Memory Usage Inference for Object-Oriented Programs

Huu Hai Nguyen[1], Wei-Ngan Chin[1,2], Shengchao Qin[1,2], and Martin Rinard[1,3]
[1]Singapore-MIT Alliance, [2]Department of Computer Science, National University of Singapore, [3]Computer Science and AI Laboratory, Massachusetts Institute of Technology

***Abstract*— We present a type-based approach to statically derive symbolic closed-form formulae that characterize the bounds of heap memory usages of programs written in object-oriented languages. Given a program with size and alias annotations, our inference system will compute the amount of memory required by the methods to execute successfully as well as the amount of memory released when methods return. The obtained analysis results are useful for networked devices with limited computational resources as well as embedded software.**

***Index Terms*— Type System, Object-Oriented Languages, Memory Management**

I. Introduction

The proliferation of mobile devices with network connections and limited computing resources poses interesting problems. It is desirable for those devices to be able to download and safely execute mobile codes without or with minimal user intervention. To that end, the device must be able to assert that downloaded codes are safe to execute. One aspect of safety is that the codes run within the limited resource supplies of the device, where resources can be memory, CPU cycles, network bandwidth, power, etc. This property is also highly desirable for embedded systems.

There are a number of works that analyze memory usage bounds of programs written in functional languages [13], [11], [12]. In [11], Hofmann and Jost develops a method to automatically derives heap memory bounds for a first-order functional language with explicit heap management. On the other hand, the other works typically require hand annotations from programmers, the systems then verify the correctness of these annotations.

Given the widespread use of object-oriented languages like Java and Java Card [16] in mobile and smart card application development, there is a strong demand to develop analyses that can derive and verify memory usage bounds for programs written in object-oriented languages. For object-oriented languages, we need to deal with different issues than for functional counterparts, namely aliasing, mutable states, and dynamic dispatch. Besides safety guarantees, results of these analyses can offer guidance to optimize resource usages of programs.

In [8], we propose a type system that can capture the bounds of memory usage in object-oriented programs. Memory bounds are described as symbolic formulae in terms of the initial sizes of the methods' parameters.

The key of our proposal is a (size-)polymorphic type system that uses symbolic constraints based on Presburger arithmetic to characterize heap usage of each method in programs written in an object-oriented language. Each method is annotated with symbolic formulae that state, in terms of the initial sizes of the method's input parameters, how much memory is needed to execute the method successfully, and how much memory will be returned to the execution platform when the method finishes. The prototype type checker that we have built demonstrates that our proposal is viable and practical.

In the current paper, in order to further enhance the practicality of the proposed type system, we develop and formalize an inference mechanism to automatically derive the memory effects of methods for a program. We also develop a prototype implementation of the analysis to confirm its viability and practicality.

The structure of the paper is as follows. Section II illustrates the basic ideas via an example. Section III presents the language we are focusing on. Section IV shows the memory notations and operations that are going to be used throughout the paper. Section V presents our inference rules. Section VI shows how we handle recursive methods. Section VII discusses related works and section VIII concludes.

II. Example

To help predict the memory usage of each program, we propose a *sized type system* [6], [7], [14] 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 $\texttt{bool}\langle v\rangle$, the size variable b is either 0 or 1 denoting `false` or `true`, respectively. For user-defined class types, we use $c\langle n_1,\ldots,n_p\rangle$ `where` ϕ ; ϕ_I with size variables $n_1,\ldots,n_p$ to denote size properties that is defined in size relation ϕ, and invariant constraint ϕ_I. As an example, consider

Shengchao Qin is with the Department of Computer Science, School of Computing, National University of Singapore,3 Science Drive 2, Singapore 117543, Republic of Singapore. Email: qinsc@comp.nus.edu.sg. Tel: +65-6874 1298

Wei-Ngan Chin is with the Department of Computer Science, School of Computing, National University of Singapore,3 Science Drive 2, Singapore 117543, Republic of Singapore. Email: chinwn@comp.nus.edu.sg

Martin Rinard is with the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology, 545 Technology Square NE43-620A Cambridge, MA 02139. Email: rinard@lcs.mit.edu

