2008 15th Asia-Pacific Software Engineering Conference

# A Heap Model for Java Bytecode to Support Separation Logic *

Chenguang Luo Guanhua He Shengchao Qin
Department of Computer Science, Durham University
{chenguang.luo, guanhua.he, shengchao.qin}@durham.ac.uk

## Abstract

*Memory usage analysis is an important problem for resource-constrained mobile devices, especially under mission- or safety-critical circumstances. Program codes running on or being downloaded into such devices are often available in low-level bytecode forms. We propose in this paper a formal heap model for Java bytecode language, on top of which we can then provide separation logic support for further memory usage verification. Our low-level heap model for Java bytecode would allow us to reason about the size and alignment properties of primitive values stored in the heap. To support type-related reasoning such as guaranteeing type and alignment safety, this model is also lifted with both base types and user-defined classes. Based on such model, we have also defined a separation logic proof system whose assertions are interpreted using the lifted heap with types. We envision, with further extension, the system would provide good support for memory usage analysis and verification for mobile devices.*

## 1 Introduction

At present Java bytecode is widely applied in diverse areas. Its key features, including platform independence and relative downloading security, agree with the need of the market and lead to its success.

Currently tens of kinds of micro-devices are running Java bytecode. Though with different shapes and functions, these micro-devices share some similarities. Generally, such a device has a processor and some memory to run a Java virtual machine for downloaded bytecode to execute. The resource of these micro-devices for bytecode programs, such as the memory space, is usually quite limited. A case in point is that many brands of mobile phones provide a memory not more than one megabyte, which is several hundredth to one thousandth of that of modern personal computers. Because of this resource restriction, it is preferable to ensure that a piece of downloaded bytecode will not require more memory resource than the micro-device can provide before we actually load the code and execute it, so as to avoid a crash due to an `OutOfMemoryError`. However, the static analysis and verification of a bytecode's memory usage is a challenging topic, not only because it is difficult to infer accurately by a termination-guaranteed approximation, but also because the verifier does not have access to the more structured and human-readable Java source codes when working on the assembly-level bytecode.

Despite these difficulties, there are already a few approaches [2, 6, 8][1] for bytecode memory resource analysis. Our aim in this paper is also to provide some theoretical support for a proposed framework for Java bytecode memory resource analysis, which consists of a semantic model, a separation logic built on it, and some verification and analysis mechanisms on top of the logic. Until now some progress has been achieved on the latter two aspects [4, 11, 3]. In those works, the traditional separation logic's semantic model and syntax were adapted in constructing a verification framework for heap-related properties. For more accuracy, we would like to reconstruct that framework upon a better mimicry of the heap for Java bytecode, and hence we set up this heap model. It simulates the data in the memory cells (where a piece of data can take up several memory cells and an object may contain many pieces of data as its fields) and records the data's types after a lifting. We anticipate that more precise result will be gained with the extra expressiveness of the separation logic founded on this model.

The main features of our proposed heap model are summarized below:

- It introduces a low-level model of the memory into separation logic to increase its expressiveness and to support reasoning about both low-level and high-level properties (such as values represented with bytes, their types and their nesting objects). Compared with this model, the one to support the traditional semantics of separation logic does not care for most of these prop-

*This work is supported in part by the EPSRC project EP/E021948/1.

[1] More discussions on these works will be given in Section 5.

1530-1362/08 $25.00 © 2008 IEEE
DOI 10.1109/APSEC.2008.72

erties like bytes in memory and variables' types.

- It provides sufficient flexibility to cater for diverse Java virtual machine implementations. To illustrate, the length of a reference, or different alignments, can be chosen for a specified virtual machine implementation.
- The model is loosely coupled with the axiomatic system, in the sense that they are explicitly distinguished in different levels in the whole system. Thus it is relatively easy to extend the separation logic (in high-level) to facilitate reasoning about various heap properties within the range of expressiveness of the model (in low-level).

The remainder of this paper is organized as follows. Section 2 illustrates the low-level crude heap model only with the bytes information. Then in Section 3 it will be lifted with types added to each allocated heap cell and also to the non-dangling references. Section 4 focuses on the separation logic system built on this model, which deals with assertions and Hoare triples abstracted from the heap states, and illustrates the application of the proof rules by an example. The last two sections discuss related work and conclude this paper.

## 2 Heap with Bytes

In this section, we propose a memory model for Java bytecode. Such model will concentrate on the low-level memory representation of variables and objects in Java. It also acts as a foundation of our separation logic implementation. In what follows, we will define the model itself and relative functions for the model to work.

### 2.1 Heap Definition

At a lower level, our heap model is a simulation of the logical memory, which is a repository numbered with addresses to store primitive values in the form of bytes. Such primitive values can be of standard base types in bytecode, which corresponds to the *BaseType* $= \{\mathsf{B}, \mathsf{C}, \mathsf{D}, \mathsf{F}, \mathsf{I}, \mathsf{J}, \mathsf{S}, \mathsf{Z}, \mathsf{L}, [\}$ (which types represent `byte`, `char`, `double`, `float`, `int`, `long`, `short`, `boolean`, object reference and array reference, respectively) in the Java virtual machine specification.

