# Establishing local temporal heap safety properties with applications to compile-time memory management

Ran Shaham[a,*], Eran Yahav[b], Elliot K. Kolodner[a],
Mooly Sagiv[b]

[a]*IBM Haifa Research Lab, University Campus, Carmel Mountains, Haifa 31905, Israel*
[b]*School of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel*

## 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.
© 2005 Elsevier B.V. All rights reserved.

*Keywords:* Abstract interpretation; Memory liveness; Garbage collection; Shape analysis; Safety properties; Verification

* Corresponding author.
*E-mail addresses:* ran.shaham@gmail.com (R. Shaham), yahave@post.tau.ac.il (E. Yahav),
kolodner@il.ibm.com (E.K. Kolodner), msagiv@post.tau.ac.il (M. Sagiv).

## 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.
(4) We show that our heap abstraction is precise enough to verify interesting compile-time memory management properties, while other points-to based heap abstractions fail to verify our properties of interest.

### 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.

During the analysis events are triggered for state machines associated with objects. It is important to note that our framework implicitly allows infinite state machines, since the number of objects is unbounded, and a state machine is associated with every object. Thus, precise information on heap paths for disambiguating 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 are not needed further, even though they are reachable. In previous work [34,36] we reported on dynamic experiments that show on average a potential for saving 39% of the space, 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 per cent 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 and variables for which it is safe to insert a free statement in order to deallocate a garbage element.
**assign-null analysis**   Statically identify source locations, variables and fields for 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 [36] 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., [21]), 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., [21]) will not issue such a free statement, since the element referenced by x 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 a 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.

