

# A Verifier for Region-Annotated Java Bytecodes

Sigmund Cherem and Radu Rugina

{siggi,rugina}@cs.cornell.edu  
Computer Science Department  
Cornell University  
Ithaca, NY 14850

---

## Abstract

This paper presents a verifier for the memory-safe execution of extended Java bytecodes that support region-based memory management and explicit deallocation primitives. The verifier reads in region-annotated bytecodes that augment the standard Java bytecodes with instructions for creating and removing memory regions, allocating objects in regions, and passing regions as parameters. The verification ensures that each region is live when objects in the region are in use and that the program does not follow dangling references.

The verifier requires region-safety certificates to be provided along with the bytecodes. The verification process consists of a load-time verification of method bodies, and a lazy linkage verification of method calls. Our region system supports both regions that are not lexically scoped and dangling pointers; the verifier proposed in this paper can successfully handle both of these features. Our experiments indicate that the sizes of certificates are acceptable in practice, and that region verification incurs little run-time overhead.

*Key words:* bytecode verification, region-based memory management

---

## 1 Introduction

A system with region-based memory management groups objects together in regions and then deallocates all of the objects in a region at once. Regions have a number of appealing properties, including data locality and efficient collective deallocation. More importantly, a compiler can statically enable region-based memory management: it can automatically determine how to group objects in regions and when to deallocate regions. This approach is especially attractive for real-time systems, which cannot afford to be interrupted for unbounded amounts of time by a garbage collector.

*This is a preliminary version. The final version will be published in  
Electronic Notes in Theoretical Computer Science  
URL: [www.elsevier.nl/locate/entcs](http://www.elsevier.nl/locate/entcs)*

In recent work, we have proposed Jreg, a region analysis and transformation system for Java programs [6] that translates input Java programs into output programs with region-based memory management. In our system, regions are not lexically scoped, so they can be created or removed<sup>1</sup> at any point in the control-flow. Furthermore, we allow dangling pointers, but require that the program never follows such references. Both features increase the accuracy and flexibility of our system, but makes it more challenging for a compiler to identify region lifetimes, and for a verifier to check them.

The region transformation in our system is a bytecode-level transformation and produces region annotated bytecodes. The generated bytecodes are executed in an extended virtual machine with run-time support for regions. However, there are no run-time checks to ensure that regions are live when the program accesses objects in those regions. Therefore, a program that incorrectly deallocates a region while objects in that region are still in use will cause the system to crash.

In this paper we present a verifier for region-annotated bytecodes. The goal of the verifier is to ensure that regions are live whenever the program accesses objects in those regions, and that the program never follows dangling pointers. Such a verifier can be used to check that the results of our region transformation are safe. Since the analysis and transformation algorithms in our compiler are fairly complex, their implementation is error-prone; in fact, the region verifier presented here has already pointed out implementation bugs in our compiler. Furthermore, the verifier could also check the correctness of region-annotated bytecodes generated by other systems. The verification process presented in this paper is aimed at verifying region manipulations only; all of the other bytecode-level checks are carried out by the standard bytecode verifier.

We present a dataflow verification algorithm that checks the region safety of each method in the program. In order to verify methods in the program, the verifier requires region certificates to be provided along with the bytecodes. The information in these certificates describes various method effects, including region aliasing and region accesses; from a proof-carrying code perspective [17], this region information can be regarded as a “proof” that the transformation is correct. We also present an algorithm for the lazy linkage verification of virtual method calls, to ensure that method calls can be safely executed. Each method call is checked only the first time it is executed.

In contrast to the approach presented in this paper, most of the existing region-based systems [20,8,12,4,7] use type systems to ensure the safety of region manipulations. However, an approach based on dataflow analysis is more appropriate in our system for two reasons. First, our system uses regions that are not lexically scoped and allows dangling references. Therefore, it is

---

<sup>1</sup> We use the terms *region creation* and *removal*, instead of *allocation* and *deallocation*, in order to avoid ambiguities between object allocation and region allocation.

not possible to determine region lifetimes using the syntactic structure of the program. Second, we operate at the bytecode level, where local variables are untyped and may hold values of different types at different points; again, a flow analysis is required to handle such situations. To the best of our knowledge, this is the first approach to check the correctness of region-based programs at the bytecode level.

The rest of the paper is structured as follows. Section 2 presents an overview of the Jreg system. Section 3 shows an example program with region annotations and Section 4 introduces the region bytecodes in our system. Section 5 presents the verification process in detail. Section 6 discusses experimental results. Finally, we discuss related work in Section 7 and conclude in Section 8.

## 2 Jreg: A Region-Based Java System

The Jreg system is an analysis and transformation system that provides region-based memory management for Java programs [6]. It consists of a region compiler and a virtual machine with region run-time support.

The region compiler is implemented in the Soot infrastructure [21] as a bytecode-level transformation: given an input program, the compiler automatically produces an output program with region constructs. The transformed program explicitly creates new regions, places objects in regions, passes regions as parameters, and explicitly removes regions. The analyses are implemented in the three-address code intermediate representation in Soot. The region-annotated bytecodes produced by the region compiler are then executed in an extended virtual machine that supports region-annotated bytecodes and provides region run-time memory management. The region virtual machine is built on top of the interpreter of the open-source Kaffe VM [22].

