Establishing Local Temporal Heap Safety Properties with Applications to Compile-Time Memory Management

Ran Shaham[1,2], Eran Yahav[1], Elliot K. Kolodner[2], and Mooly Sagiv[1]

[1] School of Computer Science, Tel-Aviv University, Tel-Aviv, Israel.
{rans,yahave,sagiv}@math.tau.ac.il
[2] IBM Haifa Research Laboratory, Haifa, Israel.
kolodner@il.ibm.com

Abstract. We present a framework for statically reasoning about temporal heap safety properties. We focus on *local temporal heap safety properties*, in which the verification process may be performed for a program object independently of other program objects. We apply our framework to produce new conservative static algorithms for compile-time memory management, which prove for certain program points that a memory object or a heap reference will not be needed further. These algorithms can be used for reducing space consumption of Java programs. We have implemented a prototype of our framework, and used it to verify compile-time memory management properties for several small, but interesting example programs, including JavaCard programs.

1 Introduction

This work is motivated by the need to reduce space consumption, for example for memory-constrained applications in a JavaCard environment. Static analysis can be used to reduce space consumption by identifying source locations at which a heap-allocated object is no longer needed by the program. Once such source locations are identified, the program may be transformed to directly free unneeded objects, or aid a runtime garbage collector collect unneeded objects earlier during the run.

The problem of statically identifying source locations at which a heap-allocated object is no longer needed can be formulated as a local temporal heap safety property — a temporal safety property specified for each heap-allocated object independently of other objects.

The contributions of this paper can be summarized as follows.

1. We present a framework for verifying local temporal heap safety properties of Java programs.
2. Using this framework, we formulate two important compile-time memory management properties that identify when a heap-allocated object or heap reference is no longer needed, allowing space savings in Java programs.
3. We have implemented a prototype of our framework, and used it as a proof of concept to verify compile-time memory management properties for several small but interesting example programs, including JavaCard programs.

1.1 Local Temporal Heap Safety Properties

This paper develops a framework for automatically verifying *local temporal heap safety properties*, i.e., temporal safety properties that could be specified for a program object independently of other program objects. We assume that a safety property is specified

using a *heap safety automaton* (HSA), which is a deterministic finite state automaton. The HSA defines the valid sequences of events that could occur for a single program object.

It is important to note that our framework implicitly allows infinite state machines, since the number of objects is unbounded. Furthermore, during the analysis an event is triggered for a state machine associated with an object. Thus, precise information on heap paths to disambiguate program objects is crucial for the precise association of an event and its corresponding program object's state machine.

In this paper, we develop static analysis algorithms that verify that on all execution paths, all objects are in an HSA accepting state. In particular, we show how the framework is used to verify properties that identify when a heap-allocated object or heap reference is no longer needed by the program. This information could be used by an optimizing compiler or communicated to the runtime garbage collector to reduce the space consumption of an application. Our techniques could also be used for languages like C to find a misplaced call to `free` that prematurely deallocates an object.

1.2 Compile-Time Memory Management Properties

Runtime garbage collection (GC) algorithms are implemented in Java and C# environments. However, GC does not (and in general cannot) collect all the garbage that a program produces. Typically, GC collects objects that are no longer reachable from a set of *root* references. However, there are some objects that the program never accesses again and therefore not needed further, even though they are reachable. In previous work [28,31] we showed that on average 39% of the space could be saved by freeing reachable unneeded objects. Moreover, in some applications, such as those for JavaCard, GC is avoided by employing static object pooling, which leads to non-modular, limited, and error-prone programs.

Existing compile-time techniques produce limited saving. For example, [1] produces a limited savings of a few percent due to the fact that its static algorithm ignores references from the heap. Indeed, our dynamic experiments indicate that the vast majority of savings require analyzing the heap.

In this paper, we develop two new static algorithms for statically detecting and deallocating garbage objects:

free analysis Statically identify source locations at which it is safe to insert a free statement in order to deallocate a garbage element.

assign-null analysis Statically identify source locations at which it is safe to assign null to heap references that are not used further in the run.

The assign-null analysis leads to space saving by allowing the GC to collect more space. In [31] we conduct dynamic measurements that show that assigning null to heap references immediately after their last use has an average space-saving potential of 15% beyond existing GCs. Free analysis could be used with runtime GC in standard Java environments and without GC for JavaCard.

Both of these algorithms handle heap references and destructive updates. They employ both forward (history) and backward (future) information on the behavior of the program. This allows us to free more objects than reachability based compile-time garbage collection mechanisms (e.g., [17]), which only consider the history.

1.3 A Motivating Example

Fig. 1 shows a program that creates a singly-linked list and then traverses it. We would like to verify that for this program a `free y` statement can be added immediately after line 10. This is possible because once a list element is traversed, it cannot be accessed along any execution path starting after line 10. It is interesting to note that even in this simple example, standard compile-time garbage collection techniques (e.g., [17]) will not issue such a free statement, since the element referenced by `y` is reachable via a heap path starting from `x`. Furthermore, integrating limited information on the future of the computation such as liveness of local reference variables (e.g., [1]) is insufficient for issuing such free statement. Nevertheless, our analysis is able to verify that the list element referenced by `y` is no longer needed, by investigating all execution paths starting at line 10.

```
   class L { // L is a singly linked list
     public L   n;   // next field
     public int val; // data field
   }
   class Main { // Creation and traversal of a singly-linked list
    public static void main(String args[]) {
      L x, y, t;
[1]  x = null;
[2]  while (...) { // list creation
[3]    y = new L();
[4]    y.val = ...;
[5]    y.n = x;
[6]    x = y;
     }
[7]  y = x;
[8]  while (y != null) {         // list traversal
[9]    System.out.print(y.val);
[10]   t = y.n;
[11]   y = t;
     }
    }
   }
```

Fig. 1. A program for creating and traversing a singly linked list.

In order to prove that a free statement can be added after line 10, we have to verify that all program objects referenced by y at line 10 are no longer needed on execution paths starting at this line. More specifically, for every execution path and every object o, we have to verify that from line 10 there is no use of a reference to o. In the sequel, we show how to formulate this property as a heap safety property and how our framework is used to successfully verify it.