Based on the description above, our heap model is defined to be a mapping from memory addresses to primitive values in bytes. Hence the memory addresses and byte values should be formalized in advance:

$$\begin{aligned} \mathsf{max\text{-}addr} &= 2^{32} - 1 \\ \mathsf{max\text{-}byte} &= 2^8 - 1 \\ \mathit{Address} &= \{r \mid r \in \mathbb{N} \wedge r \leq \mathsf{max\text{-}addr}\} \\ \mathit{Byte} &= \{b \mid b \in \mathbb{N} \wedge b \leq \mathsf{max\text{-}byte}\} \end{aligned}$$

which represents 32-bits memory address and 8-bits byte value, respectively. In the implementation of this model, these parameters should be changed to cater for the real Java virtual machine implementation, such as 32-bits or 64-bits memory address.

Along with such foundations comes the definition for our heap model:

$$\mathit{heap} :: \mathit{Address} \rightharpoonup \mathit{Byte}$$

Note that we use $\rightharpoonup$ for partial mappings, compared with the $\rightarrow$ for total ones. For this model, given an aforesaid base type, two mappings are utilized to convert any value in that type to its byte representation, and vice versa:

$$\begin{aligned} \mathsf{value\text{-}byte} &:: \mathit{BaseType} \rightarrow \mathit{Value} \rightarrow [\mathit{Byte}] \\ \mathsf{byte\text{-}value} &:: \mathit{BaseType} \rightarrow [\mathit{Byte}] \rightharpoonup \mathit{Value} \end{aligned}$$

where *BaseType* is the set of all base types in bytecode, *Value* is the set of all primitive values, and $[T]$ is the set of all lists over values in type $T$. Note that the second mapping is partial since not every byte list has a corresponding value in any base type. To ensure the correct conversion, the lifting of the heap will also carry the type information, which is introduced in the next section.

Besides the relationship between primitive values and their byte forms, the size consumption of an element of base types, and the alignment of Java virtual machine, are defined as follows:

$$\begin{aligned} \mathsf{size\text{-}of} &:: \mathit{BaseType} \rightarrow \mathbb{N} \\ \mathsf{alignment} &\in \mathbb{N} \end{aligned}$$

where the result is measured in bytes. For these two concepts the following axioms hold that

$$\begin{aligned} 0 < \mathsf{size\text{-}of}\ t &\leq \mathsf{max\text{-}addr} \\ \mathsf{alignment} \mid (\mathsf{max\text{-}addr}+1) & \end{aligned}$$

where $\mid$ is the divisibility judgment operator. These axioms are directly derived from the Java virtual machine specification, and are subject to certain implementations. As an example, for Sun's 32-bits implementation of Java virtual machine, we have $\mathsf{size\text{-}of}\ \mathsf{int} = 4$, and $\mathsf{alignment} = 8$.

The reference type (including $\mathsf{L}$ and $[$) is the way for Java to deal with heap memory. In our model, a reference type has an *Address* typed value. For the operations over the reference type, since Java does not allow direct address arithmetic (thus a developer is not able, and also does not need, to add an arbitrary address value to a reference), a means to get an offset from a reference and another base type is provided.

$$\begin{aligned} &\mathsf{ref\text{-}plus} :: \mathit{Address} \rightarrow \mathit{BaseType} \rightharpoonup \mathit{Address} \\ &\mathsf{ref\text{-}plus}\ r\ t = r + \mathsf{size\text{-}of}\ t \end{aligned}$$

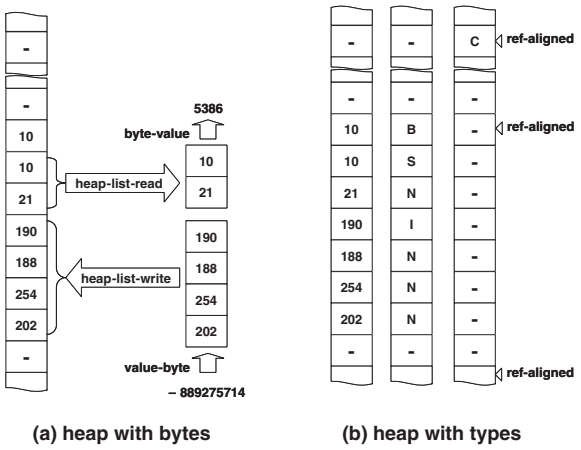


**Figure 1. An example of our model**

which is lifted to function to deal with a list of types:

$$\begin{array}{l}\textsf{ref-plus-list} :: \mathit{Address} \to [\mathit{BaseType}] \rightharpoonup \mathit{Address}\\ \textsf{ref-plus-list}\ r\ [\,] = r\\ \textsf{ref-plus-list}\ r\ (t : ts) = \textsf{ref-plus-list}\ (\textsf{ref-plus}\ r\ t)\ ts\end{array}$$

This function will be useful in the field access in the following sections.

### 2.2 Heap Access

Before we introduce the heap access mechanism, we need a function to construct a list from a series of specified addresses:

$$\begin{array}{l}\textsf{heap-list-read} :: \mathit{Heap} \to \mathit{Address} \to \mathbb{N} \to [\mathit{Byte}]\\ \textsf{heap-list-read}\ h\ r\ 0 = [\,]\\ \textsf{heap-list-read}\ h\ r\ n = \textsf{if}\ r{+}n > 1{+}\textsf{max-addr}\ \textsf{then}\ [\,]\\ \qquad \textsf{else}\ \textsf{heap-list-read}\ h\ r\ (n{-}1) +\!\!+ [h\ (r{+}n{-}1)]\\ \textsf{heap-list-write} :: \mathit{Heap} \to [\mathit{Byte}] \to \mathit{Address} \to \mathit{Heap}\\ \textsf{heap-list-write}\ h\ [\,]\ r = h\\ \textsf{heap-list-write}\ h\ (v : vs)\ r = \textsf{if}\ r{+}\mathit{length}(v : vs) > 1{+}\\ \quad \textsf{max-addr}\ \textsf{then}\ h\ \textsf{else}\ \textsf{heap-list-write}\ (h(r \mapsto v))\ vs\ r{+}1\end{array}$$