Huu Hai Nguyen is the author for correspondence. He is a student in the Computer Science Programme, Singapore-MIT Alliance. Email: nguyenh2@comp.nus.edu.sg

a stack class that is implemented with a linked list 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;
    :
```

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. 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.

Methods in the target language are annotated with memory bounds as follows:

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

ϕ_{pr} and ϕ_{po} denote the pre-condition and post-condition of the method, expressed in terms of constraints on its size variables. These constraints can provide precise size relations for the parameters and return value of the declared method. The memory effect is captured by ϵ_c and ϵ_r. 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 is 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 | 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

Fig 1 shows the fully annotated class Stack. Methods are annotated with, amongst other things, symbolic memory consumption and memory release. These annotations allow the memory effects of a method to be determined in a straightforward manner once the sizes of the input parameters are known. The notation (A |) prior to each method captures the alias annotation of the current this parameter. We use the primed notation, advocated in [10], [15], 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 post-condition for the pop method, $n'{=}n{-}1$, denotes that the size of the stack is decreased by 1 after the operation while its precondition $n>0$ ensures that pop is not invoked on an empty stack. The memory requirement for the push method, $\{(\texttt{List},1)\}$, captures the fact that one List node will be consumed. For the pop method, $\epsilon_r=\{(\texttt{List},1)\}$ indicates that one List object will be recovered. For the push3pop2 method, heap requirement is $\{(\texttt{List},2)\}$, while the net release is $\{(\texttt{List},1)\}$. This is illustrated by Fig. 2.

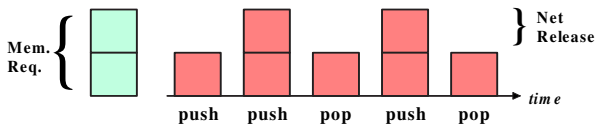

Fig. 2. push3pop2: Heap Consumption and Recovery

Our inference accepts the example above, less the memory annotations, which will be automatically derived.

III. Language and Annotations

We focus on a kernel object-oriented language called MemJ. The syntax of the source language is given in Fig 3. Memory effect annotations shall be inferred automatically. Note that the 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, without altering the external behaviour of method calls.

To support sized typing, our types and methods are augmented with size variables and constraints. For size constraints, we presently restrict to Presburger form, as decidable (and practical) constraint solvers exist, e.g. [17]. For simplicity, we are only interested in tracking size properties of objects. We therefore restrict the relation ϕ in each class declaration of $c_1\langle n_1,\ldots,n_p\rangle$ which extends $c_2\langle n_1,\ldots,n_q\rangle$ to the form $\bigwedge_{i=q+1}^{p} n_i=\alpha_i$ whereby $V(\alpha_i)\cap\{n_1,\ldots,n_p\}=\emptyset$. 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. Size constraints between components, such as those found for balancing the heights of AVL trees are disallowed here, but may be placed in ϕ_I instead.

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 access to a receiver object.

$$
\begin{array}{rl}
P & ::= \mathit{def}^* \ \mathit{meth}^* \\
\mathit{def} & ::= \texttt{class}\ c_1\langle n_{1..p}\rangle\ \texttt{extends}\ c_2\langle n_{1..q}\rangle\ \texttt{where}\ \phi\ ;\ \phi_I\ \{\mathit{fd}^*\ (A\,|\,m)^*\} \\
\mathit{fd} & ::= t\ f \\
m & ::= t\ mn((t\ v)^*)\ \texttt{where}\ \phi_{pr}; \phi_{po}\ \{e\} \\
t & ::= \tau\langle n^*\rangle\texttt{@}A \\
A & ::= \texttt{U} \mid \texttt{L} \mid \texttt{S} \mid \texttt{R} \\
\tau & ::= c \mid \mathit{prim} \\
\mathit{prim} & ::= \texttt{int} \mid \texttt{bool} \mid \texttt{void} \mid \texttt{float} \\
w & ::= v \mid v.f \\
e & ::= (c)\ \texttt{null} \mid k \mid w \mid w = e \mid t\ v = e_1\ ;\ e_2 \mid \texttt{new}\ c(v^*) \\
 & \quad \mid v.mn(v^*) \mid mn(v^*) \mid \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \\
 & \quad \mid v.\texttt{dispose}() \\
\phi & \in \mathbf{F} \qquad \textit{(Presburger Size Constraint)} \\
 & ::= b \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi \mid \exists n \cdot \phi \mid \forall n \cdot \phi \\
b & \in \mathbf{BExp} \qquad \textit{(Boolean Expression)} \\
 & ::= \texttt{true} \mid \texttt{false} \mid \alpha_1 = \alpha_2 \mid \alpha_1 < \alpha_2 \mid \alpha_1 \leq \alpha_2 \\
\alpha & \in \mathbf{AExp} \qquad \textit{(Arithmetic Expression)} \\
 & ::= k^{\texttt{int}} \mid n \mid k^{\texttt{int}} * \alpha \mid \alpha_1 + \alpha_2 \mid -\alpha \\
 & \quad \mid max(\alpha_1,\alpha_2) \mid min(\alpha_1,\alpha_2)
\end{array}
$$

where $k^{\texttt{int}} \in Z$ *is an integer constant*
$n \in SV$ *is a size variable*
$f \in Fd$ *is a field name*
$v \in Var$ *is an object variable*

Fig. 3. Syntax of the language

One important characteristic of MEMJ is that memory recovery is done explicitly. In particular, dead objects may be reclaimed via a $v.\texttt{dispose}()$ primitive. We provide an analysis to insert these calls safely and automatically based on uniqueness information.

A. Alias Checking

We introduce four alias control mechanisms U | S | R | L adopted from [3], [4], [1]. These alias mechanisms shall be used to support precise size tracking in the presence of mutable objects, and also for the explicit recovery of memory space when unique objects become dead. 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⟨n⟩ where n=m+1 ; n≥0 {
  Object⟨⟩@S val;
  RList⟨m⟩@R next;
    :
```