1.4 A Framework for Verifying Heap Safety Properties

Our framework is conservative, i.e., if a heap safety property is verified, it is never violated on any execution path of the program. As usual for a conservative framework, we might fail to verify a safety property which holds on all execution paths of the program.

Assuming the safety property is described by an HSA, we instrument the program semantics to record the automaton state for every program object. First-order logical structures are used to represent a global state of the program. We augment this repre-

sentation to incorporate information about the automaton state of every heap-allocated object.

Our abstract domain uses first-order 3-valued logical structures to represent an abstract global state of the program, which represent several (possibly an infinite number of) concrete logical structures [26]. We use *canonic abstraction* that maps concrete program objects (i.e., individuals in a logical structure) to abstract program objects based on the properties associated with a program object. In particular, the abstraction is refined by the automaton state associated with every program object.

For the purpose of our analyses one needs to: (i) consider information on the history of the computation, to approximate the heap paths, and (ii) consider information on the future of the computation, to approximate the future use of references. Our approach here uses a forward analysis, where the automaton maintains the temporal information needed to reason about the future of the computation.

In principle we could have used a forward analysis identifying heap-paths integrated into a backward analysis identifying future uses of heap references [30]. However, we find the cost of merging forward and backward information too expensive for a heap analysis as precise as ours.

1.5 Outline

The rest of this paper is organized as follows. In Section 2, we describe heap safety properties in general, and a compile-time memory management property of interest — the free property. Then, in Section 3, we give our instrumented concrete semantics which maintains an automaton state for every program object. Section 4 describes our property-guided abstraction and provides an abstract semantics. In Section 5, we describe an additional property of interest — the assign-null property, and discuss efficient verification of multiple properties. Section 6 describes our implementation and empirical results. Related work is discussed in Section 7.

2 Specifying Compile-Time Memory Management Properties via Heap Safety Properties

In this section, we introduce heap safety properties in general, and a specific heap safety property that allows us to identify source locations at which heap-allocated objects may be safely freed.

Informally, a heap safety property may be specified via a heap safety automaton (HSA), which is a deterministic finite state automaton that defines the valid sequences of events for a single object in the program. An HSA defines a prefix-closed language, i.e., every prefix of a valid sequence of events is also valid. This is formally defined by the following definition.

Definition 1 (Heap Safety Automaton (HSA)).
A heap safety automaton $A = \langle \Sigma, Q, \delta, \text{init}, F \rangle$ *is a deterministic finite state automaton, where* Σ *is the automaton alphabet which consists of observable events,* Q *is the set of automaton states,* $\delta : Q \times \Sigma \rightarrow Q$ *is the deterministic transition function mapping a state and an event to a single successor state,* $\text{init} \in Q$ *is the* initial state, $\text{err} \in Q$ *is a distinguished* violation state *(the sink state), for which for all* $a \in \Sigma$, $\delta(err, a) = err$, *and* $F = Q \setminus \{\text{err}\}$ *is the set of accepting states.*

In our framework, we associate an HSA state with every object in the program, and verify that on all program execution paths, all objects are in an accepting state. The HSA is used to define an instrumented semantics, which maintains the state of the automaton for each object. The automaton state is *independently* maintained for every program object. However, the same automaton is used for all program objects.

When an object o is allocated, it is assigned the initial automaton state. The state of an object o is then updated by automaton transitions corresponding to events associated with o, triggered by program statements.

We now formulate the free property, which allows us to issue a free statement to reclaim objects unneeded further in the run. In the sequel, we make a simplifying assumption and focus on verification of the property for a single program point. In Section 5.2 we discuss a technique for efficient verification for a set of program points.

Definition 2 (Free Property $\langle pt, \mathtt{x} \rangle$). *Given a program point pt and a program variable* `x`*, a free property $\langle pt, \mathtt{x} \rangle$ states that the references to an object referenced by* `x` *at pt are not used further on execution paths starting at pt. Therefore, it is safe to issue a* `free x` *statement immediately after pt.*

The free property allows us to free an object that is not needed further in the run. Interestingly, such an object can still be reachable from a program variable through a heap path. For expository purposes, we only present the free property for an object referenced by a program variable. However, this free property can easily handle the free for an object referenced through an arbitrary reference expression `exp`, by introducing a new program variable `z`, assigned with `exp` just after pt, and verifying that `free z` may be issued just after the statement `z = exp`.

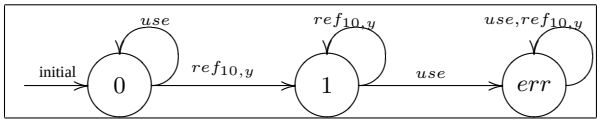

Fig. 2. A heap safety automaton $A^{free}_{10,y}$ for free `y` at line 10.

Example 1. Consider the example program of Fig. 1. We would like to verify that a `free y` statement can be added immediately after line 10, i.e., a list element can be freed as soon as it has been traversed in the loop. The HSA $A^{free}_{10,y}$ shown in Fig. 2 represents the free property $\langle 10, y \rangle$. States 0 and 1 are accepting, while the state labelled *err* is the violation state. The alphabet of this automaton consists of two events associated with a program object o: (i) *use*, which corresponds to a use of a reference to o, and (ii) $ref_{10,y}$, which is triggered when program execution is immediately after execution of the statement at line 10 and `y` references o.

The HSA is in an accepting state along an execution path leading to the program exit iff o can be freed in the program after line 10. Thus, when on all execution paths, for all program objects o, only accepting states are associated with o, we conclude that `free y` can be added immediately after line 10.

First, when an object is allocated, it is assigned the initial state of $A^{free}_{10,y}$ (state 0). Then, a use of a reference to an object o (a *use* event) does not change the state of $A^{free}_{10,y}$ for o (a self-loop on state 0). When the program is immediately after line

10 and y references an object o ($ref_{10,y}$ event), o's automaton state is set to 1. If a reference to o is used further, and o's automaton state is 1 the automaton state for o reaches the violation state of the automaton. In that case the property is violated, and it is not possible to add a `free y` statement immediately after line 10 since it will free an object that is needed later in the program. However, in the program of Fig. 1, references to objects referenced by `y` at line 10 are not used further, hence the property is not violated, and it is safe to add a `free y` statement at this program point. Indeed, in Section 4 we show how the *free* $\langle 10, y\rangle$ property is verified.