where *Heap* is the set of all possible heaps, $+\!\!+$ means list concatenation, and $h(r \mapsto v)$ stands for the same heap as $h$ except that $r$ is mapped to $v$. Then the operations to read/write data from/to any heap are defined as:

$$\begin{array}{l}\textsf{heap-read} :: \mathit{Heap} \to \mathit{Address} \to \mathit{BaseType} \rightharpoonup \mathit{Value}\\ \textsf{heap-read}\ h\ r\ t = \textsf{byte-value}\ t\\ \qquad\qquad (\textsf{heap-list-read}\ h\ r\ (\textsf{size-of}\ t))\\ \textsf{heap-write} :: \mathit{Heap} \to \mathit{Value} \to \mathit{Address} \to \mathit{BaseType}\\ \qquad \to \mathit{Heap}\\ \textsf{heap-write}\ h\ v\ r\ t = \textsf{heap-list-write}\ h\ (\textsf{value-byte}\ t\ v)\ r\end{array}$$

The composition and application of the functions are illustrated in Figure 1 (a).

Note that in our low-level representation of the heap model, the access to the heap needs explicit specification of data types, since the heap itself contains no such information. This is an ideal simulation of the memory storage during bytecode execution and useful for inferring its features such as size of data structures; however, as a strongly typed language, Java calls for a lifting from this crude heap to a typed one, which is introduced in the next section.

## 3 Heap with Types

Unlike C or C++, Java virtual machine has a special requirement for its heap space. That is, no primitive value (such as an integer or a character) can appear solely in the heap; it must be encapsulated as a field in an object. To illustrate, a Java programmer cannot apply for some space in the heap directly to fit in an integer using some syntax like `int i = new int();`. (While in C or C++ a programmer is allowed to allocate any size of heap in any primitive type, such as `int * i = (int *) malloc(sizeof(int))`.) However, a Java user is able to write `Integer i = new Integer();` to assign some heap space for an object of `Integer` which contains a primitive integer.

To simulate Java heap space, our type of heap data has two levels of meaning. First, each allocated heap cell must be assigned a base type for the primitive value stored in that cell. Moreover, since such values should be encapsulated as objects' fields, a reference, whose value corresponds to an object in the heap, is linked to a series of contiguous heap cells holding the fields of that object.[2]

As a result, our lifting of the heap model with types comprises two main issues accordingly. The first is to assign a base type for each primitive value located in any allocated heap space. The second is to assign a class type for each object allocated in the heap to enclose the primitive values as the fields of the object.

First the types for primitive values are added to the heap as follows:

$$\begin{array}{rcl}\textit{heap-type} & :: & \mathit{Address} \rightharpoonup \mathit{BaseType} \cup \{\mathsf{N}, \bot\}\\ \textit{heap-type}\ r & = & \begin{cases}\bot, \text{if } \textit{heap}\ r = \bot;\\ t \in \mathit{BaseType}, \text{if } \textit{heap}\ r \text{ is the first byte}\\ \qquad\qquad \text{of a value typed } t;\\ \mathsf{N}, \text{otherwise.}\end{cases}\end{array}$$

Thus for any allocated heap portion corresponding to a primitive value, the type of its first byte will be stamped as its base type, while the rest bytes are marked N.

For the analysis of a particular bytecode program, the type of any primitive value on the heap can be inferred from

[2]Note that the Java virtual machine requires the continuity of an object's heap space.

the field descriptors of a class (since this value itself must be a field defined in some class). Therefore, during the process of analysis, when an instance of such class is created on the heap, the location and size information of all its fields can be deducted from both the instance's descriptors and the reference to it. For this purpose, a mapping from heap address to the type of the object located in that address is maintained statically:

$$\textit{ref-type} :: \textit{Address} \rightharpoonup \textit{Type}$$

where *Type* is the set of all classes used in a program. If the reference has multiple types which consist of one class $C$ and all its inherited classes (but no classes extending $C$), then *ref-type* will map it to its exactly defined type in the program, say, a reference defined as `java.util.List l;` will have the type Ljava/util/List, even it is instantiated as a linked list. Figure 1 (b) exemplifies this typed heap.

As a semantic support of the later separation logic predicate "pointing-to", we will introduce the field offsets of a class. By scanning a bytecode program, we may achieve a mapping from classes in that program to their fields and the types of their fields:

$$\mathsf{fields\text{-}of} :: \textit{Type} \rightarrow [\textit{Field} \times \textit{BaseType}]$$

where *Field* is the set of possible field names, and all the references (regardless of objects of other classes or arrays) are treated as of base types. We also make an assumption that the fields are in the order which is maintained in the memory by the Java virtual machine, for example, some Java virtual machine will keep an ascending sequence of an object's fields in memory by the sizes of the fields. As an illustration, a class definition `class C {int x; byte y; long z;}` will be mapped to a list $[(\mathtt{y}, \mathsf{B}), (\mathtt{x}, \mathsf{I}), (\mathtt{z}, \mathsf{J})]$.