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 mutability and the lost of uniqueness.

We make use of L-mode parameters, with the *limited unique* (or *lent-once*) property [4], 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 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 the policy of 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 (by either assignment or parameter-passing) 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. 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.

$$A \leq_a A \quad \texttt{U} \leq_a \texttt{L} \quad \texttt{U} \leq_a \texttt{S}$$

A key difference of our alias checking rules, when compared to [1], is that we do not require an external "last-use" analysis for variables . Neither do we need to change the underlying semantics of programs to nullify each location whose uniqueness is lost. We achieve this with a special set of references whose uniqueness have been consumed, called *dead-set* of the form $\{w^*\}$ where $w = v \mid v.f$. This dead-set is tracked flow-sensitively in our system.

IV. MEMORY 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. For instance, $\Upsilon_1 = \{(c_1, 2), (c_2, 4), (c_3, 3)\}$ denotes a bag with c_1 appearing twice, c_2 appearing four times and c_3 appearing thrice. We provide the following two basic operations for bag abstraction to capture both the domain and the count of its element, as follows:

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

We define several operations (union, difference, exclusion) over bags. These operations combine two bags to produce a new bag:

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

We also define operations that compare two bags. These operations are used during inference to generate memory adequacy constraints:

$$\begin{array}{lll}\Upsilon_1 \sqsupseteq \Upsilon_2 & =_{df} & \bigwedge_{c\in Z} \Upsilon_1(c) \geq \Upsilon_2(c)\\ & & \textit{where } Z = \mathit{dom}(\Upsilon_1) \cup \mathit{dom}(\Upsilon_2)\\ \Upsilon_1 = \Upsilon_2 & =_{df} & \bigwedge_{c\in Z} \Upsilon_1(c) = \Upsilon_2(c)\\ & & \textit{where } Z = \mathit{dom}(\Upsilon_1) \cap \mathit{dom}(\Upsilon_2)\end{array}$$

V. Memory Inference

The inference phase analyzes the methods in the program, propagating size and memory usage information, and collecting memory adequacy constraints. These constraints are then solved, with the help of fixpoint computation when recursion is present, and memory requirement and memory release are derived from solution to the constraints.