An arbitrary free property is formulated as a heap safety property using an HSA similar to the one shown in Fig. 2 where the program point and reference expression are set accordingly.

It should be noted that in Java, free statements are not supported. Therefore, we assume that equivalent free annotations are issued, and could be exploited by the run-time environment. For example, a Java Virtual Machine (JVM) may include an internal free function, and the Just-In-Time (JIT) compiler (which is a run-time compiler included in the JVM) computes where calls to the function can be added.

3 Instrumented Concrete Semantics

We define an instrumented concrete semantics that maintains an automaton state for each heap-allocated object. In Section 3.1, we use first-order logical structures to represent a global state of the program and augment this representation to incorporate information about the automaton state of every heap-allocated object. Then in Section 3.2, we describe an operational semantics manipulating instrumented configurations.

3.1 Representing Program Configurations using First-Order Logical Structures

The global state of the program can be naturally expressed as a first-order logical structure in which each individual corresponds to a heap-allocated object and predicates of the structure correspond to properties of heap-allocated objects. In the rest of this paper, we work with a fixed set of predicates denoted by P.

Definition 3 (Program Configuration). *A* program configuration *is a* 2*-valued first-order logical structure* $C^\natural = \langle U^\natural, \iota^\natural\rangle$ *where:*

- $U^\natural$ *is the universe of the* 2*-valued structure. Each individual in* $U^\natural$ *represents an allocated heap object.*
- $\iota^\natural$ *is the interpretation function mapping predicates to their truth-value in the structure, i.e., for every predicate* $p \in P$ *of arity* k, $\iota^\natural(p)\colon {U^\natural}^k \to \{0, 1\}$.

Predicates	**Intended Meaning**
$at[pt]()$	program execution is immediately after program point pt
$x(o)$	program variable x references the object o
$f(o_1, o_2)$	field f of the object o_1 points to the object o_2
$s[q](o)$	the current state of o's automaton is q

Table 1. Predicates for partial Java semantics.

We use the predicates of Table 1 to record information used by the properties discussed in this paper. The nullary predicate $at[pt]()$ records the program location in a configuration and holds in configurations in which the program is immediately after line pt. The unary predicate $x(o)$ records the value of a reference variable `x` and holds for the individual referenced by `x`. The binary predicate $f(o_1, o_2)$ records the value of a field reference, and holds when the field `f` of o_1 points to the object o_2.

Predicates of the form $s[q](o)$ (referred to as *automaton state predicates*) maintain temporal information by maintaining the automaton state for each object. Such predicates record history information that is used to refine the abstraction. The abstraction is refined further by predicates that record spatial information, such as *reachability* and *sharing* (referred to as *instrumentation predicates* in [26]).

In this paper, program configurations are depicted as directed graphs. Each individual of the universe is displayed as a node. A unary predicate $p(o)$ which holds for an individual (node) u is drawn inside the node u. Predicates of the form $x(o)$ are shown as an edge from the predicate symbol to the node in which it holds since they can only hold for a single individual. The name of a node is written inside the node using an *italic* face. Node names are only used for ease of presentation and do not affect the analysis. A binary predicate $p(u_1, u_2)$ which evaluates to 1 is drawn as directed edge from u_1 to u_2 labelled with the predicate symbol. Finally, a nullary predicate $p()$ is drawn inside a box.

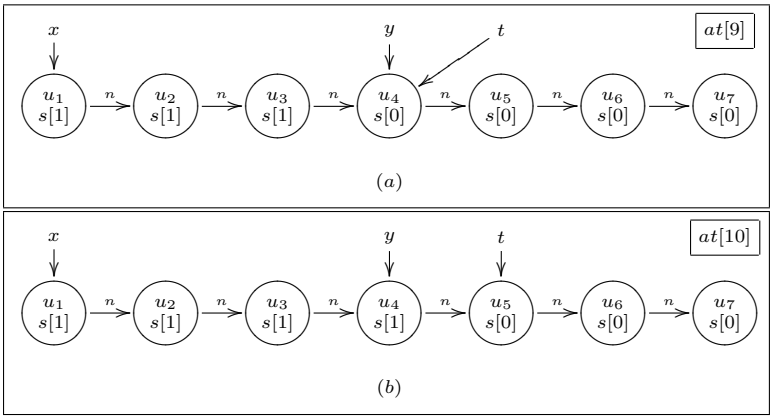

Fig. 3. Concrete program configurations (a) before — and (b) immediately after execution of `t = y.n` at line 10.

Example 2. The configuration shown in Fig. 3(a) corresponds to a global state of the program in which execution is immediately after line 9. In this configuration, a singly-linked list of 7 elements has been traversed up to the 4-th element (labelled u_4) by the reference variable y, and the reference variable t still points to the same element as y. This is shown in the configuration by the fact that both predicates $y(o)$ and $t(o)$ hold for the individual u_4. Directed edges labelled by n correspond to values of the n field. The nullary predicate $at[9]()$ shown in a box in the upper-right corner of the figure records the fact that the program is immediately after line 9. The predicates $s[0](o)$ and $s[1](o)$ record which objects are in state 0 of the automaton and which are in state 1. For

example, the individual u_3 is in automaton state 1 and the individual u_4 is in automaton state 0.

Maintaining Individual Automaton State Given a heap safety property represented as an HSA $A = \langle \Sigma, Q, \delta, \mathit{init}, F \rangle$, we define the unary predicates $\{s[q](o) : q \in Q\}$ to track the state of the automaton for every heap-allocated object. In Section 3.2 we describe how these predicates are updated.

3.2 Operational Semantics

Program statements are modelled by generating the logical structure representing the program state after execution of the statement. In [26] it was shown that first-order logical formulae can be used to formally define the effect of every statement. In particular, first-order logical formulae are used to model the change of the automaton state of every affected individual, reflecting transition-updates in an ordered sequential manner.