With the fields-of mapping, the reference to a field of an object on the heap (referred to by $r$) can be computed using the functions below:

$$\begin{array}{l}
\mathsf{offset} :: [\textit{Field} \times \textit{BaseType}] \rightarrow \textit{Field} \rightarrow \textit{Address} \rightharpoonup \textit{Address} \\
\mathsf{offset}\ [\,]\ f\ n = \bot \\
\mathsf{offset}\ ((f,t) : fs)\ f\ n = 0 \\
\mathsf{offset}\ ((f',t) : fs)\ f\ n = \mathsf{if\ size\text{-}of}\ t \leq n\ \mathsf{then} \\
\qquad \mathsf{size\text{-}of}\ t + \mathsf{offset}\ fs\ f\ (n - \mathsf{size\text{-}of}\ t) \\
\quad \mathsf{else}\ n + \mathsf{offset}\ ((f',t) : fs)\ f\ \mathsf{alignment} \\
\mathsf{field\text{-}ref} :: \textit{RefType} \rightarrow \textit{Address} \rightarrow \textit{Field} \rightharpoonup \textit{Address} \\
\mathsf{field\text{-}ref}\ rt\ r\ f = r + \mathsf{reserved} \\
\qquad\qquad + \mathsf{offset}\ (\mathsf{fields\text{-}of}(rt\ r))\ f\ \mathsf{alignment}
\end{array}$$

where the *RefType* is the set of all reference type mappings, and reserved is the size for an object of the type Ljava/lang/Object, whose value is often alignment in implementation.

In a similar way as above, the size of an object during runtime can be calculated as:

$$\begin{array}{l}
\mathsf{list\text{-}size} :: [\textit{Field} \times \textit{BaseType}] \rightarrow \mathbb{N} \\
\mathsf{list\text{-}size}\ [\,]\ n = 0 \\
\mathsf{list\text{-}size}\ ((f,t) : fs)\ n = \mathsf{if\ size\text{-}of}\ t \leq n\ \mathsf{then} \\
\qquad \mathsf{size\text{-}of}\ t + \mathsf{list\text{-}size}\ fs\ (n - \mathsf{size\text{-}of}\ t) \\
\quad \mathsf{else}\ n + \mathsf{list\text{-}size}\ ((f,t) : fs)\ \mathsf{alignment} \\
\mathsf{ref\text{-}size} :: \textit{Type} \rightarrow \mathbb{N} \\
\mathsf{ref\text{-}size}\ t = \ \mathsf{let}\ s = \mathsf{list\text{-}size}\ (\mathsf{fields\text{-}of}\ t)\ \mathsf{in} \\
\qquad\qquad \mathsf{if\ alignment} \mid s\ \mathsf{then\ reserved} + s \\
\qquad\qquad \mathsf{else\ reserved} + s + \mathsf{alignment} \\
\qquad\qquad\quad - s\ \mathsf{mod\ alignment\ end}
\end{array}$$

When a program allocates new heap space, or all the references to an address are eliminated (so it should be collected as garbage), the mappings mentioned above are to be modified with the following process:

$$\begin{array}{l}
\mathsf{heap\text{-}type\text{-}list\text{-}clear} :: \textit{HeapType} \rightarrow [\textit{Address}] \rightarrow \textit{HeapType} \\
\mathsf{heap\text{-}type\text{-}list\text{-}clear}\ ht\ [\,] = ht \\
\mathsf{heap\text{-}type\text{-}list\text{-}clear}\ ht\ (r : rs) = \mathsf{heap\text{-}type\text{-}list\text{-}clear} \\
\qquad\qquad\qquad\qquad\qquad ht(r \mapsto \mathsf{N})\ rs \\
\mathsf{heap\text{-}type\text{-}write} :: \textit{HeapType} \rightarrow \textit{Address} \rightarrow \textit{BaseType} \\
\qquad\qquad \rightarrow \textit{HeapType} \\
\mathsf{heap\text{-}type\text{-}write}\ ht\ r\ t = \mathsf{heap\text{-}type\text{-}list\text{-}clear}\ ht(r \mapsto t) \\
\qquad\qquad\qquad\qquad [r+1, \ldots, r-1+\mathsf{size\text{-}of}(ht\ r)]
\end{array}$$

where the *HeapType* contains all possible heap type mappings. The functions first "clear" the type mapping to N, and then write the first byte's mapping as $t$. Compared with this, the function to change the reference's type mapping is simpler, which only needs to modify the only byte's mapping:

$$\begin{array}{l}
\mathsf{ref\text{-}type\text{-}write} :: \textit{RefType} \rightarrow \textit{Address} \rightarrow \textit{Type} \rightarrow \textit{RefType} \\
\mathsf{ref\text{-}type\text{-}write}\ rt\ r\ t = rt(r \mapsto t)
\end{array}$$

However, when reasoning about the allocating or reclaiming an object, both functions must be utilized.

The type information allows us to define type-related guards on heaps. A type-safe guard on a field reference guarantees that each byte of the value pointed to by this reference has the correct mapping of *heap-type*:

$$\begin{array}{l}
\mathsf{heap\text{-}safe}\ ht\ r = \exists t \in \textit{BaseType}\ .\ ht\ r = t\ \wedge \\
\qquad\qquad \forall s \in \{r+1, \ldots, r-1+\mathsf{size\text{-}of}\ t\}\ .\ ht\ s = \mathsf{N}
\end{array}$$

which can still be lifted to another guard, saying that all of an object's fields are correctly mapped:

$$\begin{array}{l}
\mathsf{heap\text{-}list\text{-}safe}\ ht\ rt\ r\ [\,] = \mathsf{true} \\
\mathsf{heap\text{-}list\text{-}safe}\ ht\ rt\ r\ (f : fs) = \mathsf{heap\text{-}safe}\ ht \\
\qquad\qquad (\mathsf{field\text{-}ref}\ rt\ r\ f) \wedge \mathsf{heap\text{-}list\text{-}safe}\ ht\ rt\ r\ fs \\
\mathsf{ref\text{-}safe}\ ht\ rt\ r = \mathsf{heap\text{-}list\text{-}safe}\ ht\ rt\ r\ (\mathsf{fields\text{-}of}(rt\ r))
\end{array}$$

Moreover, the Java virtual machine requires certain alignment of the memory addresses referred to by the references, which results in that each object's size must be a

multiple of the alignment. A guard can be defined to guarantee this as

$$\mathsf{ref\text{-}aligned}\ r = \mathsf{alignment} \mid r$$

and it should be ensured that this guard holds for the creation of a new object.