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

```
      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.

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 following, 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 representation 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 [31]. We use *canonical abstraction* that maps concrete program objects (i.e., individuals in a logical structure) to abstract program objects on the basis of 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.

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

In our framework, an observable event is derived from the program state and the current statement. We assume the observable events are part of the specification. 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. For example, an object $o$ in automaton state $q$ is updated by automaton transition $\alpha$ to have a new automaton state $\delta(q, \alpha)$, if $o$ is associated with the observable event $\alpha$ occurring in the current program statement.

The states in the automaton capture history information on memory locations. Transitions in the automaton capture the changes in the history information when a statement corresponding to the event is executed. This can be formalized using trace

semantics. To make the material more accessible, we use automata directly and define self-explanatory events.

### 2.1. Free property

We now formulate the free property, which allows us to issue a free statement to reclaim objects unneeded further in the run. In the following, 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.

In order to formulate the free property we first consider the notions of a program state and a program trace. A *program state* $\sigma_i = \langle store_i, pt_i \rangle$ represents the global state of the program, which consists of the store ($store_i$) and the current program point ($pt_i$). A *trace* $\pi = \sigma_1, \sigma_2, \ldots$ is a (possibly infinite) sequence of program states $\sigma_i$. A trace reflects a program execution.

In order to define the *free* property, we also define the notion of *dynamic location liveness*.

**Definition 2** (*Dynamic Location Liveness*). **A memory location l is dynamically live in a program state $\sigma_i$ along a trace** $\pi$ if (i) $l$ is used in $\sigma_j$, for some $j \geq i$, and (ii) $l$ is not assigned in all $\sigma_i, \ldots, \sigma_{j-1}$.

Intuitively, an object can be collected as soon as its references are no longer used. This observation leads to the following intuitive definition of the free property.

**Definition 3** (*Free Property $\langle pt, x \rangle$*). **The free property $\langle \mathbf{pt}, x \rangle$ holds** if there exists no trace $\pi$ with a program state $\sigma_i = \langle store_i, pt \rangle$ such that there exists a reference to the object referenced by x in $\sigma_{i+1}$, which is dynamically live in $\sigma_{i+1}$ in $\pi$.

The free property allows us to free an object that is not needed further in the run. In particular, when a free property $\langle pt, x \rangle$ holds for a program point $pt$ and a reference variable x, it guarantees that it is *safe* to issue a free(x) statement immediately after $pt$. That is, it guarantees that adding such a free(x) statement preserves the semantics of the original program (for a more formal treatment of semantic preserving transformations see [33]). Interestingly, such an object can still be reachable from a program variable through a heap path. For simplicity, we assume that a free(x) statement does nothing (and in particular does not abort) when x references the special null value. Finally, for expository purposes, we only present the free property for an object referenced by a program variable. However, this free property can easily handle 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.

### 2.1.1. Free property for the running example

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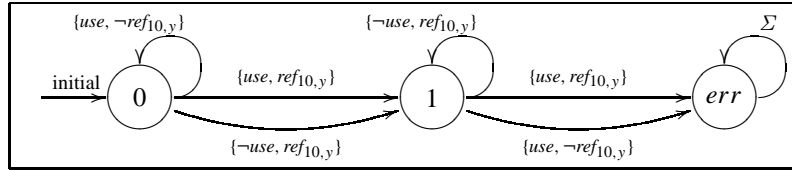
Fig. 2. A heap safety automaton $A_{10,y}^{free}$ for free y at line 10.

The HSA $A_{10,y}^{free}$ 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. 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 program variable are set accordingly. In particular, for a free property $\langle pt, x \rangle$, the corresponding HSA $A_{pt,x}^{free}$ may be obtained from the automaton in Fig. 2 by replacing 10 with $pt$, and by replacing $y$ with $x$.