The region compiler performs the transformation in three steps. First, it uses a flow-insensitive, but context-sensitive points-to analysis to produce a region points-to graph for each method. The nodes in the graph represent regions and the edges represent points-to relations. Second, the system uses a flow-sensitive region liveness analysis to determine region lifetimes. Finally, it uses the computed points-to graphs and region lifetimes to place region annotations in the program. The goals of the transformation are to ensure that, whenever the program accesses an object, the region that contains the object is live; and, at the same time, to statically minimize region lifetimes.

The system also uses run-time support for exceptions and multithreading. Whenever objects in a region are concurrently accessed by multiple threads, each of the threads removes the region when it no longer needs it. The run-time system uses thread counters for shared regions to ensure that only the last thread actually removes the region. Finally, when exceptions are thrown, the exception run-time system ensures that local regions are being removed as it walks up the program stack.

### 3 Example

We illustrate the kind of region-based programs that our system supports using an example. Figure 1 shows a region-annotated program produced by our region compiler; the input program is the same, but without the region annotations. The goal of the verifier is to successfully check that the execution of this program is memory-safe. For readability, we show the Java version of the program, instead of bytecodes. This program manipulates linked lists: method `make` iteratively constructs a list of `Integer` objects; method `reverses` recursively reverses a list; and method `test` invokes these methods to perform a simple list computation.

The region annotations are self-explanatory: `create` dynamically creates one or more regions, `remove` dynamically removes regions; the clauses “`in r`” on object allocations indicate the regions where new objects will be placed; and region arguments in angle brackets represent regions being passed from caller to callee methods. Therefore, each method receives two kinds of parameters: the standard Java arguments, and the region parameters.

Method `make` has two region parameters `r1` and `r2`. The method expects these regions to be created by its caller; when invoked, `make` will place the new list in these regions: `r1` will hold the spine of the list, and `r2` will hold the integer objects representing the list data.

Method `reverse` recursively reverses the list in the receiver object. The reversed list is a fresh list, but has the same data objects as the original list. This method receives region `r3` as parameter and places the spine of the reversed list in this region. In the method body, region `r3` is being passed as actual parameter at the recursive call. Although the reversal method accesses objects in the original list, there is no need to pass the region holding that list to `reverse`, since `reverse` does not explicitly refer to this region.

Finally, method `test` invokes `make` to create a list of 10 elements, reverses the list, and prints the contents of the reversed list. The method creates regions `r6` (for the list spine) and `r7` (for the list elements), passes them to `make`, and places the result in `x`. The program then creates a region `r8` for the reversed list, passes it to `reverse`, and stores the result in `y`. After the call to `reverse`, the spine of the old list `x` in region `r6` is no longer needed and can be discarded. After printing the result, regions `r8` and `r7` can also be reclaimed. Note that regions `r6` and `r8` are partially overlapping, and we can accurately describe their lifetimes just because regions are not lexically scoped.

The table in Figure 2 shows the key pieces of information that the region compiler uses to generate this program. This information is also key in the verification process and represents most of the information in the region certificate that must be provided along with the bytecodes. It includes region points-to information and information about different classes of regions. The points-to information describes the aliasing relations between regions using triples of the form  $(r, f, r')$  indicating that the field  $f$  of any object in region

```

class List {
  Object data;
  List next;
  static List make<r1,r2>(int n){
    List t, y = null;
    while (n-- > 0) {
      t = new List() in r1;
      t.data = new Integer(n) in r2;
      t.next = y; y = t;
    }
    return y;
  }
  List reverse<r3>(){
    List t = new List() in r3;
    t.data = data;
    if (next == null)
      t.next = null;
    else
      t.next = next.reverse<r3>();
    return t;
  }
  void test() {
    create r6,r7;
    List x = make<r6,r7>(10);
    create r8;
    List y = x.reverse<r8>();
    remove r6;
    for (; y != null; y = y.next)
      System.out.println(y.data);
    remove r7,r8;
  }
}

```

Fig. 1. Example: region-based list manipulation

<i>Method</i>	make	reverse	test
<i>Points-to relations</i>	(r1,next,r1) (r1,data,r2)	(r3,next,r3) (r3,data,r5) (r4,next,r4) (r4,data,r5)	(r6,next,r6) (r6,data,r8) (r7,next,r7) (r7,data,r8)
<i>Local regions</i>	{}	{}	{r6,r7,r8}
<i>Incoming regions</i>	{r1,r2}	{r3,r4,r5}	{}
<i>Parameter regions</i>	{r1,r2}	{r3}	{}
<i>Regions of standard args</i>	{r1}	{r3,r4}	{}
<i>Used regions</i>	{r1,r2}	{r3,r4}	{}

Fig. 2. Region information for methods in example

$r$  references an object in region  $r'$ . The points-to relations hold throughout the method (as they are computed in a flow-insensitive manner) and describe the aliasing effects of executing the method. The verifier doesn't assume the correctness of certificates, but rather checks the validity of the provided information.