In general, the operational semantics associates a program statement with a set of HSA events that update the automaton state of program objects. The translation from the set of HSA events to first-order logical formulae reflecting the change of the automaton state of every affected individual is automatic. We now show how program statements are associated with $A^{free}_{pt,x}$ events. For expository purposes, and without loss of generality, we assume the program is normalized to a 3-address form. In particular, a program statement may manipulate reference expressions of the form `x` or `x.f`.

Object Allocation For a program statement `x = new C()` for allocating an object, a new object o_{new} is allocated, which is assigned the initial state of the HSA, i.e., we set the predicate $s[init](o_{new})$ to 1.

Example 3. Consider the HSA $A^{free}_{10,y}$ of Example 1. For this HSA we define a set of predicates $\{s[0](o), s[1](o), s[err](o)\}$ to record the state of the HSA individually for every heap-allocated object. Initially, when an object o is allocated at line 3 of the example program, we set $s[0](o)$ to 1, and other state predicates of o to 0.

Use Events Table 2 shows the use events fired by each kind of a program statement, where (i) a use of `x` in a program statement updates the automaton state of the object referenced by `x` with a *use* event, and (ii) a use of the field `f` of the object referenced by `x` in a program statement updates the automaton state of the object referenced by `x.f` with a *use* event. For example, the statement `x = y.f` triggers use events for y and $y.f$, which update the automaton state of the object referenced by `y` with a *use* event, and update the automaton state of the object referenced by `y.f` with a *use* event. The order in which use events are triggered does not matter. However, in general, the order should be consistent with the HSA.

$ref_{pt,x}$ Events For a free property $\langle pt, \mathtt{x} \rangle$, the corresponding automaton $A^{free}_{pt,x}$ employs $ref_{pt,x}$ events in addition to *use* events. A $ref_{pt,x}$ event is triggered to update the automaton state of the object referenced by `x` when the current program point is pt. This event is triggered only after the *use* events corresponding to the program statement at pt are triggered.

statement	use events are triggered for an object referenced by
x = y	y
x = y.f	$y, y.f$
x.f = null	x
x.f = y	x, y
x binop y	x, y

Table 2. Use events triggered by program statements.

Example 4. Fig. 3 shows the effect of the `t = y.n` statement at line 10, where the statement is applied to the configuration labelled by (a). First, this statement updates the predicate $t(o)$ to reflect the assignment by setting it to 1 for u_5, and setting it to 0 for u_4. In addition, it updates the program point by setting $at[10]()$ to 1 and $at[9]()$ to 0. Then, 2 *use* events followed by a $ref_{10,y}$ event are triggered: (i) use of the object referenced by `y`, causing the object u_4 to remain at automaton state 0, i.e., $s[0](u_4)$ remains 1; (ii) use of the object referenced by `y.n`, causing the object u_5 to remain at automaton state 0, i.e., $s[0](u_5)$ remains 1; and (iii) the event $ref_{10,y}$ for the object referenced by `y`, causing the object u_4 to change its automaton state to 1, i.e., setting the predicate $s[1]$ to 1 for u_4, and setting the predicate $s[0]$ to 0. After applying the above updates we end up with the logical structure shown in Fig. 3(b), reflecting both the changes in the store, and the transitions in the automaton state for program objects.

4 An Abstract Semantics

In this section, we present a conservative abstract semantics [10] abstracting the concrete semantics of Section 3. In Section 4.1, we describe how abstract configurations are used to finitely represent multiple concrete configurations. In Section 4.2, we describe an abstract semantics manipulating abstract configurations.

4.1 Abstract Program Configurations

We conservatively represent multiple concrete program configurations using a single logical structure with an extra truth-value $1/2$ which denotes values which may be 1 and may be 0.

Definition 4 (Abstract Configuration). *An* abstract configuration *is a* 3*-valued logical structure* $C = \langle U, \iota \rangle$ *where:*

- U *is the universe of the* 3*-valued structure. Each individual in* U *represents possibly many allocated heap objects.*
- ι *is the interpretation function mapping predicates to their truth-value in the structure, i.e., for every predicate* $p \in P$ *of arity* k, $\iota(p) \colon U^k \to \{0, 1/2, 1\}$. *For example,* $\iota(p)(u) = 1/2$ *indicates that the truth value of* p *may be* 1 *for some of the objects represented by* u *and may also be* 0 *for some of the objects represented by* u.

We allow an abstract configuration to include a *summary node*, i.e., an individual which corresponds to one or more individuals in a concrete configuration represented by that abstract configuration. Technically, we use a designated unary predicate *sm* to

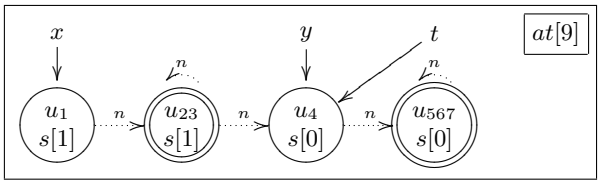

Fig. 4. An abstract program configuration representing the concrete configuration of Fig. 3(a).

maintain summary-node information. A summary node u has $sm(u) = 1/2$, indicating that it may represent more than one node.

Abstract program configurations are depicted by enhancing the directed graphs from Section 3 with a graphical representation for $1/2$ values: a binary predicate $p(u_1, u_2)$ which evaluates to $1/2$ is drawn as dashed directed edge from u_1 to u_2 labelled with the predicate symbol, and a summary node is drawn as circle with double-line boundaries.

Example 5. The abstract configuration shown in Fig. 4 represents the concrete configuration of Fig. 3(a). The summary node labelled by u_{23} represents the linked-list items u_2 and u_3, both having the same values for their unary predicates. Similarly, the summary node u_{567} represents the nodes u_5, u_6, and u_7.

Note that this abstract configuration represents many configurations. For example, it represents any configuration in which program execution is immediately after line 10 and a linked-list with at least 5 items has been traversed up to some item after the third item.

Embedding We now formally define how configurations are represented using abstract configurations. The idea is that each individual from the (concrete) configuration is mapped into an individual in the abstract configuration. More generally, it is possible to map individuals from an abstract configuration into an individual in another less precise abstract configuration. The latter fact is important for our abstract transformer.