The alphabet of the automaton consists of sets of observable object attributes. For the purpose of verifying the free property, we maintain the following object attributes in the instrumented semantics (see Section 3) for an object $o$: (i) *use* attribute, which holds for $o$ if the r-value of reference expression $e$ (of the form x or of the form x.f) is used in the current statement execution, and the r-value of $e$ is $o$, and (ii) $ref_{10,y}$ attribute, which holds for $o$ if the program execution is immediately after execution of the statement at line 10 and y references $o$ after the execution of the statement at line 10.

On the basis of the above object attributes we define the alphabet of the HSA $A_{10,y}^{free}$ to be $\Sigma = \{\{use, ref_{10,y}\}, \{use, \neg ref_{10,y}\}, \{\neg use, ref_{10,y}\}\}$. For readability purposes, we show for a set of attributes (an alphabet symbol) the attributes that hold for an object as well as the attributes that do not hold for an object.[1] For example, the alphabet symbol $\{use, \neg ref_{10,y}\}$ denotes that the attribute *use* holds for an object (i.e., a reference to that object is used in the current statement), while the attribute $ref_{10,y}$ does not hold for that object (i.e., either the current statement is not at $pt$, or this object is not referenced by y after the current statement is executed). Finally, we use $\Sigma$ in the self-loop emanating from the *err* state (see Fig. 2) as shorthand expressing the fact that for all alphabet symbols the *err* state may only be transitioned to itself (i.e., when reaching the violation state, the automaton state cannot be changed, since the property is violated).

The HSA is in an accepting state along an execution path if and only if $o$ 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 $A_{10,y}^{free}$ (state 0). Then, a use of a reference to an object $o$ (the *use* attribute holds for $o$) when the program execution is not immediately after line 10 (the $ref_{10,y}$ attribute does not hold for $o$) does not change the state of $A_{10,y}^{free}$ for $o$ (the self-loop on state 0 labelled with $\{use, \neg ref_{10,y}\}$ is taken).

---

[1] An equivalent way of writing the alphabet would be $\Sigma = \{\{use, ref_{10,y}\}, \{use\}, \{ref_{10,y}\}\}$, where only attributes that hold for an object are shown.

Table 1
Predicates for partial Java semantics

| Predicates | Intended meaning |
| --- | --- |
| $after[pt]()$ | 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 $o_2$ |
| $use(o)$ | A reference to $o$ is used in the current program statement |
| $ref_{pt,x}(o)$ | $o$ is referenced by $x$ and the execution is immediately after $pt$ |
| $s[q](o)$ | The current state of $o$'s automaton is $q$ |

When the program is immediately after line 10 and $y$ references an object $o$ (the $ref_{10,y}$ attribute holds for $o$), $o$'s automaton state is set to 1 (if the *use* attribute holds for $o$ the labelled edge $\{use, ref_{10,y}\}$ is taken; otherwise if the *use* attribute does not hold for $o$ then the labelled edge $\{\neg use, ref_{10,y}\}$ is taken). If a reference to $o$ is used further (i.e., in the subsequent program configurations along the execution path a reference to $o$ is used), and $o$'s automaton state is 1, the automaton state for $o$ reaches the violation state of the automaton (either via the $\{use, ref_{10,y}\}$ edge or via the $\{use, \neg ref_{10,y}\}$ edge). 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.

## 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 4** (*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 $k$, $\iota^{\natural}(p): U^{\natural k} \to \{0, 1\}$.

We use the predicates of Table 1 to record information used by the properties discussed in this paper. The nullary predicate $after[pt]()$ records the program location
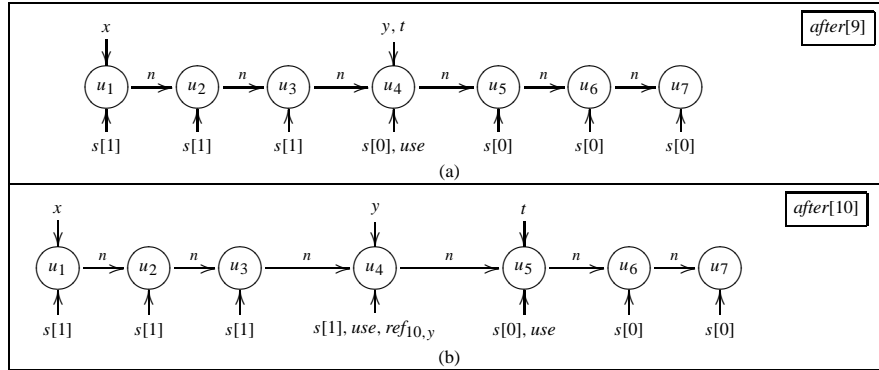
Fig. 3. Concrete program configurations (a) before and (b) immediately after execution of `t = y.n` at line 10.

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 $o_2$.

The predicates $use(o)$ and $ref_{pt,x}$ maintain the object attributes needed for triggering events in the HSA $A_{pt,x}^{free}$. We describe these object attributes more completely in Sections 3.2.2 and 3.2.3.

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 [31]).

In this paper, program configurations are depicted as directed graphs. Each individual of the universe is displayed as a node. A unary predicate of the form $p(o)$ is shown as an edge from the predicate symbol to a node in which it holds. 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.