The regions for each method include, but are not limited to regions that occur in the transformed code of that method. For instance, only `r3` shows up in the code of `reverse`; however, the region information for this method also talks about two other regions `r4` and `r5`. The regions of each method are classified into two categories: 1) *local regions*, regions that are created in the method; and 2) *incoming regions*, regions that are being created by the callers of the current method. For instance, all regions in `make` and `reverse` are incoming regions, and all regions in `test` are local regions. Incoming regions include the following three sub-categories:

- *Parameter regions*. These are the regions passed as parameters in the code. For instance, `r1` and `r2` in `make`, or `r3` in `reverse`. Such regions are passed because the callee will allocate new objects in those regions and need them at the allocation sites. Other regions (e.g. `r4`) need not be passed, even though the callee may access objects in them.
- *Regions of the standard arguments*. These are the regions where the standard method arguments are placed. Standard arguments include the receiver object `this` and the returned value. For short, we will refer to the regions of the standard arguments as argument regions, and distinguish them from the above parameter regions. In our example, the argument regions of `reverse` are `r3` (for the return value) and `r4` (for `this`); region `r4` is an argument region but not a parameter region. Note that the incoming regions are exactly the regions reachable from the argument regions in the points-to graph.
- *Used regions*. These are the regions that the method (and its invoked methods) accesses or places objects in. It is therefore a superset of the parameter regions; it is usually a superset of the argument regions, too, unless standard arguments are not used in the method.

The generated code refers only to local regions and parameter regions. Although the remaining incoming regions don't occur in the program, they are nevertheless needed by the compiler in the program transformation process, and by the verifier in the verification process.

## 4 Extended Region Bytecodes

In our system, each frame contains, besides the standard local variables and operand stack, two additional components to deal with region manipulations:

- *A set of local region variables.* These are identified using region index values, starting from 0. Region variables hold references to region handles.
- *A separate region stack* that is used to pass regions as parameters to calls.

Therefore, we keep region variables and region parameters separate from standard variables and parameters. This allows us to build our extensions in a clean manner, on top of the existing bytecode system, without interfering with the structure of the underlying standard bytecodes. This separation has two advantages. First, we can use the standard verifier to perform the standard safety checks; and use our verifier to check just the region extensions. Second, region-annotated programs can be executed with minimal changes on a standard VM – just by replacing the region versions of allocation instructions with their non-region versions and ignoring the other region instructions.

Our system uses the following 8 region bytecode instructions:

- **create** *ri*: allocates a new region handle and stores it in the local region variable at index *ri*. The variable must not reference a valid region handle before this statement (i.e., creating a region multiple times is not allowed).
- **remove** *ri*: deallocates the region handle that region variable at index *ri* references. The variable must reference a valid local region handle.
- **newreg** *ci ri*, **anewarrayreg** *ci ri*, **newarrayreg** *typ ri*, **multianewarrayreg** *ci dim ri*: allocates an object or array into the region indicated by the region local variable at index *ri*. Here *ci* is the class index, *typ* is a constant indicating a primitive type, and *dim* is the number of array dimensions. We require that the region variable at index *ri* holds a valid region handle when the allocation instruction executes.
- **pushreg** *ri*: pushes on the region stack the reference stored in region variable at index *ri*. The region variable at index *ri* must hold a valid region handle when such an instruction is executed.

## 5 The Verification Process

The verification process consists of a load-time verification phase and a linkage verification phase. The verification of each method requires a region-safety certificate that contains information about the region effects of that method (including its points-to graph, its accessed regions, and other information). This information must be provided along with the bytecodes and must be available during verification.

The load-time verification occurs when new classes are being loaded and consists of a dataflow analysis that checks the body of each method. Verification is performed only intra-procedurally and method calls are ignored. The load-time verification checks, however, the consistency of region effects for overridden methods in the class hierarchy.

The linkage verification checks method calls dynamically, when methods

get invoked. The goal is to ensure the consistency of region effects between callers and callees. The verification occurs only the first time when the method is invoked; afterward, the call site is marked as successfully checked and subsequent invocations at that site will not incur the verification overhead.

### 5.1 Region-Safety Certificates

Our system requires that certificates proving the safety of region manipulations are available for each method, and are provided along with the bytecodes (as attributes in the class files). Such information is automatically generated by our region compiler as part of the transformation process. Let us denote by  $R_m$  the set of all regions of method  $m$ , and by  $I_m$  the incoming regions of  $m$ . The class files will include, for each method  $m$ , the following information:

- The region points-to information  $G_m \subseteq R_m \times F \times R_m$ , where  $F$  is the set of field names. Each element  $(r, f, r') \in G_m$  indicates that, throughout the execution of  $m$ , the field  $f$  of each object in  $r$  references an object in  $r'$ . This information captures the region aliasing effects of method  $m$ .
- For each call site in the method body, the region of the returned value at that call site (only if the returned value is a reference). We represent this information as a map  $c_m : C_m \rightarrow R_m$ , where  $C_m$  is the set of call sites in  $m$ .
- The ordered set  $A_m \subseteq I_m$  of argument regions. These are the regions of the standard arguments of  $m$  (including the receiver and a return parameter).
- The ordered set of incoming regions  $I_m$ .
- The ordered set  $P_m \subseteq I_m$  of parameter regions passed to method  $m$ .
- The set  $U_m \subseteq I_m$  of regions used (accessed) by  $m$ .

The verifier does not take the correctness of the certificates for granted. In particular, it does not rely on the points-to graphs being correct. Instead, the verification fails when the provided information is not correct.

### 5.2 Load-Time Verification

When a new class is being loaded, the verifier performs a dataflow analysis for each method. The verifier will: a) determine regions for local variables and references on the operand stack; b) keep track of live regions; c) check that field accesses always occur in live regions and are consistent with the declared sets of used regions for the enclosing method.

