal{X}(\Gamma_1){\wedge}\phi_{\mathit{pr}}{\wedge}\mathit{inv}(\Gamma_1) \quad \Delta \vdash \epsilon_c {\sqsupseteq} \emptyset \\ \Gamma_1;\Delta;\epsilon_c \vdash e :: t, \Delta_1, \Upsilon_1 \quad \phi_{\mathit{pr}}{\wedge}\Delta_1 \vdash \Upsilon_1 \sqsupseteq \epsilon_r \quad \Delta \vdash \epsilon_r {\sqsupseteq} \emptyset \quad \vdash t <: \hat{t}, \rho \\ (\_, \_, N_i) = \mathbf{V}_{\mathit{field}}(\hat{t}_i),\ i{\in}1..p \quad Y = \bigcup_{i=1}^{p} N_i \quad (\exists\ \mathit{prime}(Y){\cdot}\Delta_1){\Rightarrow}\rho(\phi_{\mathit{po}})\end{array}}{\Gamma \vdash_{\mathit{meth}} \hat{t}\ \mathit{mn}((\hat{t}_i\ v_i)_{i:1..p})\ \mathtt{where}\ \phi_{\mathit{pr}}; \phi_{\mathit{po}}; \epsilon_c; \epsilon_r\ \{e\}}$$

Fig. 5. Some Type Rules for Memory Checking

5.1 Notations

We use function *V* to return size variables of a formula, e.g. $V(x'{=}z{+}1{\wedge}y{=}2){=}\{x', y, z\}$. We extend it to annotated type, type environment, and memory specification, e.g.,

$V(\tau\langle n^*\rangle@A){=}\{n^*\}$, $V(\{(c,4{\times}d{+}8)\}){=}\{d\}$. The function *prime* takes a set of size variables and returns their primed version, e.g. $prime(\{s_1,\ldots,s_n\}){=}\{s'_1,\ldots,s'_n\}$. Note that prime operation is idempotent, namely $(v')'{=}v'$. We extend this to (annotated) type, type environment, and even substitution. For example, $prime(\tau\langle n_1,\ldots,n_k\rangle) = \tau\langle n'_1,\ldots,n'_k\rangle$, and $prime([x{\mapsto}a, y{\mapsto}b]) = [x'{\mapsto}a', y'{\mapsto}b']$. Often, we need to express a no-change condition on a set of size variables. We define a $no\mathcal{X}$ operation as follows which returns a formula for which the original and primed variables are made equal.

$$no\mathcal{X}(\{\}) =_{df} \texttt{true} \qquad no\mathcal{X}(\{x\}{\cup}X) =_{df} (x'{=}x){\wedge}no\mathcal{X}(X)$$

We extend this function to annotated types (and type environments), as follows: $no\mathcal{X}(t) =_{df} no\mathcal{X}(V(t))$. Also, we use $n^* = fresh()$ to generate new size variables n^*. We extend it to annotated type, so that $\hat{t} = fresh(t)$ will return a new type $\hat{t}$ with the same underlying type as t but with fresh size variables instead. Function $rename(t_1, t_2)$ returns an equality substitution, e.g. $rename(\texttt{Int}\langle r\rangle, \texttt{Int}\langle s'\rangle){=}[r{\mapsto}s']$. The operator $\cup$ combines two domain disjoint substitutions into one.

The function *fdList* is used to retrieve a full list of fields for a given class, together with its size relation. The function *inv* is used to retrieve the size invariant that is associated with each type. This function shall also be extended to type environment and list of types. The function $\mathbf{V}_{\mathit{field}}$ classifies size variables from each field into three groups : (i) immutable, (ii) mutable but unique, (iii) otherwise (non-trackable).