With the information of both base and object types added, we have founded a typed heap model lifted from the byte-based low-level heap. This model guarantees the type-safety and the alignment properties for bytecode execution, and is ready to serve as the semantics of our separation logic system.

## 4 Separation Logic

Separation logic [7, 12] is now prevalent in reasoning about programs mainly concerned with reference-based heap models. It enhances the original Hoare logic with the new logical operator, separation conjunction, to express heap-related program states. The separation conjunction follows the idea that the heap can be divided into disjoint parts, and each part can be described by an assertion. If so, then the conjunction of such assertions is a "separation conjunction" of them since the domains of the assertions are separated. In this way pointer aliases can be specified explicitly with ease, and program reasoning can also be restricted to a minimum heap part without interference of other parts.

Since separation logic is a powerful contender for formal reasoning of heap-manipulating imperative programs, we have utilized it in our former works to construct our verification framework for heap properties. The original intention of this work is to provide a semantic model for our separation logic system to be built on and to verify and infer heap-related properties, such as size of data structures which is closely linked to program memory consumption.

In this section, an embedding of separation logic over this memory model will be introduced. To begin with, the assertion language for our separation logic is as follows:

$$\begin{aligned} P =_{df}\ & \mathsf{true} \mid \mathsf{false} \mid \neg P \mid P \wedge P \mid P \vee P \mid P \Rightarrow P \mid r = r' \mid \\ & P * P \mid P -\!\!* P \mid \mathsf{emp} \mid r :: C \mid r.f \mapsto r' \mid r.f \hookrightarrow r' \end{aligned}$$

Generally, for Java bytecode verification, the assertions should describe three aspects of program runtime status: the operand stack, the variable stack and the heap. To focus on our real interest, we will pay less attention to the first part (which is well investigated by other works [4, 3]), and hence the assertions used in our system are interpreted as functions on *Heap*, *HeapType* and *RefType* to return a boolean value, which mainly concern the heap status of the program.

emp states that no part of the heap is allocated and thus all the mappings are undefined:

$$\mathsf{emp} = \lambda\, h\ ht\ rt\ .\ \forall\, r\ .\ h\ r{=}\bot \wedge ht\ r{=}\bot \wedge rt\ r{=}\bot$$

$r :: C$ is a novel assertion to specify that an object of class $C$ resides at $r$. Traditional separation logic does not have this since its model has no type information. It is defined as

$$r{::}C = \lambda\, h\ ht\ rt\ .\ rt\ r{=}C$$

Or we can still require that such assertion be safer with the following restricted definition

$$r{::}C = \lambda\, h\ ht\ rt\ .\ rt\ r{=}C \wedge \mathsf{ref\text{-}safe}\ ht\ rt\ r \wedge \mathsf{ref\text{-}aligned}\ r$$

which depends on user demand.

According to the definition in separation logic, the semantics of the singleton heap assertion $r.f \mapsto r'$ is that the heap contains exactly one cell located at $r'$:

$$\begin{aligned} r.f \mapsto r' = \lambda\, h\ ht\ rt\ .\ & \mathsf{heap\text{-}read}\ h \\ & (\mathsf{field\text{-}ref}\ rt\ r\ f)\ (ht\ r) = r'\ \wedge \\ & \mathsf{dom}\, rt = \{r\}\ \wedge\ \mathsf{dom}\, h = \mathsf{dom}\, ht = \\ & \{r + \mathsf{reserved}, \ldots, r - 1 + \mathsf{ref\text{-}size}\ (\mathsf{rt}\ \mathsf{r}')\} \end{aligned}$$

For the separation conjunction and implication, their semantics are direct translations from the classical ones to a version in this model. First comes the definition of two operations over the three mappings for our model.

$$\begin{aligned} h_1 \bot h_2 = {} & \mathsf{dom}\, h_1 \cap \mathsf{dom}\, h_2 = \emptyset \\ h_1 \uplus h_2 = {} & \{r \mapsto a \mid r{\in}\, \mathsf{dom}\, h_1 \cup \mathsf{dom}\, h_2 \wedge (r{\in}\, \mathsf{dom}\, h_1 \\ & \Rightarrow a{=}h_1\ r \wedge r{\notin}\, \mathsf{dom}\, h_1 \Rightarrow a{=}h_2\ r)\} \end{aligned}$$

where the first one describes that the two heaps (modeled as mappings) are disjoint, and the second is a union of two (disjoint) heaps. Note that these operations are polymorphic and thus are applicable for all three mappings (bytes and types). They assist in the expression of the separation connectives semantics:

$$\begin{aligned} P * Q = {} & \lambda\, h\ ht\ rt\ .\ \exists\, h_1\ h_2\ ht_1\ ht_2\ rt_1\ rt_2\ .\ h_1 \bot h_2 \wedge \\ & h{=}h_1 \uplus h_2 \wedge ht_1 \bot ht_2 \wedge ht{=}ht_1 \uplus ht_2 \wedge rt_1 \bot rt_2 \\ & \wedge rt{=}rt_1 \uplus rt_2 \wedge P\ h_1\ ht_1\ rt_1 \wedge Q\ h_2\ ht_2\ rt_2 \\ P -\!\!* Q = {} & \lambda\, h\ ht\ rt\ .\ \forall\, h'\ ht'\ rt'\ .\ h \bot h' \wedge ht \bot ht' \wedge rt \bot rt' \\ & \wedge P\ h'\ ht'\ rt' \Rightarrow Q\ (h \uplus h')\ (ht \uplus ht')\ (rt \uplus rt') \end{aligned}$$