To formalize the analysis, we will assume that each method is represented as a control-flow graph whose nodes are one of the following instructions:

$$\begin{aligned}
 i \in Instr \quad i &::= \text{load } e \mid \text{store } e \mid \text{push } p \mid \text{op} \mid \text{invoke } m \mid \text{return} \mid \\
 &\quad \text{newreg } r \mid \text{pushreg } r \mid \text{create } r \mid \text{remove } r \\
 e \in Expr \quad e &::= x \mid x.f \qquad p \in Prim \quad p ::= n \mid \text{true} \mid \text{false} \mid \text{null}
 \end{aligned}$$

where  $x \in Var$  are local variables,  $f \in F$  are field names,  $r$  are region variables (if  $m$  is the current method, then  $r \in R_m$ ), and  $p$  are primitive values (integer constants  $n$ , booleans, and `null`). Loads and stores move data between the top of the operand stack and local variables or fields. Push instructions push primitive values on stack. Operations `op` pop the first two values at the top of the stack, perform an operation (e.g., arithmetic, logic, or comparison), and place the result back on the operand stack; the operands must be both references or both primitive values, and the result is always a primitive value. Method invocations pop their arguments from the operand stack, pop all the regions from the region stack, and place their result back on the operand stack. Return statements pop the method's result (if any) from the operand stack. A dynamic allocation `newreg`  $r$  creates a new object and places it in region  $r$ . Instruction `pushreg`  $r$  pushes region  $r$  on the region stack. Finally, `create` and `remove` have the behavior discussed in the previous sections.

The analysis models the regions of objects on the operand stack using stacks of regions, defined as follows:

$$S ::= \emptyset \mid S : r \mid S : \top \mid S : \perp$$

where  $\perp$  indicates a null reference,  $\top$  indicates either a non-null primitive value or a reference for which the analysis could not determine a precise region. The region stack is similar, but contains no  $\top$  or  $\perp$  values.

The dataflow information is a tuple  $(S_o, S_r, V, L)$  where  $S_o$  models regions in the operand stack,  $S_r$  models regions on the region stack,  $V : Var \rightarrow R_m \cup \{\top, \perp\}$  maps variables to their regions,  $\top$ , or  $\perp$ ; and  $L \subseteq R_m$  is the set of live regions. We assume that all of the information presented in Section 5.1 is available during the analysis. If the currently analyzed method is  $m$ , the transfer functions for load, store, push, and operations are as follows:

$$\begin{aligned} \llbracket \text{load } x \rrbracket (S_o, S_r, V, L) &= (S_o : V(x), S_r, V, L) \\ \llbracket \text{load } x.f \rrbracket (S_o, S_r, V, L) &= (S_o : v, S_r, V, L), \\ &\text{if } V(x) = r' \in L \wedge r' \in U_m \wedge (v = \top \vee (v = r \wedge (r', f, r) \in G_m)) \\ \llbracket \text{store } x \rrbracket (S_o : v, S_r, V, L) &= (S_o, S_r, V[x \mapsto v], L) \\ \llbracket \text{store } x.f \rrbracket (S_o : v, S_r, V, L) &= (S_o, S_r, V, L), \\ &\text{if } V(x) = r' \in L \wedge r' \in U_m \wedge (v \in \{\top, \perp\} \vee (v = r \wedge (r', f, r) \in G_m)) \\ \llbracket \text{push } p \rrbracket (S_o, S_r, V, L) &= \begin{cases} (S_o : \top, S_r, V, L) & \text{if } p \neq \text{null} \\ (S_o : \perp, S_r, V, L) & \text{if } p = \text{null} \end{cases} \\ \llbracket \text{op} \rrbracket (S_o : v_1 : v_2, S_r, V, L) &= (S_o : \top, S_r, V, L) \end{aligned}$$

If the side-conditions in the transfer functions for field accesses `load`  $x.f$  and `store`  $x.f$  are not met, verification fails. These instructions require that the accessed object is in a precisely known region  $r' \notin \{\perp, \top\}$ , the region is live, is in the set of accessed regions for the method, and that region of the accessed

field is found in the region graph (unless the field has a primitive value). The other four instructions above require no checks and always succeed. In particular, `load  $x$`  and `store  $x$`  are allowed to manipulate dangling references, as long as the program does not subsequently use those references to access their target objects.

When the analysis reaches a call site  $cs$  that invokes a method  $m$  with  $k$  standard parameters and  $j$  region parameters, it just pops these arguments and pushes the region of the return value at this call site, available in  $c_m$ :

$$\llbracket \text{invoke } m \rrbracket (S_o : v_k : \dots : v_1, r_j : \dots : r_1, V, L) = (S_o : c_m(cs), \emptyset, V, L)$$

The verification of the invoked method is postponed until the execution of the program invokes this method. The analysis saves the ordered set  $A_{cs} = \{c_m(cs), v_1, \dots, v_k\}$  of actual argument regions (including the returned value), the ordered set  $P_{cs} = \{r_1, \dots, r_j\}$  of actual parameter regions, and the live set  $L$  at this site. Finally `return` instructions are exit nodes in the flow graph, so they have no transfer function. However, the verifiers checks that the returned value is stored in the return region of  $A_m$ .