The inference works in a bottom-up order of the call graph, processing one strongly connected component at a time. The processing of each strongly connected component consists of the following steps:

1) Calculating symbolic program state, building constraint abstractions, and collecting memory adequacy constraints
2) Solving constraint abstractions using fixpoint
3) Deriving memory requirement and memory release based on the computed invariant and collected memory adequacy constraints

The type inference rule for expressions has the following form:

$$\Gamma; \Delta; \Upsilon \vdash e :: t, \Delta', \Upsilon', \Phi$$

where type environment Γ maps variables to their annotated types, $\Upsilon(\Upsilon')$ are memory available before and after evaluation of e, respectively, $\Delta(\Delta')$ are the size constraints before and after evaluation of expression e, respectively. Φ is a set of (Δ, φ) pairs where φ is the constraint that enforces memory adequacy and Δ is the size context at the program point where φ is generated. We need to carry the size context along so that we can generate memory requirement and to suitably combine them.

The inference rule for methods has the following form:

$$\Gamma \vdash_{\mathit{meth}} \mathit{meth} \hookrightarrow \mathit{meth'} \mid \mathcal{Q}$$

where Γ is empty for static methods and contains one single entry for `this` for instance methods. The method *meth'* is the transformed version of *meth* where memory effects annotations are added. The constraint abstraction $\mathcal{Q}$ captures the relations between the sizes of the method's parameters and the memory effects of the method.

Fig. 4 presents our rules for memory effects inference. In the following subsections, we will take a closer look at the more important inference rules.

A. Notations

This subsection defines the notations that we use when formulating the inference. We use function V to return size variables in a size formula, e.g.,$V(x'{=}z{+}1 \wedge y{=}2) = \{x', y, z\}$ We extend it to annotated type, type environment, and memory specification.

The function *prime* takes a set of size variables and returns their primed version, e.g. $\mathit{prime}(\{s_1, \ldots, s_n\}) = \{s_1', \ldots, s_n'\}$. This operation is idempotent, namely $v'' = v'$. We extend it to annotated type, type environment, memory, and substitution.

Often, we need to express a no-change condition on a set of size variables. We define a $n\alpha\mathcal{X}$ operation as follows which returns a formula for which the original and primed variables are made equal.

$$n\alpha\mathcal{X}(\{\}) =_{df} \texttt{true} \qquad n\alpha\mathcal{X}(\{x\} \cup X) =_{df} (x' = x) \wedge n\alpha\mathcal{X}(X)$$

We use $n^* = \mathit{fresh}()$ to generate new size variables n^*. We extend it to annotated type, so that $\hat{t} = \mathit{fresh}(t)$ will return a new type $\hat{t}$ with the same underlying type as t but with fresh size variables instead. The function $\mathit{equate}(t_1, t_2)$ generates equality constraints for the corresponding size variables of its two arguments, usually when both arguments share the same underlying type. For example, we have $\mathit{equate}(\texttt{Int}\langle r\rangle\ , \texttt{Int}\langle s'\rangle) = (r = s')$. Conditional is expressed as $\xi_1 \lhd b \rhd \xi_2 =_{df} \begin{cases}\xi_1 & \textit{if } b\\ \xi_2 & \textit{otherwise.}\end{cases}$ The function $\mathit{rename}(t_1, t_2)$ returns an equality substitution, e.g. $\mathit{rename}(\texttt{Int}\langle r\rangle, \texttt{Int}\langle s'\rangle){=}[r \mapsto s']$. 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.

Imperative size changes occur in the presence of assignment and are captured by the sequential operator, which is defined as:

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

For example, if we have $\texttt{x} = 2; \texttt{x} = \texttt{x} + 1$, then size changes can be captured as:

$$\begin{array}{rcl}\Delta & \equiv & x' = 2 \circ_{\{x\}} x' = x + 1\\ & \equiv & \exists r \cdot r = 2 \wedge x' = r + 1\\ & \equiv & x' = 3\end{array}$$

The result correctly reflects the state of variable `x` after the code sequence.

B. Memory Operations

Heap space is directly changed by the `new` and `dispose` primitives. [NEW] generates constraints to ensure that sufficient memory is available for consumption by `new`. [DISPOSE] credits back space relinquished by `dispose`, thus no constraint is generated. The memory effect is accumulated according to the flow of computation. The following example illustrates these rules. Suppose $\Gamma = \{\texttt{x} :: \texttt{List}\langle x\rangle\texttt{@U}, \texttt{y} :: \texttt{List}\langle y\rangle\texttt{@U}\}$, $\Upsilon = \{(\texttt{List}, c)\}$.