The following predicates do not increase the expressiveness of the logic, but are present for convenience or historical reasons. The first is the intuitive "pointing-to" $r.f \hookrightarrow r'$, which is equivalent as

$$r.f \hookrightarrow r' = r.f \mapsto r' * \mathsf{true}$$

where true has its meaning based on the heap. And the following existential "pointing-to"'s (the pointing-to assertions with the heap value pointed to being existentially quantified) are also useful:

$$\begin{aligned} r.f \mapsto - &= \lambda\, h\ ht\ rt\ .\ \exists r'\ .\ (r.f \mapsto r')\ h\ ht\ rt \\ r.f \hookrightarrow - &= \lambda\, h\ ht\ rt\ .\ \exists r'\ .\ (r.f \hookrightarrow r')\ h\ ht\ rt \end{aligned}$$

In such separation logic, the Hoare triples are in the form

$$\{P\ h\ ht\ rt\}\ S\ \{Q\ h\ ht\ rt\}$$

which has a canonical semantics based on our storage model. We may omit the parameters to abbreviate them as $\{P\}\ S\ \{Q\}$ for convenience if no confusion is caused.

For the verification of Java bytecode, the separation logic rules are presented according to bytecode instructions. Here we still concentrate on the heap-related rules. First is the $new$ rule to allocate a new object on the heap:

$$\frac{}{\{\mathtt{emp}\}\ \mathtt{new}\ C\ \{\exists\ r\ .\ r :: C\}}\ (new)$$

This is a local rule which only considers the newly allocated part of the heap. Later we will see that it can be extended to the global heap using the $frame$ rule. Meanwhile, since Java virtual machine specification does not specify any memory management strategy, the newly allocated $r$ is existentially quantified instead of calculated as a concrete value.

The $assignment$ rule (of object fields) in this separation logic system is:

$$\frac{}{\{r.f \mapsto -\}\ \mathtt{putfield}\ r.f\ \{r.f \mapsto e\}}\ (assign\text{-}1)$$

under the assumption that $r.f$ and $e$ are loaded onto the operand stack. Since the operation does not change any other part of the heap except for the $f$ of $r$, this local version of assignment can be extended to a global one as

$$\frac{}{\{r.f \mapsto - * R\}\ \mathtt{putfield}\ r.f\ \{r.f \mapsto e * R\}}\ (assign\text{-}2)$$

where $R$ cannot mention $r.f$ for the assertion to hold. Meanwhile, like the classical Hoare logic, we provide yet another backward $assignment$ rule:

$$\frac{}{\{r.f \mapsto - * (r.f \mapsto e \mathrel{-\!\!*} P)\}\ \mathtt{putfield}\ r.f\ \{P\}}\ (assign\text{-}3)$$

The global $assignment$ rule is an instantiation of the local assignment and another rule of particular significance, viz. the $frame$ rule, which is the bridge over local and global reasonings. It is in the following form

$$\frac{\{P\}\ S\ \{Q\}}{\{P * R\}\ S\ \{Q * R\}}\ (frame)$$

where $\mathrm{mods}(S) \cap \mathrm{dom}(R) = \emptyset$
and $\mathrm{deps}(S) \cap \mathrm{dom}(R) = \emptyset$.

Here $\mathrm{mods}(S)$ and $\mathrm{deps}(S)$ represent the very part of heap, heap type and reference type mappings that $S$ modifies and depends on, respectively. This rule guarantees that $S$ never modifies the (part of) heap mapping or heap/reference type mappings referred to freely by $R$, and it never depends on the two type mappings inside the domain of $R$ in any of its expression. With this rule, when we reason about a triple, we can safely ignore the parts of the pre- and postconditions irrelative to $S$ to focus on the minimum heap part that $S$ modifies and/or depends on. For example, when verifying a specification of a method which invokes other methods, if we know that these method calls refer to disjoint parts of the heap, then we can verify those method specifications separately and conjoin them afterwards.

```
class Node { int value; Node next; ... }
class List {
  private Node head;
  public void add(int i) {
    Node n = new Node(i, head);
    this.head = n;
  }
  public void addTwo(int i) {
    add(i);
    add(i);
  }
}
```

**Figure 2. Java source code example**

For modular reasoning, we also need a rule to verify method invocations initiated by bytecode instruction `invokevirtual`. Supposing that we have already verified the specification of the method to be invoked, we can use it to verify the invocation, with some simple variable substitutions:

$$\frac{\{P\}\ C.\mathtt{m} : (\overline{x})\ \{Q\}}{\{\theta(P)\}\ \mathtt{invokevirtual}\ y.\mathtt{m}\ \{\theta(Q)\}}\ (inv\text{-}v)$$

where $\theta = [\overline{e}, y\ /\ \overline{x}, \mathtt{this}]$ and $\overline{e}$ should be loaded onto operand stack.