Formally, let $C = \langle U, \iota \rangle$ and $C' = \langle U', \iota' \rangle$ be abstract configurations. A function $f\colon U \to U'$ such that f is surjective is said to *embed C into C'* if for each predicate p of arity k, and for each $u_1, \ldots, u_k \in U$ one of the following holds:

$$\iota(p(u_1, \ldots, u_k)) = \iota'(p(f(u_1), \ldots, f(u_k))) \text{ or } \iota'(p(f(u_1), \ldots, f(u_k))) = 1/2$$
$$\textit{and}$$
$$\text{for all } u' \in U' \text{ s.t. } |\{u \mid f(u) = u'\}| > 1 : \iota^{S'}(sm)(u') = 1/2$$

One way of creating an embedding function f is by using *canonical abstraction*. Canonical abstraction maps concrete individuals to an abstract individual based on the values of the individuals' unary predicates. All individuals having the same values for unary predicate symbols are mapped by f to the same abstract individual. Only summary nodes (i.e., nodes with $sm(u) = 1/2$) can have more than one node mapped to them by the embedding function.

Note that since automaton states are represented using unary predicates, the abstraction is refined by the automaton state of each object. This provides a simple property-guided abstraction since individuals at different automaton states are not summarized

together. Indeed, adding unary predicates to the abstraction increases the worst-case cost of the analysis. However, as noted in [26] in practice this abstraction refinement often decreases significantly the cost of the analysis. Finally, our analysis is relational, allowing multiple 3-valued logical structures at a single program point, reflecting different behaviors.

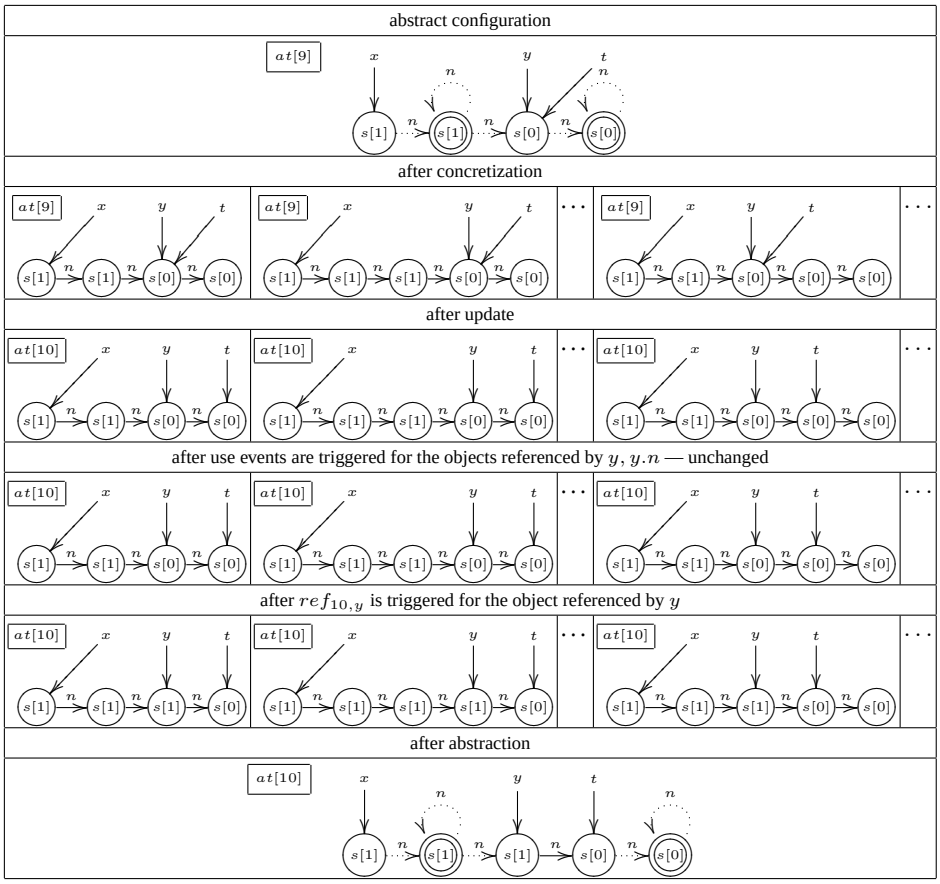

Fig. 5. Concretization, predicate-update, automaton transition updates, and abstraction for the statement `t = y.n` in line 10.

4.2 Abstract Semantics

Implementing an abstract semantics directly manipulating abstract configurations is non-trivial since one has to consider all possible relations on the (possibly infinite) set of represented concrete configurations.

The *best* conservative effect of a program statement [10] is defined by the following 3-stage semantics: (i) a concretization of the abstract configuration is performed, resulting in all possible configurations *represented* by the abstract configuration; (ii) the program statement is applied to each resulting concrete configuration; (iii) abstraction

of the resulting configurations is performed, resulting with a set of abstract configurations *representing* the results of the program statement.

Example 6. Fig. 5 shows the stages of an abstract action: first, concretization is applied to the abstract configuration resulting with an infinite set of concrete configuration represented by it. the program statement update is then applied to each of these concrete configurations. Following the program statement update, automaton transition updates are applied as described in Section 3.2. That is, at first *use* events are triggered to update the automaton states of the objects referenced by `y` and `y.n`. Then, a $ref_{10,y}$ event is triggered to update the automaton state of the object referenced by `y`. Finally, after all transition updates have been applied, the resulting concrete configurations are abstracted resulting with a finite representation.

Our prototype implementation described in Section 6.1 operates directly on abstract configurations using *abstract transformers*, thereby obtaining actions which are more conservative than the ones obtained by the best transformers. Interestingly, since temporal information is encoded as part of the concrete configuration via automaton state predicates, the soundness of the abstract transformers is still guaranteed by the *Embedding Theorem* of [26]. Our experience shows that the abstract transformers used in the implementation are still precise enough to allow verification of our heap safety properties.

When the analysis terminates, we verify that in all abstract configurations, all individuals are associated with an accepting automaton state, i.e., in all abstract configurations, for every individual o, the predicate $s[err](o)$ evaluates to 0.

The soundness of our abstraction guarantees that this implies that in all concrete configurations, all individuals are associated with an accepting automaton state, and we conclude that the property holds.