To effect a change ϕ to an existing poststate Δ, we provide an operator, $\circ_Y$, with $Y = \{s^*\}$ to denote the set of size variables that is to be updated, as follows:

$$\begin{array}{l} \Delta \circ_Y \phi =_{df} \exists\, r_1 \cdots r_n \cdot \rho_2(\Delta) \wedge \rho_1(\phi) \\ \text{where } Y = \{s_1,\ldots,s_n\}\ ;\ \{r_1,\ldots,r_n\} = fresh()\ ;\ \rho_1 = [s_i \mapsto r_i]_{i=1}^{n}\ ;\ \rho_2 = [s'_i \mapsto r_i]_{i=1}^{n} \end{array}$$

5.2 Assignment

The [ASSIGN] rule captures imperative updates (to object fields and variables) by modifying the current size constraint to a new updated state with changes to the imperative size variables from the LHS. From the rule, note that $\Gamma \vdash w :: t, \phi, Y$ is to identify Y as a set of imperative size variables and also to gather a constraint ϕ for this set. The subtype relation $\vdash t_1 <: t, \rho$ will return a substitution that maps the size variables of supertype to that of the subtype. This mapping ignores all non-trackable size variables that may be globally aliased, but immutable and unique mutable size variables are captured.

5.3 Memory Operations

The heap space is directly changed by the `new` and `dispose` primitives. Their corresponding type rules, [NEW] and [DISPOSE], would ensure that sufficient memory is available for consumption by `new` and will credit back space relinquished by `dispose`. The memory effect is accumulated according to the flow of computation. Consider:

$$\dfrac{\begin{array}{c}\dfrac{\Delta{\vdash}\Upsilon{\sqsupseteq}\{(\texttt{List},1)\} \quad \Delta_1{=}\Delta{\circ_{\{x\}}}x'{=}x{+}1}{\Gamma;\Delta;\Upsilon \vdash \texttt{x = new List(o,x)} :: \texttt{void}\langle\rangle@\texttt{S},\ \Delta_1, \Upsilon{-}\{(\texttt{List},1)\}} \\ \dfrac{\Upsilon_1{=}(\Upsilon{-}\{(\texttt{List},1)\}){\uplus}\{(\texttt{List},1)\}}{\Gamma;\Delta_1;\Upsilon{-}\{(\texttt{List},1)\} \vdash \texttt{y.dispose()} :: \texttt{void}\langle\rangle@\texttt{S},\ \Delta_1, \Upsilon_1}\end{array}}{\Gamma;\Delta;\Upsilon \vdash \texttt{x = new List(o,x); y.dispose()} :: \texttt{void}\langle\rangle@\texttt{S},\ \Delta_1, \Upsilon}$$

The `new` operation consumes a `List` node, while the `dispose` operation releases back a `List` node. The net effect is that available memory Υ is unchanged. However, due to the order of the two operations, we require $\Delta \vdash \Upsilon \sqsupseteq \{(\texttt{List}, 1)\}$ which affects the maximum memory required.

Another rule which has a direct effect on memory is the method invocation rule [IMI]. Sufficient memory must be available for consumption prior to each call (as specified by $\Delta_1 \vdash \Upsilon \sqsupseteq \epsilon_c$), with the net memory release added back in the end (as specified by $\Upsilon_1 = \Upsilon - \epsilon_c \uplus \epsilon_r$). Each method precondition must be met by the pre-state of its caller. This is checked by $\Delta \approx\!\!> _{V(\Gamma)} \exists V(\epsilon_c) \cup V(\epsilon_r) \cdot \rho\ \phi_{pr}$ which uses a relation $\approx\!\!>_X$, defined as:

$$\Delta \approx\!\!>_X \phi \ =_{df} \ (\Delta \Rightarrow \rho\phi), \text{ where } \rho = [s_1 \mapsto s_1', .., s_n \mapsto s_n'] \ \wedge \ V_u(\phi) \cap X = \{s_1, .., s_n\}.$$

Note that V_u returns size variables in unprimed form, e.g. $V_u(x'{=}z{+}1 \wedge y{=}2) = \{x, y, z\}$.

5.4 Conditional

Our type rule for conditional [IF] is able to track both the size-constraints and memory usages in a path-sensitive manner. Path-sensitivity is encoded by adding $b'{=}1$ and $b'{=}0$ to the pre-states of the two branches, respectively. We achieve path-sensitivity for memory usage specification by integrating it with relational size constraints derived. Take note that the *unify* operation merges the post-state constraints and memory usages from the two branches via a disjunction, a formal definition and an example can be found in our report [10]. Path-sensitivity makes our analysis more accurate and is critical for analysing the memory requirement of recursive methods.