The transfer functions for the region instructions are as follows:

$$\llbracket \text{newreg } r \rrbracket (S_o, S_r, V, L) = (S_o : r, S_r, V, L),$$

$$\text{if } r \in L \wedge r \in U_m \wedge (r \in I_m \Rightarrow r \in P_m)$$

$$\llbracket \text{pushreg } r \rrbracket (S_o, S_r, V, L) = (S_o, S_r : r, V, L), \text{ if } (r \in I_m \Rightarrow r \in P_m)$$

$$\llbracket \text{remove } r \rrbracket (S_o, \emptyset, V, L) = (S_o, \emptyset, V, L - \{r\}), \text{ if } r \in L \wedge r \notin I_m$$

$$\llbracket \text{create } r \rrbracket (S_o, \emptyset, V, L) = (S_o \triangleleft r, \emptyset, V \triangleleft r, L \cup \{r\}), \text{ if } r \notin L \wedge r \notin I_m$$

$$\text{Where } (S : r') \triangleleft r = \begin{cases} (S \triangleleft r) : \top & \text{if } \text{reach}(r', r) \\ (S \triangleleft r) : r' & \text{otherwise} \end{cases}$$

$$(V \triangleleft r)(x) = \begin{cases} \top & \text{if } V(x) = r', \text{ and } \text{reach}(r', r) \\ V(x) & \text{otherwise} \end{cases}$$

$$\text{reach}(r', r) \Leftrightarrow r' = r \vee (\exists r'', f. (r', f, r'') \in G_m \wedge \text{reach}(r'', r))$$

A dynamic allocation `newreg` checks that the allocation region is live. It also checks that if the region  $r$  specified in this instruction is not a local region ( $r \in I_m$ ), then it has to be passed as parameter to the current method ( $r \in P_m$ ). A similar check occurs for `pushreg`. However, when passing a region with `pushreg`, the analysis does not check yet whether the pushed region is live; instead, the system defers this check for later, during the linkage verification. The transfer functions for `create` and `remove` forbid consecutive region creation operations, or consecutive remove operations; and restrict region creation and removal to local regions only. Therefore, incoming regions can not be removed in the callees. This choice simplifies verification (it is possible to relax this condition, but then the analysis and the certificates

must be extended with information about the regions that each method may remove). Finally, `create` and `remove` require the region stack  $S_r$  to be empty since region parameters are expected to be pushed right before a method call, without regions being created or removed in between.

The rule for `create r` is more complicated because it must take special care of invalid programs such as the following:

```
create r; x.f = new C in r; remove r;
create r; x.f.g = 1;
```

In this example, there are two dynamic regions with the same name  $r$ . After `remove`, field  $x.f$  holds a dangling reference into the “old” region  $r$ . The second `create` must not transform this into a valid reference. The verifier disallows such cases by setting to  $\top$  each region in  $S_o$  and  $V$  that can reach the newly created region  $r$  through zero or more field accesses. This is done using the filtering operation  $\triangleleft$ .

To complete the dataflow analysis, we define the merge operation that combines dataflow information control-flow join points. The merge of two tuples  $(S_o, S_r, V, L) \sqcup (S'_o, S'_r, V', L')$  is point-wise and yields  $(S_o \sqcup S'_o, S_r \sqcup S'_r, V \sqcup V', L \sqcup L')$ . To join two operand stacks  $S_o \sqcup S'_o$ , we require that the stacks have the same size. Then, the join is component-wise:  $(S : v) \sqcup (S' : v') = (S \sqcup S') : (v \sqcup v')$  and  $\emptyset \sqcup \emptyset = \emptyset$ . Here, the merge of values corresponds to a flat lattice for regions:  $r \sqcup r = r$ ,  $r \sqcup r' = \top$  if  $r \neq r'$ ,  $\perp \sqcup v = v \sqcup \perp = v$ , and  $\top \sqcup v = v \sqcup \top = \top$ . Note that this lattice has height 2, as opposed to the lattice for standard bytecode verification which can have arbitrary height, depending on the class hierarchy. Therefore, our dataflow verification will converge faster. For region stacks, we require that  $S_r = S'_r = \emptyset$ , that is, region stacks must be empty at join points.

The merge of variable mappings is also point-wise:  $V \sqcup V' = V''$ , where  $V''(x) = V(x) \sqcup V'(x)$ . Finally, we require that live sets are equal at join points:  $L' = L$ ; otherwise, the verifier yields an error.

The analysis initializes the dataflow information at the beginning of the method to  $(\emptyset, \emptyset, V_m, U_m)$ , where  $V_m$  maps each formal reference parameter of  $m$  to its corresponding region in  $A_m$ , and maps all other variables to  $\top$ . Finally, all regions used by the current method ( $U_m$ ) must be live at the method entry.

Besides the dataflow analysis presented in this section, the load-time verification also checks the region consistency of virtual methods in the class hierarchy. Because this verification uses techniques similar to those employed during linkage verification, we postpone its description until Section 5.4.

### 5.3 Linkage Verification

The goal of linkage verification is to check that the invoked methods do not violate the assumptions made about their region effects in their callers. The system checks methods lazily: each call site is verified only when the execution

reaches it the first time; afterward, the call site is marked as safe and subsequent calls at the same site do not incur the overhead of linkage verification.