5 Extensions

In this section, we extend the applicability of our framework by: (i) formulating an additional compile-time memory management property — the assign-null property; and (ii) extending the framework to simultaneously verify multiple properties.

5.1 Assign-Null Analysis

The assign-null problem determines source locations at which statements assigning null to heap references can be safely added. Such null assignments lead to objects being unreachable earlier in the program, and thus may help a run-time garbage collector collect objects earlier, thus saving space. As in Section 2, we show how to verify the assign-null property for a single program point and discuss efficient verification for a set of program points in Section 5.2.

Definition 5 (Assign-Null Property $\langle pt, \mathtt{x}, \mathtt{f} \rangle$). *Given a program point pt, an arbitrary reference expression* `x`*, and a reference field* `f`*, an assign-null property $\langle pt, \mathtt{x}, \mathtt{f} \rangle$ states that a reference $x.f$ at pt is not used further before being redefined on execution paths starting at pt. Therefore, it is safe to add an* `x.f = null` *statement immediately after pt.*

The assign-null property allows us to assign null to a dead heap reference. As in the free property case, our assign-null property can also handle arbitrary reference expressions by introducing a new program variable `t`, assigned with `exp` just after a program point pt.

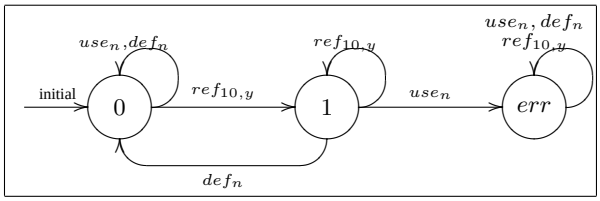

Fig. 6. A heap safety automaton $A^{an}_{10,y,n}$ for assign null to $y.n$ at 10.

Example 7. Consider again the example program of Fig. 1. We would like to verify that a `y.n = null` statement can be added immediately after line 10, i.e., a reference connecting consecutive list elements can be assigned null as soon as it is traversed in the loop. The HSA $A^{an}_{10,y,n}$ shown in Fig. 6 represents the assign-null $\langle 10, \mathtt{y}, \mathtt{n}\rangle$ property. The alphabet of this automaton consists of the following events for an object o: (i) use_n, which corresponds to a use of the field n of the object; (ii) def_n, which corresponds to a definition of the field n of the object; (iii) $ref_{10,y}$, which is triggered when program execution is immediately after execution of the statement in 10 and `y` references the object o. Our implementation verifies assign-null $\langle 10, \mathtt{y}, \mathtt{n}\rangle$ property, by applying the framework with $A^{an}_{10,y,n}$ to the example program. Notice that this automaton contains a back arc and thus is more complex than the one for the free property.

An arbitrary assign-null property is formulated as a heap safety property using an HSA similar to the one shown in Fig. 6 where the program point, variable and field names are set accordingly.

5.2 Simultaneous Verification of Multiple Properties

So far we showed how to verify the free and assign-null properties for a single program point. Clearly, in practice one wishes to verify these properties for a set of program points without repeating the verification procedure for each program point. Our framework supports simultaneous verification of multiple properties, and in particular verification of properties for multiple program points. Assuming $\text{HSA}_1, \ldots, \text{HSA}_k$ describe k verification properties, then k automaton states $s_1, \ldots, s_k$ are maintained for every program object, where s_i maintains an automaton state for HSA_i. Technically, as described in Section 3, a state s_i of an individual o is represented by automaton state predicates $s_i[q](o)$, where q ranges over the states of HSA_i. The events associated with the automata $\text{HSA}_1, \ldots, \text{HSA}_k$ at a program point are triggered simultaneously, updating the corresponding automaton state predicates of individuals.

Interestingly, if we limit our verification of free $\langle pt, \mathtt{x}\rangle$ properties to ones where `x` is used at pt (i.e., `x` is used in the statement at pt), then the following features are obtained: (i) an object is freed just after it is referenced last, i.e., exactly at the earliest time possible, and (ii) an object is freed "exactly once", i.e., there are no redundant frees of variables referencing the same object.

A similar choice for assign-null properties assigns null to a heap reference immediately after its last use. The motivation for this choice of verification properties comes from our previous work [31], showing an average of 15% potential space savings beyond a run-time garbage collector if a heap reference is assigned null just after its last use. However, we note that our framework allows verification of arbitrary free and assign-null properties, which may yield further space reduction. In fact, in [31] we show an average of 39% potential space savings beyond a run-time garbage collector assuming complete information on the future use of heap references.

6 Empirical Results

We implemented the static analysis algorithms for verifying free and assign-null properties, and applied it to several programs, including JavaCard programs.

Our benchmarks programs were used as a proof of concept, thus we did not measure the total savings obtained by our analysis. In particular the benchmarks provide three kinds of proof of concept: (i) we use small programs manipulating a linked-list to demonstrate the precision of our technique; (ii) we demonstrate how our techniques could be used to verify/automate manual space-savings rewritings. In particular, in our previous work [29] the code of the `javac` Java compiler was manually rewritten in order to save space. Here, we verify the manual rewritings in `javac`, which assign null to heap references, by applying our prototype implementation to a Java code fragment emulating part of the Parser facility of `javac`; (iii) we demonstrate how our techniques could play an important role in the design of future JavaCard programs. This is done by rewriting existing JavaCard code in a more modular way, and showing that our techniques may be used to avoid the extra space overhead due to the modularity.

6.1 Implementation

Our implementation consists of the following components: (i) a front-end, which translates a Java program (.class files) to a TVLA program [21]; (ii) an analyzer, which analyzes the TVLA program; (iii) a back-end, which answers our verification question by further processing of the analyzer output.

The front end (J2TVLA), developed by R. Manevich, is implemented using the Soot framework [33]. The analyzer, implemented using TVLA, includes the implementation of static analysis algorithms for the free and assign-null property verification. TVLA is a parametric framework that allows the heap abstractions and the abstract transformers to be easily changed. In particular, for programs manipulating lists we obtain a rather precise verification algorithm by relying on spatial instrumentation predicates, that give sharing, reachability and cyclicity information for heap objects [26]. For other programs, allocation-site information for heap objects suffices for the verification procedure. In both abstractions interprocedural information is computed [24]. Finally, our implementation allows simultaneous verification of several free or assign-null properties, by maintaining several automaton states per program object.