5.5 Method Declaration

Each method declaration is checked to see if its definition is consistent with the memory usage specification given in its declaration header by the [METH] rule. The initial memory is ϵ_c. The final available memory of the method body e is Υ_1 which must not be less than the declared net memory release (as specified by $\phi_{pr} \wedge \Delta_1 \vdash \Upsilon_1 \sqsupseteq \epsilon_r$).

Function subtyping for the OO paradigm is used to support method overriding. This is captured by the [OVERRIDE] rule in Fig 5. Each method which overrides another is expected to be *contravariant* on its precondition (and memory consumption) and *covariant* on its postcondition (and memory releases)

6 Soundness of Type System

We have proposed a small-step operational semantics (denoted by $\hookrightarrow$ transitions) instrumented with alias and size notations[10], and have also formalised two safety theorems for our type rules. The first theorem states that each well-typed expression preserves its type under reduction with a runtime environment Π and a store ϖ that are consistent with the compile-time counterparts, Γ (type environment) and Σ (store typing). Also, final size constraint is consistent with the value obtained on termination.

Theorem 1 (Preservation).

(a) (Expression) If $\quad \mathit{\Gamma};\mathit{\Sigma};\mathit{\Delta};\mathit{\Theta};\mathit{\Upsilon} \vdash e :: t, \mathit{\Delta}_1, \mathit{\Theta}_1, \mathit{\Upsilon}_1 \qquad \mathit{\Gamma};\mathit{\Sigma};\mathit{\Delta};\mathit{\Theta};\mathit{\Upsilon} \models \langle \mathit{\Pi}, \varpi, \sigma\rangle$

$$\langle \mathit{\Pi}, \varpi, \sigma\rangle\,[e] \hookrightarrow \langle \mathit{\Pi}_1, \varpi_1, \sigma_1\rangle\,[e_1]$$

then there exist $\mathit{\Sigma}_\alpha \supseteq \mathit{\Sigma}$*,* $\mathit{\Gamma}_\alpha$*,* $\mathit{\Delta}_\alpha$*,* $\mathit{\Theta}_\alpha$*, and* $\mathit{\Upsilon}_\alpha$*, such that*

$$\mathit{\Gamma} - \mathit{diff}(e, e_1) = \mathit{\Gamma}_\alpha - \mathit{diff}(e_1, e) \qquad \mathit{\Gamma}_\alpha;\mathit{\Sigma}_\alpha;\mathit{\Delta}_\alpha;\mathit{\Theta}_\alpha;\mathit{\Upsilon}_\alpha \vdash e_1 :: t, \mathit{\Delta}_1, \mathit{\Theta}_1, \mathit{\Upsilon}_1$$
$$\mathit{\Gamma}_\alpha;\mathit{\Sigma}_\alpha;\mathit{\Delta}_\alpha;\mathit{\Theta}_\alpha;\mathit{\Upsilon}_\alpha \models \langle \mathit{\Pi}_1, \varpi_1, \sigma_1\rangle\ .$$

(b) (Value) If $\quad \mathit{\Gamma};\mathit{\Sigma};\mathit{\Delta};\mathit{\Theta};\mathit{\Upsilon} \vdash (A, \delta) :: t, \mathit{\Delta}_1, \mathit{\Theta}_1; \mathit{\Upsilon}_1 \qquad \mathit{\Gamma};\mathit{\Sigma};\mathit{\Delta};\mathit{\Theta};\mathit{\Upsilon} \models \langle \mathit{\Pi}, \varpi, \sigma\rangle$
then the following hold:

$$\mathit{\Theta} = \mathit{\Theta}_1 \qquad \mathit{\Gamma} + \{x :: t\};\mathit{\Sigma};\mathit{\Delta}_2;\mathit{\Theta}_1;\mathit{\Upsilon}_1 \models \langle \mathit{\Pi} + \{x \mapsto (A, \delta)\}, \varpi, \sigma\rangle$$

where $x = \mathit{fresh}()\ ,\ \mathit{\Delta}_2 = [v \mapsto v']_{v \in V(t)} \mathit{\Delta}_1$.