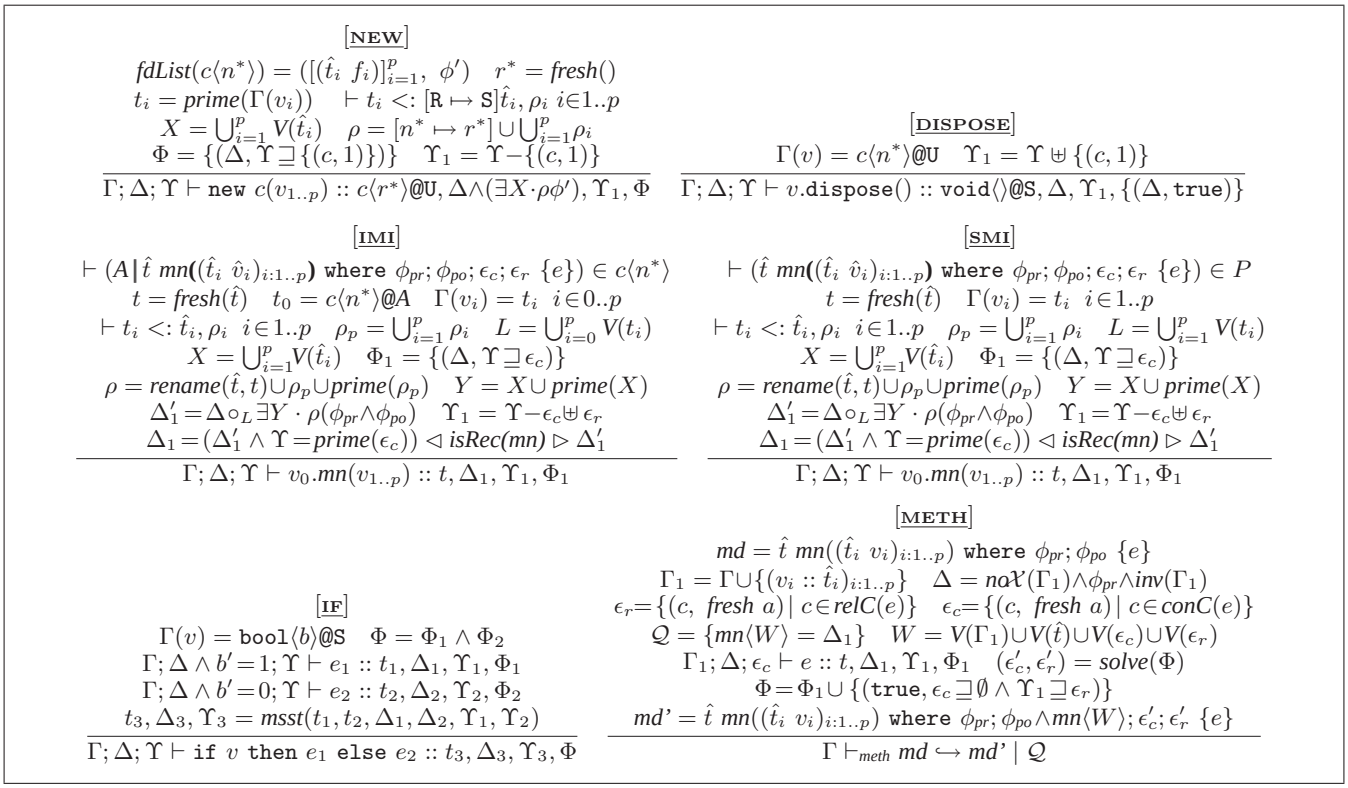

Fig. 4. Memory Inference Rules

$$\frac{\frac{\frac{\Phi_1 = \{(\Delta, \Upsilon \sqsupseteq \{(\mathtt{List}, 1)\})\} = \{(\Delta, c \geq 1)\}}{\Delta_1 = \Delta \circ_{\{x\}} x' = x+1 \quad \Upsilon_1 = \Upsilon - \{(\mathtt{List}, 1)\}}}{\Gamma; \Delta; \Upsilon \vdash \mathtt{x} = \mathtt{new}\ \mathtt{List(o,x)} :: \mathtt{void}\langle\rangle@\mathtt{S}, \Delta_1, \Upsilon_1, \Phi_1}}{\Upsilon_2 = \Upsilon_1 \uplus \{(\mathtt{List}, 1)\}}}{\Gamma; \Delta_1; \Upsilon_1 \vdash \mathtt{y.dispose}() :: \mathtt{void}\langle\rangle@\mathtt{S}, \Delta_1, \Upsilon_2, \Phi_1}}{\Gamma; \Delta; \Upsilon \vdash \mathtt{x{=}new\ List(o,x);y.dispose}() :: \mathtt{void}\langle\rangle@\mathtt{S}, \Delta_1, \Upsilon_2, \Phi_1}$$

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, i.e. $\Upsilon_2 = \Upsilon$. However, due to the order of the two operations, we require that there must be at least one `List` node initially. This is captured by the following memory adequacy constraint:

$$\varphi = \Upsilon \sqsupseteq \{(\mathtt{List}, 1)\} = c \geq 1$$

C. Method Invocation

For method invocation, parameter passing is modelled by substitution. This process replaces size variables of formal parameters by that of actual ones. Those size variables that cannot be tracked due to global sharing are removed by existential quantifier. To ensure memory adequacy, the rule generates constraints that force available memory to be no less than what is needed for the callee.

D. Methods

The inference rule for method ([METH]) generates a fresh memory variable for each class of which objects may be allocated or released by the method. These variables are placeholders for the amount of memory available at the start of the method. Our inference then builds a constraint abstraction that captures the relation between these memory variables and the sizes of the method input parameters.

Some auxiliary functions are used in this rule. *conC(e)* returns the sets of object classes that may be consumed by expression e; *relC(e)* returns the sets of object classes that may be released by expression e. These sets can be computed separately by a simple bottom-up effect analysis.