The back-end, implemented using TVLA libraries, traverses the analysis results, i.e., the logical structures at every program point, and verifies that all individuals are associated with an accepting state. For a single property, we could abort the analyzer upon reaching a non-accepting state on some object and avoid the back-end component.

Program	Description	Free		Assign Null	
		space	time	space	time
Loop	the running example	1.71	1.93	1.37	1.76
CReverse	constructive reverse of a list	3.03	5.17	2.58	4.79
Delete	delete an element from a list	5.33	19.66	4.21	13.84
DLoop	doubly linked list variant of Loop	2.09	2.91	1.75	2.68
DPairs	processing pairs in a doubly-linked list	2.76	5.01	2.54	4.86
small javac	emulation of `javac`'s parser facility	N/A	N/A	16.02	43.84
JavaPurse' slice	a JavaCard simple electronic purse	56.3	979	56.15	991
GuessNumber' slice	a JavaCard distributed guess number game	9.99	17.3	N/A	N/A

Table 3. Analysis cost for the benchmark programs. Space is measured in MB, and time is measured in seconds.

However, in the case of simultaneous verification of multiple safety properties, this would not work and the back-end is required.

6.2 Benchmark Programs

Table 3 shows our benchmark programs. The first 4 programs involve manipulations of a singly-linked list. `DLoop, DPairs` involve a doubly-linked list manipulation. `small javac` is motivated by our previous work [29], where we manually rewrite the code of the `javac` compiler, issuing null assignments to heap references. We can now verify our manual rewriting by applying the corresponding assign-null properties to Java code emulating part of the Parser facility in `javac`.

The last two benchmarks are JavaCard programs. `JavaPurse` is a simple electronic cash application, taken from Sun JavaCard samples [18]. In `JavaPurse` a fixed set of loyalty stores is maintained, so every purchase grants loyalty points at the corresponding store. `GuessNumber` [23] is a guess number game over mobile phone SIM cards, where one player (using a mobile phone) picks a number, and other players (using other mobile phones) try to guess the number.

Due to memory constraints, JavaCard programs usually employ a static allocation regime, where all program objects are allocated when the program starts. This leads to non-modular and less reusable code, and to more limited functionality. For example, in the `GuessNumber` program, a global buffer is allocated when the program starts and is used for storing either a server address or a phone number. In `JavaPurse`, the number of stores where loyalty points are granted is fixed.

A better approach that addresses the JavaCard memory constraints is to rewrite the code using a natural object-oriented programming style, and to apply static approaches to free objects not needed further in the program. Thus, we first rewrite the JavaCard programs to allow more modular code in the case of `GuessNumber`, and to lift the limitation on the number of stores in `JavaPurse`. Then, we apply our free analysis to the rewritten code, and verify that an object allocated in the rewritten code can be freed as soon it is no longer needed. In `JavaPurse` we also apply our assign null analysis and verify that an object allocated in the rewritten code can be made unreachable as soon it is no longer needed (thus, a run-time garbage collector may collect it). Concluding, we show that in principle the enhanced code bears no space overhead compared to the original code when the free or the assign-null analysis is used.

6.3 Results

Our experiments were done on a 900 Mhz Pentium-III with 512 MB of memory running Windows 2000. Table 3 shows the space and time the analysis takes. In `Loop` we verify our free $\langle 10, \mathtt{y} \rangle$ and assign-null $\langle 10, \mathtt{y}, \mathtt{n} \rangle$ properties. For `CReverse` we verify an element of the original list can be freed as soon it is copied to the reversed list. In `Delete` we show an object can be freed as soon it is taken out of the list (even though it is still reachable from temporary variables). Turning to our doubly linked programs, we also show objects that can freed immediately after their last use, i.e., when an object is traversed in the loop (`DLoop`), and when an object in a pair is not processed further (`DPairs`). We also verify corresponding null-assignments that make an object unreachable via heap references as soon these references are not used further.

For `small javac` we verify that heap references to large objects in a parser class may be assigned null just after their last use. Finally, for scalability reasons we analyze slices of rewrritten JavaCard programs. Our current implementation does not include a slicer, thus we manually slice the code. Using the sliced programs we verify that objects allocated due by our rewritings, can be freed as soon they are not needed.

7 Related Work

One of the main difficulties in verifying local temporal heap safety properties is considering the effect of aliasing in a precise-enough manner. Some of the previous work on software verification allows universally quantified specifications similar to our local heap safety properties (e.g., [4,9]). We are the first to apply such properties to compile-time memory management and to employ a high-precision analysis of the heap.

ESP [11] uses a preceding pointer-analysis phase and uses the results of this phase to perform finite-state verification. Separating verification from pointer-analysis may generally lead to imprecise results.

Some prior work used automata to dynamically monitor program execution and throw an exception when the property is violated (e.g.,[27,8]). Obviously, dynamic monitoring cannot verify that the property holds for all program executions.

Recoding history information for investigating a particular local temporal heap safety property was used for example in [16,25] (approximating flow dependencies) and [20] (verification of sorting algorithms). The framework presented here generalizes the idea of recording history information by using a heap safety automaton.

Our free property falls in the *compile-time garbage collection* research domain, where techniques are developed to identify and recycle garbage memory cells at compile-time. Most work has been done for functional languages [5,17,12,14,19]. In this paper, we show a free analysis, which handles a language with destructive updates, that may reclaim an object still reachable in the heap, but not needed further in the run.

Escape analysis (e.g., [7]), which allows stack allocating heap objects, has been recently applied to Java. In this technique an object is freed as soon as its allocating method returns to its caller. While this technique has shown to be useful, it is limited to objects that do not escape their allocating method. Our technique applies to all program objects, and allows freeing objects before their allocating method returns.

In region-based memory management [6,32,2,13], the lifetime of an object is predicted at compile-time. An object is associated with a memory region, and the alloca-

tion and deallocation of the memory region are inferred automatically at compile time. It would be interesting to instantiate our framework with a static analysis algorithm for inferring earlier deallocation of memory regions.