Proof: By induction over the depth of type derivation for expression e. Details are given in the technical report [10]. □

The second safety theorem on progress captures the fact that well-typed programs cannot go wrong. Specifically, this theorem guarantees that no memory adequacy errors are ever encountered for well-typed MEMJ programs, as follows:

Theorem 2 (Progress). *If* $\mathit{\Gamma};\mathit{\Sigma};\mathit{\Delta};\mathit{\Theta};\mathit{\Upsilon} \vdash e :: t, \mathit{\Delta}_1, \mathit{\Theta}_1, \mathit{\Upsilon}_1$ *and* $\mathit{\Gamma};\mathit{\Sigma};\mathit{\Delta};\mathit{\Theta};\mathit{\Upsilon} \models \langle \mathit{\Pi}, \varpi, \sigma\rangle$*, then either* e *is a value, or* $\langle \mathit{\Pi}, \varpi, \sigma\rangle\,[e] \hookrightarrow$ **Err-Null***, or there exist* $\mathit{\Pi}_1, \varpi_1, \sigma_1, e_1$ *such that* $\quad \langle \mathit{\Pi}, \varpi, \sigma\rangle\,[e] \hookrightarrow \langle \mathit{\Pi}_1, \varpi_1, \sigma_1\rangle\,[e_1]$.

Proof: By induction over the depth of type derivation for expression e. Details are given in the technical report [10]. □

7 Implementation

We have constructed a type checker for MEMJ, and have also built a preprocessor to allow a more expressive language to be accepted. The entire prototype was built using a Haskell compiler[18] where we have added a library (based on [19]) for Presburger arithmetic constraint-solving.

The main objective of our initial experiments is to show that our memory usage specification mechanism is expressive and that such an advanced form of type checking is viable. We converted to MEMJ a set of programs from the Java version of the Olden benchmark suite [7] and another set of smaller programs from the RegJava benchmark[11], before subjecting them to memory adequacy checking. Our initial experimental results are encouraging; however this is a proof-of-concept implementation and there is scope for optimization and more exhaustive experimentation.

Programs	Size (lines)		Checking (in sec.)		Verified
	Source	Ann.	Alias	Memory	Methods
bisort	340	7	0.01	2.56	6/6
em3d	462	19	0.05	1.14	20/20
health	562	22	0.05	6.37	15/15
mst	473	31	0.02	1.26	22/22
power	765	24	0.06	4.28	19/19
treeadd	195	6	0.02	0.32	4/4
tsp	545	10	0.02	3.54	9/9
perimeter	745	12	0.02	21.81	8/8
n-body	1128	31	0.60	1.25	22/22
Voronoi	1000	45	0.03	3.51	39/40
stack	122	12	0.01	0.08	10/10
sieve	88	7	0.01	0.09	6/6
m-sort	183	13	0.01	0.36	12/12
life	164	9	0.02	2.95	7/7
Mandelbrot	194	11	0.01	1.72	10/10
Reynolds3	98	6	0.01	0.18	4/4

Fig. 6. Type Checking Experimental Results

Figure 6 summarises the statistics obtained for each program that we have verified via our type checker. Column 3 illustrates the size and memory annotation overheads which must be made in the header declarations of each class and method. Columns 4 and 5 highlight the CPU times used (in seconds) for alias and memory checking, respectively. Our experiments were done under Redhat Linux 9.0 platform on Pentium 2.4 GHz with 768MB main memory. Except for the perimeter program (which has more conditionals from using a quadtree data structure), all programs take under 10 seconds to verify, despite them being medium-sized programs and the high complexity of Presburger solving. We attribute this to the fact that memory declarations are verified in a summary-based fashion for each method definition. The last column highlights the number of methods that have been successfully verified as using memory spaces that are bounded by symbolic Presburger formulae. All methods' heap usage could be statically bounded, except[2] for a method in Voronoi that has an allocation inside a loop, with a complex termination condition. Apart from the memory checking system described above, we have also conducted some preliminary investigation on memory inference which is described in [17].

8 Related Work

Past research on memory models for object-oriented paradigm have focused largely on efficiency and safety. We are unaware of any prior type-based work on analysing heap memory usage by OO programs for the purpose of checking for memory adequacy. The closest related work on memory adequacy are based on first-order functional paradigm, where data structures are mostly immutable and thus easier to handle.

Hughes and Pareto [15] proposed a type and effect system on space usage estimation for a first-order functional language, extended with region language constructs of Tofte and Talpin's[20]. The use of region model facilitates recovery of heap space. However, as each region is only deleted when all its objects become dead, more memory than necessary may be used, as reported by [4].