And the class constructor's invocation is a special case of virtual method invocation, where the $\theta$ and $\overline{e}$ are still the same as above:

$$\frac{\{P\}\ C.\texttt{"<init>"} : (\overline{x})\ \{Q\}}{\{\theta(P)\}\ \mathtt{invokespecial}\ y.\texttt{"<init>"}\ \{\theta(Q)\}}\ (inv\text{-}s)$$

Here is an example to illustrate the verification performed by our system. Consider the Java source code in Figure 2 where `List.add` will add one node to the head of the list, and `List.addTwo` will add two nodes to the list's head at once. Figure 3 shows its compiled bytecode.

Given that class `Node`'s constructor initializes its two fields `value` and `next` with the values from parameters, the methods `List.add` and `List.addTwo` in the bytecode may have the following triples as their specifications, respectively:

$$\{\mathtt{this.head} \mapsto r\}$$
$$\mathtt{List.add(int\ i)}$$
$$\{\exists\ r_1\ .\ \mathtt{this.head} \mapsto r_1 * r_1\mathtt{.next} \mapsto r\}$$

```
public void add(int);
 Code:
   1  new             Node;
   2  dup
   3  iload_1
   4  aload_0
   5  getfield        head:LNode;
   6  invokespecial   Node."<init>":(ILNode;)V;
   7  astore_2
   8  aload_0
   9  aload_2
  10  putfield        head:LNode;
  11  return

public void addTwo(int);
 Code:
   1  aload_0
   2  iload_1
   3  invokevirtual   add:(I)V;
   4  aload_0
   5  iload_1
   6  invokevirtual   add:(I)V;
   7  return
```

**Figure 3. Corresponding bytecode example**

$$\begin{array}{c}\{\texttt{this.head} \mapsto r\} \\ \texttt{List.addTwo(int i)} \\ \{\exists\, r_1, r_2 \,.\, \texttt{this.head} \mapsto r_1 \\ \quad * \, r_1\texttt{.next} \mapsto r_2 * r_2\texttt{.next} \mapsto r\}\end{array}$$

Here we present a sketch of forward verification of these specifications. First, for the verification of `List.add`, using $new$ rule for Line 1, we have

$$\{\mathsf{emp}\}\ \texttt{new Node}\ \{\exists\, r_1 \,.\, r_1 :: \texttt{Node}\}$$

With the preparation from Line 2 to Line 5 (which has little direct relevance with the heap) and the $inv$-$s$ rule to link the specification of `Node."<init>"` with the call site, we get for Line 6 its postcondition as

$$\exists\, r_1 \,.\, r_1\texttt{.next} \mapsto r$$

Again, Lines 7 to 9 are not related to the heap, but their effects on the operand stack can be verified by other rules out of the scope of this paper. Following these lines, Line 10 can be verified using the $assignment$ rule, with the postcondition

$$\exists\, r_1 \,.\, \texttt{this.head} \mapsto r_1 * r_1\texttt{.next} \mapsto r$$

to complete the verification for `List.add`.

Verification for method `List.addTwo` mainly utilizes the $inv$-$v$ rule to discharge the two subgoals concerning the two `invokevirtual`'s. The postcondition of the first follows directly the specification of `List.add` as above. The second `invokevirtual` can also be verified in a similar manner, except that, to change global reasoning to local reasoning, the $frame$ rule should be used as

$$\frac{\begin{array}{c}\{\texttt{this.head} \mapsto r_2\}\ \texttt{invokevirtual} \\ \{\exists\, r_1 \,.\, \texttt{this.head} \mapsto r_1 * r_1\texttt{.next} \mapsto r_2\}\end{array}}{\begin{array}{c}\{\exists\, r_2 \,.\, \texttt{this.head} \mapsto r_2 * r_2\texttt{.next} \mapsto r\} \\ \texttt{invokevirtual} \\ \{\exists\, r_1, r_2 \,.\, \texttt{this.head} \mapsto r_1 \\ \quad * \, r_1\texttt{.next} \mapsto r_2 * r_2\texttt{.next} \mapsto r\}\end{array}}$$

to gain the postcondition of Line 6:

$$\begin{array}{l}\exists\, r_1, r_2 \,.\, \texttt{this.head} \mapsto r_1 \\ \quad * \, r_1\texttt{.next} \mapsto r_2 \ * \ r_2\texttt{.next} \mapsto r\end{array}$$

which is also the postcondition of the method specification.

Since our semantic model is loosely coupled with the axiomatic verification system, the latter is still extensible by adding new assertion constructs, as long as the new construct does not exceed the expressiveness of the lower levels of the model. For example, we may define used-heap-size $s$ to specify the heap size that is consumed by the program as

$$\begin{array}{l}\mathsf{ref\text{-}size}\ \bot = 0 \\ \mathsf{size\text{-}count} :: \mathit{RefType} \to \mathit{Address} \to \mathbb{N} \\ \mathsf{size\text{-}count}\ rt\ r = \mathsf{if}\ r < \mathsf{max\text{-}addr}\ \mathsf{then}\ \mathsf{ref\text{-}size}\ (rt\ r) + \\ \qquad\qquad \mathsf{size\text{-}count}\ rt\ (r{+}1) \\ \qquad\qquad \mathsf{else}\ \mathsf{ref\text{-}size}\ (rt\ r) \\ \mathsf{used\text{-}heap\text{-}size}\ s = \lambda\, h\ ht\ rt\ .\ \mathsf{ref\text{-}size}\ rt\ 0 = s\end{array}$$

where we first redefine that for ref-size, the "undefined" reference types have size zero, to avoid the judgment whether an address has a reference type defined on it. Then the function size-count should be incorporated in the lower model (typed heap) to support our newly added assertion used-heap-size. With this assertion, another version of method specification for `List.addTwo` can be written as

$$\begin{array}{c}\{\mathsf{used\text{-}heap\text{-}size}\ s\} \\ \texttt{List.addTwo(int i)} \\ \{\mathsf{used\text{-}heap\text{-}size}\ (s + 2 \times \mathsf{ref\text{-}size}\ \texttt{Node})\}\end{array}$$

which may be verified in an analogous way to the one above, with an extra effort for the theorem prover to infer that the size-count of the heap has its value changed by the two `add`'s in the method body.