To illustrate how the rule works, we consider the `push` method of the `Stack` class. This method allocates new `List` node, so the [METH] generates its initial memory as $\epsilon_c = \{(\mathtt{List}, cl)\}$. While checking the method body, the `new` operator generates the constraint $\epsilon_c \sqsupseteq \{(\mathtt{List}, 1)\} \equiv cl \geq 1$. Hence we can derive the memory requirement for the `push` method as $\{(\mathtt{List}, 1)\}$.

E. Generating Memory Effects

Generated preconditions can be combined to reduce the size of the memory effects. This process strengthens the preconditions, and makes them more compact. This is important for scalability. The idea is that if the conjunction of the precondition of one branch and the size context of another branch implies the precondition of the latter, then the latter's precondition is redundant. In this case the precondition of the former suffices for both. More formally, this is described as follows:

$$\begin{array}{l} reduce((\Delta_1, pre_1), (\Delta_2, pre_2)) =_{df} \\ \quad \textit{if } \Delta_1 \wedge pre_2 \implies pre_1 \textit{ then} \\ \quad\quad \{(\Delta_1 \vee \Delta_2, pre_2)\} \\ \quad \textit{else if } \Delta_2 \wedge pre_1 \implies pre_2 \textit{ then} \\ \quad\quad \{(\Delta_1 \vee \Delta_2, pre_1)\} \\ \quad \textit{else} \\ \quad\quad \{(\Delta_1, pre_1), (\Delta_2, pre_2)\} \end{array}$$

VI. Memory Analysis for Recursive Methods

A. Deriving Memory Invariant

For each recursive method, we construct a constraint abstraction that relates the sizes of the input parameters, the amount of memory available at the beginning of the method to the sizes of the parameters of passed to the recursive call and the amount of memory available just prior to the recursive call. This one-step relation is subjected to a fixpoint procedure to compute the multi-step relation. Let $I\langle n^*, m^*, \hat{n}^*, \hat{m}^*\rangle$ be the one-step relation, the fixpoint computation is formulated as follows [20]. Note that n^* and m^* are the sizes of the input parameters and memory available at the beginning, respectively, whereas $\hat{n}^*$ and $\hat{m}^*$ are that of the recursive call.