Hofmann and Jost [14] proposed a solution to obtain linear bounds on the heap space usage of first-order functional programs. A key feature of their solution is the use

[2] For Olden programs which built tree-like data structure, we make a minor change to take total nodes rather than heights as parameters. This avoids exponential formulae.

of linear typing which allows the space of each last-use data constructor (or record) to be directly recycled by a matching allocation. With this approach, memory recovery can be supported within each function, but *not across functions* in general. Moreover, their model does not track the symbolic sizes of data structures. Nevertheless, one significant advance of their work is an inference mechanism through linear programming (LP) technique. The main advantage of LP technique is that no fix-point analysis is required, but it restricts the memory effects to a linear form without disjunction.

Apart from the above memory analysis work on high level languages, Aspinall and Compagnoni [3] presented a first-order linearly typed assembly language to allow safe reuse of heap space. Their system is a target for the compilation of a functional programming language with a similar type systems (e.g. Hofmann's LFPL) . More recently, Cachera et. al. [6] proposed a constraint-based memory analysis for Java Bytecode-like languages. For a given program their loop-detecting algorithm can detect methods and instructions that execute an unbounded number of times, thus can be used to check whether the memory usage is bounded or not. However, their analysis cannot check whether a given amount of memory is adequate or not, while our system does.

9 Concluding Remarks

We have proposed a memory usage type system for a non-trivial object-oriented core language. We have designed a flexible specification mechanism to allow memory needs of user programs to be declared abstractly, and then verifies if memory adequacy property holds for the given definitions. Our approach requires heap space to be explicitly deallocated, which can be handled automatically. We have also built a prototype type checker to confirm the viability and practicality of our approach. We envision our framework to be useful for embedded system, where memory is considered to be a critical resource. We also envision the synergy of predicable memory bounds with region-based memory management systems. In particular, bounded memory regions can result in better performance. Synergistically, region-based system can provide timely recovery for shared objects that are dead, providing us with tighter memory bounds.

Acknowledgement The authors would like to acknowledge the invaluable help of Florin Craciun with the evaluation of a set of the benchmark programs.

References

1. J. Aldrich, V. Kostadinov, and C. Chambers. Alias Annotation for Program Understanding. In *ACM OOPSLA*, Seattle, Washington, November 2002.
2. B. Alpern, D. Attanasio, J. J. Barton, A. Cocchi, S. F. Hummel, D. Lieber, M. Mergen, T. Ngo, J. Shepherd, and S. Smith. Implementing Jalapeño in Java. In *ACM OOPSLA*, Denver, Colorado, November 1999.
3. D. Aspinall and A. Compagnoni. Heap bounded assembly language. *Journal of Automated Reasoning*, 31:261–302, 2003.
4. E. D. Berger, B. G. Zorn, and K. S. Mckinley. Reconsidering Custom Memory Allocation. In *ACM OOPSLA*, November 2002.
5. J. Boyland, J. Noble, and W. Retert. Capabilities for Sharing: A Generalization of Uniqueness and Read-Only. In *ECOOP*, Budapest, Hungary, June 2001.