**Example 5.** 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 *after*[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 predicate $use(o)$ holds for an object $o$ if a reference to $o$ is used in the current statement. For example, a reference

Table 2
Use attributes set by program statements

| Statement | Use attribute is set to *true* 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 | $x$, $y$ |

to $u_4$ is used (due the use of y in the statement at line 9); thus we see an edge connecting *use* and $u_4$. The predicate $ref_{10,y}$ does not hold for any objects in this configuration, since the execution is not immediately after line 10. Finally, 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.

### 3.2. Operational semantics

Program statements are modelled by generating the logical structure representing the program state after execution of the statement. First-order logical formulae can be used to formally define the effect of every statement (see [31]). In particular, first-order logical formulae are used to model the change of the automaton state of every affected individual.

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 (see Section 7). We now show how program statements are associated with $A_{pt,x}^{free}$ 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.

#### 3.2.1. Object allocation

For a program statement x = new C(), 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 6.** Consider the HSA $A_{10,y}^{free}$ of the example in Section 2.1.1. 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.

#### 3.2.2. Maintaining the use attribute

The *use* attribute reflects information for an object depending on the current state of the program. Thus, conceptually, this means that before executing a statement the *use* attribute is set to *false* for all program objects, and then the *use* property is set to *true* for some of the objects depending on the executed program statement, as shown in Table 2.

In general, a use of a program variable x in a program statement updates the *use(o)* attribute to 1 for the object referenced by x. In addition, a use of the field f of the object referenced by x in a program statement updates the *use(o)* attribute to 1 for the object referenced by x.f. For example, as shown in Table 2, the statement x = y.f sets *use(o)* to 1 for the objects referenced by $y$ and $y.f$.

### 3.2.3. Maintaining the $ref_{pt,x}$ attribute

As in the case of the *use* attribute, the $ref_{pt,x}$ attribute reflects information for an object depending on the current state of the program. Thus, conceptually, this means that before executing a statement the $ref_{pt,x}$ attribute is set to *false* for all program objects, and then this property is set to *true* for some of the objects depending on the program statement executed. In particular, we set the $ref_{pt,x}$ attribute to true for the object referenced by $x$ when the execution is immediately after *pt* (i.e., when the currently executed statement is at program point *pt*). For example, for the $ref_{10,y}$ attribute, $ref_{10,y}(o)$ is set to 1 for the object referenced by $y$, when the execution is immediately after line 10.

### 3.2.4. Maintaining s[q] predicates

We can now determine the transition taken in the automaton for an object $o$ changing its associated automaton state from $q_i$ to $q_j$. The idea is that an edge emanating from $q_i$ is taken if the label on that edge matches the values of $o$'s *use*, $ref_{pt,x}$ attributes. For example, in our running example, if an object $o$ is associated with state 0, and both *use*, $ref_{10,y}$ attributes hold for $o$, then the edge labelled {*use*, $ref_{10,y}$} connecting state 0 to state 1 (see Fig. 2) is taken, updating $s[0](o)$ to 0, and $s[1](o)$ to 1. In general, a transition from state $q_i$ to state $q_j$ for an object $o$ is reflected by setting $s[q_i](o)$ to 0, and setting $s[q_j](o)$ to 1.

**Example 7.** 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 *after*[10]() to 1 and *after*[9]() to 0. Then, *use(o)* is set to 1 for both $u_4$, $u_5$. This is due to the use of y and y.f in this statement. Also, $ref_{10,y}(o)$ is set to 1 for $u_4$, since the execution is after line 10 and $u_4$ is referenced by y at that time.

We can now update the automaton states associated with program objects. For $u_4$ the current associated automaton state is 0. The attributes *use*, $ref_{10,y}$ hold for $u_4$; thus, the {*use*, $ref_{10,y}$} edge connecting automaton state 0 to automaton state 1 is taken, updating $s[0](u_4)$ to 0, and $s[1](u_4)$ to 1. In addition, for $u_5$, the attribute *use* holds, and the attribute $ref_{10,y}$ does not hold; thus the {*use*, $\neg ref_{10,y}$} edge connecting state 0 to itself is taken, leaving $s[0](u_5)$ unchanged with the value 1.