The key part in the verification of a method invocation is to establish a correspondence between the incoming regions of the callee and the regions of the caller. Consider a call site in method  $m$  that invokes method  $n$ . The verifier then constructs a mapping  $\alpha$  from  $n$ 's regions to  $m$ 's regions using a worklist algorithm. The algorithm starts with the argument regions  $A_n = \{r_0, \dots, r_k\}$  of the callee and the actual regions  $A_{cs} = \{v_0, \dots, v_k\}$  recorded during the load-time verification of the caller  $m$ . The sizes of  $A_n$  and  $A_{cs}$  must match, otherwise verification fails. Also, for each reference parameter of  $n$ , the corresponding value in  $A_{cs}$  must be either a region or  $\perp$ ; if it is  $\top$ , verification fails. For simplicity, assume that  $v_0, \dots, v_k$  are all regions, and they include the region for the returned value. The algorithm is as follows:

```

MAPPING ALGORITHM()
1  let  $\{r_0, \dots, r_k\} = A_n$ 
2  let  $\{v_0, \dots, v_k\} = A_{cs}$ 
3  for  $i = 0$  to  $k$  do  $\alpha(r_i) = v_i$ 
4   $W = A_n$ 
5  while ( $W$  is not empty)
6      remove  $r_{1m}$  from  $W$ 
7      for each field  $f$  such that  $(r_{1m}, f, r_{2m}) \in G_m$ 
8          if  $\alpha(r_{2m})$  is not defined
9              then if  $(\alpha(r_{1m}), f, r_{2n}) \in G_n$ 
10                 then  $\alpha(r_{2m}) = r_{2n}$ 
11                     $W = W \cup \{r_{2m}\}$ 
12                 else verifier error
13             else if  $(\alpha(r_{1m}), f, \alpha(r_{2m})) \notin G_n$ 
14                 then verifier error
    
```

The algorithm initializes the mapping  $\alpha$  by matching the corresponding regions in  $A_n$  and  $A_{cs}$ ; then, it uses a worklist approach to traverse the region graphs  $G_n$  and  $G_m$  starting from those sets, building the map, and checking that the callee information  $G_n$  is conservatively “embedded” in  $G_m$ . Once the mapping is successfully constructed,  $(r, f, r') \in G_n$  implies that  $(\alpha(r), f, \alpha(r')) \in G_m$ , which describes the notion of embedding.

In the example from Section 3, the call site mapping for the invocation of `reverse` in method `test` is:  $\alpha(\mathbf{r3}) = \mathbf{r8}$ ,  $\alpha(\mathbf{r4}) = \mathbf{r6}$ , and  $\alpha(\mathbf{r5}) = \mathbf{r7}$ .

Once the mapping  $\alpha$  has been constructed, the verifier checks the following:

- (P1) The incoming regions used by the callee correspond to live regions at the call site:  $\{\alpha(r) \mid r \in U_n\} \subseteq L_{cs}$ ;
- (P2) The incoming regions used by the callee correspond to used regions in the caller:  $\{\alpha(r) \mid r \in U_n\} \cap I_m \subseteq U_m$ ;

- (P3) The formal parameter regions of the caller  $P_n = \{p_1, \dots, p_j\}$  match the actual region passed at the call site  $P_{cs} = \{r_1, \dots, r_j\}$ :  $\alpha(p_i) = r_i$ , for  $i = 1..j$ .

If all of these checks have been successfully verified, it is safe to invoke the method and execute the call.

#### 5.4 Method Overriding Verification

The linkage verification described above guarantees memory safety only if method calls are not dynamically dispatched. To ensure the safe execution in the presence of virtual method calls, the verifier must ensure that the region effects of each method subsumes the region effects of all the methods that override it.

Such method overriding checks occur at load-time and work as follows. Whenever a new method  $m$  is loaded, the verifier looks for the method  $m'$  that  $m$  immediately overrides in the class hierarchy. If such a method exists, the verifier checks that the region information of  $m'$  approximates that of  $m$ . The verification essentially consists a fabricating a dummy call from  $m'$  to  $m$  that passes the formal parameters of  $m'$  as actual parameters at the call site, and analyzing this dummy call in a similar manner to what has been described in Section 5.3. The algorithm will construct a mapping between the regions of  $m$  and the regions of the overridden method  $m'$ . The actual argument regions and actual parameter regions at the dummy call site are the exactly the formal argument and parameter regions of the caller method  $m'$ . After the region mapping has been constructed, the verifier checks properties (P2) and (P3) hold for the dummy call site (property (P1) needs not be checked).

#### 5.5 Verification of Other Constructs

A few language constructs require special treatment. Because it is difficult to identify the lifetimes of objects stored in static fields, our region compiler places such objects in an immortal region whose lifetime spans over the lifetime of the entire program. The verifier works accordingly: the immortal region is a special region (with index 0) in the frame of each method; and the region loaded from, or stored into a static field must be the immortal region. Exceptions also require the thrown object to be in the immortal region.

For multithreading, the verifier requires that all of the regions used by the child thread must be passed as parameters to the thread start method  $m$ , that is,  $U_m = P_m$ . Furthermore, the verifier treats all of these parameter regions as local regions in the thread start method. These regions are shared regions and the run-time system will ensure that only the last thread that attempts to remove such regions will actually remove them, as mentioned in Section 2.