## 5 Related Work

As is aforementioned, this work is part of an extension for our former works [4, 11, 3]. Our direct motivation to build a low-level model for Java bytecode comes from Tuch et al. [14], which has a similar idea of such a model for

the C language. Compared with their work, our model depicts not only the base types of the heap, but also the reference types due to the object-oriented essence of Java. Also our verification rules are based on bytecode instructions but not source code. The original storage model for separation logic, proposed in Reynolds [12], could be substituted with ours in order to simulate the memory usage of bytecode execution to get more precise bounds of heap consumption.

There are many approaches for the verification of bytecode and for its memory consumption. Due to Leroy [9], the dataflow analysis is a significant means utilizing the type algebra to construct an order over the subtyping relationship, which is a bit complicated and calls for too much resource for memory requirement analysis, however. Therefore, as a substitute, Klohs and Kastens [8] attempts to use the proof carrying code technique to embed proof of safety rules and properties into the compiled bytecode and provides an mechanism to check the proof on-card. Giambiagi and Schneider [6] provides a slightly different solution with a model of constraints on bytecode instructions and an algorithm to operate on that model, which is similar as a CSP model on bytecode to some extent. Albert has two papers [1, 2] concentrating on the heap space consumption of a bytecode program by constructing the control flow graph of that program and changing it to some equations and relations to solve. Compared with their works, ours also focuses on the heap memory and is adept at reasoning about the reference relationship of objects and their fields, and heap size consumption of the program. With the help of the framework built up by Nguyen et al. [11], our system will be capable to deal with recursively defined data structures and their related algorithms.

## 6 Conclusion

In this paper we have proposed a low-level heap model as the storage semantics for Java bytecode, which allows us to reason about the size and alignment properties of primitive values. Then this model is lifted with both base types and user-defined classes to support type-related reasoning, such as type and alignment safety. We have also described a verification system upon separation logic whose assertions are interpreted with such heap model with types, to verify the heap-related properties of bytecode programs.

This paper is part (and the semantic foundation) of our main work at present, which consists of an integrated framework to analyze and verify the memory consumptions of Java bytecode. As we have already completed some work to verify programs' shape and size properties based on the traditional heap model of separation logic [4, 11, 3], the next step is to extend such work with the model presented in this paper to improve its expressiveness and precision.

## References


[1] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of Java bytecode. In *Proceedings of 16th European Symposium on Programming* (ESOP'07). LNCS 4421, Springer, March 2007.

[2] E. Albert, S. Genaim, and M. Gómez-Zamalloa. Heap space analysis for Java bytecode. In *Proceedings of the 6th International Symposium on Memory Management* (ISMM'07), pages 105–116, Montréal, Québec, Canada, October 2007.

[3] W.-N. Chin, H. Nguyen, C. Popeea, and S. Qin. Analysing memory resource bounds for low-level programs. In *Proceedings of the 7th International Symposium on Memory Management* (ISMM'08), Tucson, Arizona, USA, June 2008.

[4] W.-N. Chin, H. Nguyen, S. Qin, and M. Rinard. Memory usage verification for OO programs. In *Proceedings of the 12th International Static Analysis Symposium* (SAS'05), London, UK, 2005.

[5] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java(tm) language specification*. Addison Wesley, third edition, 2005.

[6] P. Giambiagi, and G. Schneider. Memory consumption analysis of Java smart cards. In *Proceedings of the XXXI Latin American Informatics Conference* (CLEI'05), pages 12–23, Cali, Colombia, October 2005.

[7] S. Isthiaq and P. O'Hearn. BI as an assertion language for mutable data structures. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL'01), London, UK, January 2001.

[8] K. Klohs, and U. Kastens. Memory requirements of Java bytecode verification on limited devices. *Electronic Notes in Theoretical Computer Science*, 132 (2005): 95–111.

[9] X. Leroy. Java bytecode verification: an overview. In *Proceedings of the 13th International Conference on Computer Aided Verification* (CAV'01). LNCS 2102, Springer, 2001.

[10] T. Lindholm and F. Yellin. *The Java(tm) virtual machine specification*. Addison Wesley, second edition, 1999.

[11] H. Nguyen, C. David, S. Qin, and W.-N. Chin. Automated verification of shape and size properties via separation logic. In *Proceedings of the 8th International Conference on Verification, Model Checking and Abstract Interpretation* (VMCAI'07), Nice, France, January 2007.

[12] J. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proceedings of 17th Annual IEEE Symposium on Logic in Computer Science*. Springer-Verlag, 2002.

[13] R. Stärk, J. Schmid, and E. Börger. *Java and the Java virtual machine — definition, verification, validation*. Springer-Verlag, 2001.

[14] H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL'07), Nice, France, January 2007.