6. D. Cachera, T. Jensen, D. Pichardie, and G. Schneider. Certified Memory Usage Analysis. In *13th International Symposium of Formal Methods Europe (FM'05)*, July 2005.
7. M. C. Carlisle and A. Rogers. Software caching and computation migration in Olden. In *4th Principles and Practice of Parallel Programming*, Santa Barbara, California, May 1993.
8. E. C. Chan, J. Boyland, and W. L. Scherlis. Promises: Limited Specifications for Analysis and Manipulation. In *Proceedings of the International Conference on Software Engineering*, pages 167–176, Kyoto, Japan, April 1998.
9. W.N. Chin, S.C. Khoo, S.C. Qin, C. Popeea, and H.H. Nguyen. Verifying Safety Policies with Size Properties and Alias Controls. In *27th International Conference on Software Engineering (ICSE05)*, St. Louis, Missouri, May 2005.
10. W.N. Chin, H.H. Nguyen, S.C. Qin, and M. Rinard. Predictable Memory Usage for Object-Oriented Programs. Technical report, SoC, Natl Univ. of Singapore, November 2004. avail. at http://www.dur.ac.uk/shengchao.qin/papers/memj.ps.gz.
11. M. V. Christiansen and P. Velschow. Region-Based Memory Management in Java. Master's Thesis, Department of Computer Science (DIKU), University of Copenhagen, 1998.
12. M. Fahndrich and R. Leino. Declaring and checking non-null types in an object-oriented language. In *ACM OOPSLA*, Anaheim, CA, October 2003.
13. C. A. R. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.
14. M. Hofmann and S. Jost. Static prediction of heap space usage for first order functional programs. In *ACM POPL*, New Orleans, Louisiana, January 2003.
15. J. Hughes and L. Pareto. Recursion and Dynamic Data-Structures in Bounded Space: Towards Embedded ML Programming. In *Proceedings of the International Conference on Functional Programming (ICFP '99)*, September 1999.
16. L. Lamport. The temporal logic of actions. *ACM Trans. on Programming Languages and Systems*, 16(3):872–923, May 1994.
17. H. H. Nguyen. Memory Usage Inference for Object-Oriented Programs. Technical report, CS Programme, Singapore-MIT Alliance, July 2004. (Term Paper).
18. S Peyton-Jones and et al. Glasgow Haskell Compiler. http://www.haskell.org/ghc.
19. W. Pugh. The Omega Test: A fast practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–114, 1992.
20. M. Tofte and J. Talpin. Region-based memory management. *Information and Computation*, 132(2), 1997.
21. H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *ACM PLDI*. ACM Press, June 1998.

A Alias Checking

We introduce four alias control mechanisms `U` | `S` | `R` | `L` adopted from [5, 8, 1]. These alias mechanisms shall be used to support precise size tracking in the presence of mutable objects, and also for the automatic recovery of dead unique objects. For size-tracking, we introduce `R`-mode fields to allow size-immutable properties to be accurately tracked for all objects. For example, an alternative class declaration for the list data type is given below, where its `next` field is marked as read-only (or immutable). Note that the `val` field remains mutable.

`class RList`$\langle n \rangle$ `where` $n{=}m{+}1$; $n{\geq}0$ { `Object`$\langle\rangle$`@S val;` `RList`$\langle m \rangle$`@R next;` $\cdots$ }

The size property of such an `RList` type can be analysed at compile-time, while allowing its objects to be freely shared. However, this comes at the cost of losing both mutability and uniqueness.

We make use of L-mode parameters, with the *limited unique* (or *lent-once*) property [8], to capture unique references that are temporarily lent out to method calls. They allow the preservation of uniqueness together with precise size-tracking across methods. Consider the following method with two List parameters.

```
void⟨⟩@S join(List⟨m⟩@L x, List⟨n⟩@U y) where n > 0; m′=n+m; ···
  { if isNull(x.next) then x.next = y else join(x.next, y) }
```

The first parameter is annotated as *lent-once* and can thus be tracked for size properties without loss of uniqueness. However, the second parameter is marked *unique* as its reference escapes the method body (into the tail of the List from the first parameter). In other words, the parameter y can have its uniqueness consumed but not x, as reflected in the body of the above method declaration. Given two unique lists, a and b, the call join(a, b) would consume the uniqueness of b but not that of a. Our lent-once policy is more restrictive than normal lending [1] as we require each lent-once parameter to be unaliased within the scope of its method. For example, join(a, a) is allowed by the type rules of [1], but disallowed by our lent-once's policy.

In our alias type system, uniqueness may be transferred from one location (variable, field or parameter) to another location. Consider a type environment {x::Object⟨⟩@U, y::Object⟨⟩@U, z::Object⟨⟩@S} where variables x and y are unique, while z is shared. In the following code, {x = y; z = x}, the uniqueness of y is first transferred to location x, followed by the consumption of uniqueness of x that is lost to the shared variable z. In our type judgement, we track variables/fields that have become dead using:

$$\Gamma ; \Theta \vdash e :: t, \Theta_1$$

Here, each dead-set $\Theta(\Theta_1)$ captures the set of references with consumed uniqueness before(after) the evaluation of expression e. Γ is a type enviroment which maps variables to their annotated types. Other type judgements for methods, classes and programs have the following forms.

$$\Gamma \vdash_{meth} meth \qquad \vdash_{def} def \qquad \vdash_{P} def_{i:1..p}\ meth_{i:1..q}$$

The full set of alias checking rules are given in our technical report [10]).