Program	Original BC	Region BC	Reg. BC +Cert	Percent Increase	Verified	
					Methods	Calls
bh	30133	31677	36573	21%	280	535
bisort	8080	8491	9556	18%	236	403
em3d	11834	12784	14854	26%	245	437
health	16143	17214	20464	27%	250	444
mst	11667	12090	14738	26%	256	433
perimeter	15663	15943	18161	16%	269	432
power	24274	24767	26598	9%	243	417
treeadd	5206	5438	5912	14%	229	391
tsp	10972	11319	12378	13%	239	433
voronoi	25601	26696	30786	20%	270	558

Fig. 3. Program sizes in bytes, certification overhead; methods and calls verified.

## 6 Experiments

We have implemented the region certificate generation in the Jreg region compiler, built in the Soot infrastructure [21]. The compiler produces region certificates along with the region-annotated bytecodes using the results of our compiler analyses; both the analyses and the certification generation takes place in a three-address intermediate representation. This is possible because all of the region information, including the points-to information, is independent of how the intermediate code gets lowered to bytecodes.

The region compiler generates the region certificates as method attributes in the class files. Most of the attributes are independent of the code structure; only the information about returned regions for call sites depends on the program counters of the calls. This particular attribute is encoded in the same fashion as the line number attributes.

We have also implemented the verifier in the interpreter of the Kaffe virtual machine [22]. We perform the load-time verification right after the VM performs the standard bytecode verification. Linkage verification is done lazily, at runtime, when a call is first encountered. For this, we included a flag to indicate whether a call has already been verified.

We evaluated the verifier using the Olden benchmarks for Java [5]. The left part of Figure 3 presents bytecode size statistics for these programs: the first column shows the original size of the application; the second column shows the size of bytecodes with region instructions, but with no region certificates; the third column shows the size of the bytecodes including the certificates; the fourth column shows the percentage increase in size of programs with full annotations over the original ones.

These numbers show that, the average size increase for this set of programs amounts to an acceptable 19%. As expected, region certificates are responsible for most of the size increase of region bytecodes. Inspecting the certificate sizes, we have determined that, in average, 35% of the certificate size is due to points-to graphs; more interestingly, 22% of the certificate size is

required to store the strings that represent the fields in the points-to relations. We point out that we have not attempted to optimize these sizes and better representations and compression techniques may yield smaller certificates.

We have also collected statistics about the actual verification process for these benchmarks. The left part of Figure 3 shows the number of methods verified during class loading and the number of call sites checked during program execution. These numbers include the verification of library methods.

Finally, the execution times show that the verification run-time overhead in the Kaffe interpreter is insignificant: with program running times ranging between 23 and 672 seconds, the verification overhead was in average 0.08%, with a maximum of 0.3%. These results confirm the expectation that safety checking becomes cheap once the key information in certificates is available to the verifier. We envision that the region verification overhead remains small even for a just-in-time compilation system.

### 6.1 Bug detection and fixes

The verifier has already been successful at finding several bugs in our region compiler. A first error that the verifier has detected was the presence of inconsistent live sets of regions at merge points. The rejected code revealed that, in certain situations, the compiler was not placing the appropriate `remove` instruction for conditional statements with empty false branches. A second problem that the verifier has identified was that all strings, including constants, were treated as heap objects. Of course, constant strings should not be assigned regions, but rather placed in the immortal region. A third bug detected was that our compiler was generating inconsistent region parameter signatures in the presence of inheritance and method overriding. We have successfully fixed all these bugs.

The verifier proved to be helpful in two respects. First, it caught errors that would silently pass otherwise because they would not corrupt the memory. Second, it gave good indications about the cause of errors and it made it easier to fix them.

## 7 Related Work

There has been a large body of research in the area of language support for region-based memory management. Tofte and Talpin [20] propose a region inference algorithm for a simply typed lambda calculus and use this technique to provide automatic region support for ML Programs [2,19]. Regions in their system are lexically scoped, which imposes a stack discipline on region lifetimes. Aiken, Fahndrich, and Levien [1] build on this approach, but lexically decouple region allocation from region deallocation.

The RC system [10] and Cyclone [12] propose language support for regions in C programs: they provide a combination of static checks and run-time sup-

port to ensure that region accesses are safe. Several other systems extend Java with region support. Christiansen and Velschow propose RegJava [8], which extends Java with region-annotated class types. Boyapati et. al. present a system that combines ownership and region types in Java programs [4]. Our own recent work [6] and that of Chin et. al. [7] present region inference algorithms automatically transform input Java programs into output program with region support [7]. Finally, the real-time specification for Java (RTSJ) [3] provides an API to support scoped memory regions and immortal memory; however, it requires run-time checks to ensure memory safety (e.g., the absence of dangling references). In contrast, our approach provides lightweight region support at the bytecode level and the verifier presented in this paper eliminates the need of run-time checks.

Most of the region systems for C, Java, and ML except [1,13,10] use regions that are lexically scoped, use region type systems, and generate well-typed output programs that can be type-checked to ensure correctness. Systems like [1] do not provide a way to check that the result of the transformation is correct. And systems like RC [10] use run-time techniques to ensure memory safety. To the best of our knowledge, the work in this paper is the first to address the verification of region-safety at the bytecode level.

In the area of bytecode verification, various analysis algorithms have been proposed, as presented in a recent survey [15]. The basic Java bytecode verification algorithm is due to Gosling and Yellin at Sun [16,11] and is based on dataflow analysis. Our region verification is orthogonal to the standard JVM verification, as it just checks the region constructs, but otherwise relies on the standard verifier to carry out all of the other checks.