$$I_1\langle n^*, m^*, \hat{n}^*, \hat{m}^*\rangle = I\langle n^*, m^*, \hat{n}^*, \hat{m}^*\rangle$$

$$I_{i+1}\langle n^*, m^*, \hat{n}^*, \hat{m}^*\rangle = I_i\langle n^*, m^*, \hat{n}^*, \hat{m}^*\rangle \vee \exists n_0^*, m_0^* \cdot I_i\langle n_0^*, m_0^*, \hat{n}^*, \hat{m}^*\rangle \wedge I\langle n^*, m^*, n_0^*, m_0^*\rangle$$

For the computation to converge, we may need to apply approximation techniques such as hulling and widening [9], [20].

B. Deriving Memory Requirement

The technique we use to derive memory requirement is similar to that used to derive precondition for array bound checks [20]. A memory adequacy constraint φ is turned into a memory requirement by the following steps. First we compute *pre* $\equiv \Delta \gg\!\!\!\approx \varphi$. Δ is the size state at the program point where memory constraint φ is generated. The operator $\gg\!\!\!\approx$ is defined as:

$$\Delta \gg\!\!\!\approx \varphi = (\Delta \implies \rho\ \varphi) \quad \texttt{where} \quad \rho = s_1 \mapsto s_1', \ldots, s_n \mapsto s_n' \quad s_1, \ldots, s_n = V(\varphi)$$

Then we project *pre* into the sizes of the method's parameters by using the $\downarrow_V$ operator, which is defined as follows:

$$\phi \downarrow_V = \forall W \cdot \phi \ \texttt{where}\ W = V(\phi) - V$$

What the $\downarrow_V$ operator does is it eliminates all free size variables in a formulae ϕ by means of universal quantifier, except for those specified in V.

C. Deriving Memory Release

To derive memory release, we build a constraint abstraction that relates the amount of memory available at the end of the method with the amount of memory available at the beginning and the sizes of the method parameters. We then solve this constraint abstraction by fixpoint method to get a safe approximation of the relation. This solved form will allow us to determine the amount of memory released when the method returns.

In general, given a constraint abstraction for a recursive method

$$q\langle n^*, o\rangle = \phi_0 \vee \phi_1[q\langle n_1^*, o_1\rangle, q\langle n_2^*, o_2\rangle]$$

where ϕ_0 represents the base case, and $\phi_1[q\langle n_1^*, o_1\rangle, q\langle n_2^*, o_2\rangle]$ represents the recursive case, postcondition can be derived as follows [9]:

$$q_0\langle n^*, o\rangle = \texttt{false}$$
$$q_{i+1}\langle n^*, o\rangle = \phi_0 \vee \phi_1[q_i\langle n_1^*, o_1\rangle, q_i\langle n_2^*, o_2\rangle]$$

VII. Related Work

Past research on improving memory models for object-oriented paradigm have focused largely on efficiency, minimization and safety. We are unaware of any prior work on analysing 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. We shall review two recent works in this area before discussing other complimentary works.

Hughes and Pareto [13] proposed a type and effect system on space usage estimation for a first-order functional language, extended with region language constructs of Tofte's and Talpin[19]. Their sized-region type system maintains upper bounds on the height of the run-time stack, and on the memory used by regions. 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 [2]. Stack usage has been modelled in Hughes and Pareto's work, but tail-call optimization is not supported.

More recently, Hofmann and Jost [11] proposed a solution to obtain linear bounds on the heap space usage by first-order functional programs. A key feature of their solution is the use of linear typing which allows the space of each last-use data constructor (or record) to be directly recycled by a matching allocation. As a consequence, memory recovery can be supported within each function, but not across functions unless memory tokens are explicitly passed. Also, stack usage is not covered by their model. To infer memory bounds of functions, they derive a linear program based on the function's typing derivations and the fact that all type annotations in their system are non-negative. The solution of the generated linear program directly yields suitable memory bounds required.

This derivation technique can be put under the more general framework by Rugina and Rinard [18]. They transform the bounds of accessed memory regions, including pointers and array accesses, into linear programs. The derived results can be used in a number of analyses, including parallelization, data race detection, array bound check elimination.