Liveness analysis [22] may be used in the context of a run-time to reduce the size of the root set (i.e., ignoring dead stack variables and dead global variables) or to reduce the number of scanned references (i.e., ignoring dead heap references). In [3,1,15] liveness information for root references is used to reclaim more space.

In [31] we conduct dynamic measurements estimating the potential space savings achieved by communicating the liveness of stack variable references, global variables references and heap references to a run-time garbage collector. We conclude there that heap liveness information yields a potential for space savings significantly larger than the one achieved by communicating liveness information for stack and global variables. One way of communicating heap liveness information to a run-time GC is by assigning null to heap references. In this paper we present a static analysis algorithm for assigning null to heap references.

8 Conclusion

In this paper we present a framework for statically reasoning about local temporal heap safety properties. This framework is instantiated to produce two new static analysis algorithms for calculating the liveness of heap objects (free property) and heap references (assign-null property). Our initial experience shows evidence for the precision of our techniques, leading to space savings in Java programs. In the future we intend to apply our techniques to more "real-world" programs by integrating a code slicer and cheaper pointer analysis algorithms. It may be also interesting to explore opportunites for deallocating space using richer constructs than `free exp`. For example, using a new `free-list` construct for deallocating an entire list.

Acknowledgements

We would like to thank Giesecke & Devrient, Munich for their assistance and financial support. We would like to thank Roman Manevich and Thomas Stocker for many insights contributing to this research. Finally, we thank Nurit Dor for providing useful comments on earlier drafts of this paper.

References

1. O. Agesen, D. Detlefs, and E. Moss. Garbage Collection and Local Variable Type-Precision and Liveness in Java Virtual Machines. In *Prog. Lang. Design and Impl.*, June 1998.
2. M. F. Alexander Aiken and R. Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *Prog. Lang. Design and Impl.*, June 1995.
3. A. W. Appel. *Compiling with Continuations*, chapter 16, pages 205–214. CUP, 1992.
4. T. Ball and S. Rajamani. SLIC: A Specification Language for Interface Checking (of C). Technical Report MSR-TR-2001-21, MSR, 2001.
5. J. M. Barth. Shifting garbage collection overhead to compile time. *Commun. ACM*, 20(7):513–518, 1977.
6. L. Birkedal, M. Tofte, and M. Vejlstrup. From region inference to von neumann machines via region representation inference. In *Symp. on Princ. of Prog. Lang.*, pages 171–183, 1996.

7. B. Blanchet. Escape analysis for object oriented languages. application to Java^{tm}. In *Conf. on Object-Oriented Prog. Syst., Lang. and Appl.*, Denver, 1998.
8. T. Colcombet and P. Fradet. Enforcing trace properties by program transformation. In *Proc. of 27th POPL*, pages 54–66, Jan. 19–21, 2000.
9. J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, R. Shawn, and L. Hongjun. Bandera: Extracting finite-state models from Java source code. In *Proc. 22nd ICSE*, June 2000.
10. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Symp. on Princ. of Prog. Lang.*, New York, NY, 1979. ACM Press.
11. M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *Prog. Lang. Design and Impl.*, pages 57–68, Berlin, June 2002.
12. I. Foster and W. Winsborough. Copy avoidance through compile-time analysis and local reuse. In *Proceedings of International Logic Programming Sympsium*, pages 455–469, 1991.
13. N. Hallenberg, M. Elsman, and M. Tofte. Combining region inference and garbage collection. In *Prog. Lang. Design and Impl.*, pages 141–152, Berlin, 2002.
14. G. W. Hamilton. Compile-time garbage collection for lazy functional languages. In *Memory Management, International Workshop IWMM 95*, 1995.
15. M. Hirzel, A. Diwan, and A. L. Hosking. On the usefulness of type and liveness accuracy for garbage collection and leak detection. In *Trans. on Prog. Lang. and Syst.*, 2002.
16. S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *Prog. Lang. Design and Impl.*, pages 28–40, New York, NY, 1989. ACM Press.
17. K. Inoue, H. Seki, and H. Yagi. Analysis of functional programs to detect run-time garbage cells. *Trans. on Prog. Lang. and Syst.*, 10(4):555–578, Oct. 1988.
18. Java card 2.2 development kit. Available at java.sun.com/products/javacard.
19. R. Jones. *Garbage Collection. Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, 1999.
20. T. Lev-Ami, T. W. Reps, R. Wilhelm, and S. Sagiv. Putting static analysis to work for verification: A case study. In *ISSTA*, pages 26–38, 2000.
21. T. Lev-Ami and M. Sagiv. TVLA: A framework for kleene based static analysis. In *Static Analysis Symposium*. Springer, 2000.
22. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
23. Oberthur card systems. www.oberthurcs.com.
24. N. Rinetzky and M. Sagiv. Interprocedural shape analysis for recursive programs. *Lecture Notes in Computer Science*, 2027:133–149, 2001.
25. J. Ross and M. Sagiv. Building a bridge between pointer aliases and program dependences. In *Proceedings of the 1998 European Symposium On Programming*, Mar. 1998.
26. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(3):217–298, 2002.
27. F. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, Feb. 2000.
28. R. Shaham, E. K. Kolodner, and M. Sagiv. Automatic removal of array memory leaks in java. In *Int. Conf. on Comp. Construct.* Springer, Apr. 2000.
29. R. Shaham, E. K. Kolodner, and M. Sagiv. Heap profiling for space-efficient java. In *Prog. Lang. Design and Impl.* ACM, June 2001.
30. R. Shaham, E. K. Kolodner, and M. Sagiv. Backward shape analysis to statically predict heap behavior. Unpublished manuscript, 2002.
31. R. Shaham, E. K. Kolodner, and M. Sagiv. Estimating the impact of heap liveness information on space consumption in java. In *Int. Symp. on Memory Management*. ACM, 2002.
32. M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value lambda-calculus using a stack of regions. In *Symp. on Princ. of Prog. Lang.*, pages 188–201, 1996.
33. R. Vallée-Rai, L. Hendren, V. Sundaresan, E. G. P. Lam, and P. Co. Soot - a java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.