The standard bytecode verification is usually complicated by object initialization and subroutine verification. Object initialization requires a form of must-alias analysis to determine that objects are correctly initialized. Fairly simple ways of reasoning about aliases have been proposed, along with certain restrictions [9], to ensure proper initialization. Another issue is that of subroutines, where context-sensitive approaches are required to allow a larger class of valid programs to be successfully verified. Both of these approaches are orthogonal to region verification and can be handled by the standard verifier (in fact, our system uses Soot [21], which eliminates subroutines via inlining<sup>2</sup>).

More relevant to this work is the general proof-carrying code approach [17], where the executable code is accompanied by a correctness proof. For bytecode verification, this idea has been used to include certificates in the bytecodes to speed up verification [18,14]. More precisely, certificates provide type invariants at control-flow join points, thus replacing flow-based type inference with a single-pass type-checking. Our system also follows the proof-carrying code paradigm: the region information provided with the bytecodes is a proof of memory safety for the region manipulations.

---

<sup>2</sup> This is possible since subroutines are not allowed to be recursive in the JVM[16]§4.8.2

## 8 Conclusions

We have presented algorithms and techniques for the verification of region-annotated bytecodes that support non-lexically scoped regions and dangling references. The successful verification ensures region-safety: the program execution never deallocates regions while their objects are still in use, and it never follows dangling pointers. The verifier requires region-safety certificates to be provided with the bytecodes; it then performs the verification process through a combination of load-time dataflow verification of method bodies and lazy linkage verification of method calls. We believe that this work makes region-annotated bytecodes a more attractive alternative for region-based memory management in Java.

## References

- [1] A. Aiken, M. Fahndrich, and R. Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *Proceedings of the SIGPLAN '95 Conference on Program Language Design and Implementation*, La Jolla, CA, June 1995.
- [2] L. Birkedal, M. Tofte, and M. Vejlstrup. From region inference to von Neumann machines via region representation inference. In *Proceedings of the 23rd Annual ACM Symposium on the Principles of Programming Languages*, St. Petersburg Beach, FL, January 1996.
- [3] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, D. Hardin, and M. Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, Reading, Mass., 2000. <http://www.rtj.org/>.
- [4] C. Boyapati, A. Salcianu, W. Beebe, and M. Rinard. Ownership types for safe region-based memory management in real-time Java. In *Proceedings of the SIGPLAN '03 Conference on Program Language Design and Implementation*, San Diego, CA, June 2003.
- [5] Brendon Cahoon and K. S. McKinley. Data flow analysis for software prefetching linked data structures in Java. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT '01)*, Barcelona Spain, September 2001.
- [6] S. Chereem and R. Rugina. Region analysis and transformation for Java programs. In *Proceedings of the International Symposium on Memory Management*, Vancouver, Canada, October 2004.
- [7] W. Chin, F. Craciun, S. Qin, and M. Rinard. Region inference for an object-oriented language. In *Proceedings of the SIGPLAN '04 Conference on Program Language Design and Implementation*, Washington, DC, June 2004.
- [8] M. Christiansen and P. Velschow. Region-based memory management in Java. Master's thesis, DIKU, University of Copenhagen, May 1998.

- [9] S. Freund and J. Mitchell. A type system for object initialization in the Java bytecode language. *ACM Transactions on Programming Languages and Systems*, 21(6):1196–1250, 1999.
- [10] D. Gay and A. Aiken. Language support for regions. In *Proceedings of the SIGPLAN '01 Conference on Program Language Design and Implementation*, Snowbird, UT, June 2001.
- [11] J. Gosling. Java intermediate bytecodes. In *ACM SIGPLAN Workshop on Intermediate Representations*, San Francisco, CA, January 1995.
- [12] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *Proceedings of the SIGPLAN '02 Conference on Program Language Design and Implementation*, Berlin, Germany, June 2002.
- [13] F. Henglein, H. Makhholm, and H. Niss. A direct approach to control-flow sensitive region-based memory management. In *Proceedings of the 3rd international ACM SIGPLAN conference on Principles and Practice of Declarative Programming*, Florence, Italy, September 2001.
- [14] G. Klein and T. Nipkow. Verified lightweight bytecode verification. *Concurrency: Practice & Experience*, 13(13):1133–1151, 2001.
- [15] X. Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, 30(3–4):235–269, 2003.
- [16] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, second edition edition, 1999.
- [17] G. Necula. Proof-carrying code. In *Proceedings of the 24th Annual ACM Symposium on the Principles of Programming Languages*, pages 106–119, Paris, France, January 1997.
- [18] E. Rose and K. Rose. Lightweight bytecode verification. In *OOPSLA Workshop on Formal Underpinnings of Java*, Vancouver, Canada, October 1998.
- [19] M. Tofte and L. Birkedal. A region inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4):724–767, July 1998.
- [20] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value lambda-calculus using a stack of regions. In *Proceedings of the 21st Annual ACM Symposium on the Principles of Programming Languages*, Portland, OR, January 1994.
- [21] R. Vallye-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *CASCON '99*, Toronto, Canada, November 1999.
- [22] Tim Wilkinson. Kaffe – a free Java virtual machine. <http://www.kaffe.org>.