Another recent work on resource verification is [5]. They use a combination of both static checks to be verified by compilers and dynamic checks to be verified by runtime systems. Dynamic checks are associated with operations that reserve resources from execution environments whereas static checks are associated with operations that consume the reserved resources. The reserve operations may fail at runtime due to resource insufficiency. The static checks ensure that, once the reserve operations succeed, the consumption operations never fail due to insufficiency of resources.

VIII. Conclusion

We have proposed a type-based analysis to automatically derive the memory effects of methods in object-oriented programs. The analysis can derive both memory consumption and release. Analysis of recursive methods is carried out with the help of fixpoint analysis of derived constraint abstractions. The

memory effects of methods are propagated interprocedurally. Moreover, the ideas presented here can be applied to analyze other kind of computational resource usages as well.

Although constraints used in the analysis are limited by the linear nature of Presburger arithmetic, memory effects are not necessarily one *single* linear combination of the sizes of method's parameters. Instead we can have multiple linear formulae, each representing the memory effects of one or more execution paths through the method's body, thereby allowing more expressiveness and better accuracy.

REFERENCES

[1] J. Aldrich, V. Kostadinov, and C. Chambers. Alias Annotation for Program Understanding. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, Seattle, Washington, November 2002.

[2] E. D. Berger, B. G. Zorn, and K. S. McKinley. Reconsidering Custom Memory Allocation. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, November 2002.

[3] J. Boyland, J. Noble, and W. Retert. Capabilities for Sharing: A Generalization of Uniqueness and Read-Only. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Budapest, Hungary, June 2001.

[4] E. C. Chan, J. Boyland, and W. L. Scherlis. Promises: Limited Specifications for Analysis and Manipulation. In *Proceedings of the IEEE International Conference on Software Engineering (ICSE)*, pages 167–176, Kyoto, Japan, April 1998.

[5] Ajay Chander, David Espinosa, Peter Lee, and George C. Necula. Optimizing resource bounds enforcement via static verification of dynamic check placement. In *Submitted to the 11th ACM Conference on Computer and Communications Security*, October 2004.

[6] W.N. Chin and S.C. Khoo. Calculating sized types. In *Proceedings of the ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, pages 62–72, Boston, Massachusetts, United States, January 2000.

[7] 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 *Proceedings of the 27th International Conference on Software Engineering*, May 2005. To appear.

[8] 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, March 2004. avail. at http://www.comp.nus.edu.sg/∼qinsc/papers/memj.ps.gz.

[9] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the ACM Symposium on the Principles of Programming Languages (POPL)*, pages 84–96. ACM Press, 1978.

[10] C. A. R. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.

[11] M. Hofmann and S. Jost. Static prediction of heap space usage for first order functional programs. In *Proceedings of the ACM Symposium on the Principles of Programming Languages (POPL)*, New Orleans, Louisiana, January 2003.

[12] Martin Hofmann. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, 7, December 2000.

[13] J. Hughes and L. Pareto. Recursion and Dynamic Data-Structures in Bounded Space: Towards Embedded ML Programming. In *Proceedings of the ACM Conference on Functional Programming (ICFP)*, September 1999.

[14] J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *Proceedings of the ACM Symposium on the Principles of Programming Languages (POPL)*, pages 410–423. ACM Press, January 1996.

[15] L. Lamport. The temporal logic of actions. *ACM Trans. on Programming Languages and Systems*, 16(3):872–923, May 1994.

[16] Sun Microsystems. Java Card platform specification. http://java.sun.com/products/javacard/.

[17] W. Pugh. The Omega Test: A fast practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–114, 1992.

[18] R. Rugina and M. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 182–195. ACM Press, June 2000.

[19] M. Tofte and J. Talpin. Region-based memory management. *Information and Computation*, 132(2), 1997.

[20] D.N. Xu, C. Popeea, S.C. Khoo, and W.N. Chin. Modular inference for array checks optimization. Technical report, SoC, Natl Univ. of Singapore, November 2003. avail. at http://www.comp.nus.edu.sg/∼xun/research/array_tech.ps.