## 4. An abstract semantics

In this section, we present a conservative abstract semantics [11] 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.
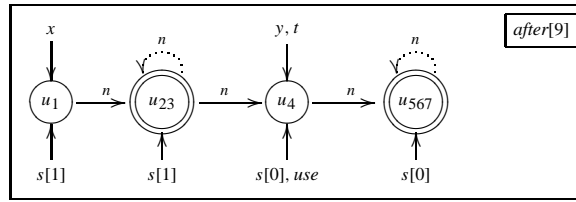
Fig. 4. An abstract program configuration representing the concrete configuration of Fig. 3(a).

### 4.1. Abstract program configurations

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

**Definition 8** (*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.
- $\iota$ is the interpretation function mapping predicates to their truth-value in the structure, i.e., for every predicate $p \in P$ of arity $k$, $\iota(p): U^k \rightarrow \{0, 1/2, 1\}$. 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 maintain summary-node information. A summary node $u$ has $sm(u) = 1/2$, indicating that it may represent more than one node. An individual with $sm(u) = 0$ corresponds to exactly one individual in a concrete configuration. For technical reasons we do not allow $sm(u)$ to be 1.

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 9.** 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 $u_3$, 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 4 items has been traversed up to some item after the third item.

#### 4.1.1. 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 transformers.

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

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

One way of creating an embedding function $f$ is by using *canonical abstraction*. Canonical abstraction maps concrete individuals to an abstract individual on the basis of 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.

Since automaton states are represented using unary predicates, the soundness of our approach is guaranteed by the *embedding theorem* of [31]. For a given program and HSA, if there exists a concrete program state in which the automaton is in its error state (according to the instrumented semantics of Section 3), then embedding guarantees that there exists an abstract state in which the automaton is possibly in its error state.

Moreover, using unary predicates to represent automaton states also refines the abstraction 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 [31], in practice this abstraction refinement often decreases significantly the cost of the analysis. Finally, our analysis allows multiple 3-valued logical structures at a single program point, reflecting different behaviors.

### 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 [11] is defined by the following three-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 in a set of abstract configurations *representing* the results of the program statement.

**Example 10.** 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. The program statement update also includes the update of the *use* and *ref*$_{pt,x}$, attributes, and the application of automaton transition updates described
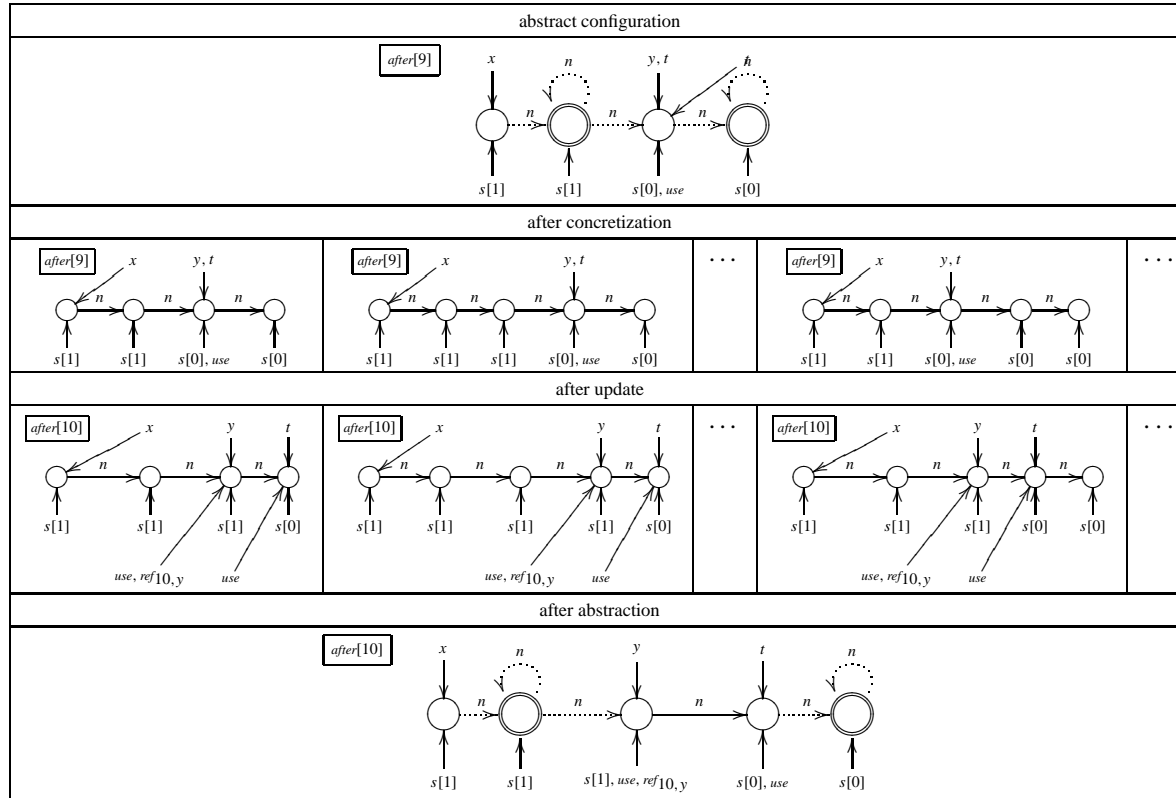
Fig. 5. Concretization, predicate-update including automaton transition updates, and abstraction for the statement `t = y.n` at line 10.

in Section 3.2. That is, the *use* attribute is set to 1 for the objects referenced by y and y.n, and the $ref_{10, y}$ attribute set to 1 for the object referenced by y. Then, $s[1]$ is set to 1 for the object referenced by y, and $s[0]$ is set to 0 for 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*. The implemented actions 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 [31]. 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 runtime 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 11** (*Assign-Null Property* $\langle pt, \mathtt{x}, \mathtt{f} \rangle$). **The assign-null property** $\langle pt, \mathtt{x}, \mathtt{f} \rangle$ **holds** if there exists no trace $\pi$ that includes a program state $\sigma_i = \langle \text{store}_i, pt \rangle$ such that the location denoted by $x.f$ in $\sigma_{i+1}$ is dynamically live in $\sigma_{i+1}$ in $\pi$.

The assign-null property allows us to assign null to a dead heap reference. In particular, when an assign-null property $\langle pt, \mathtt{x}, \mathtt{f} \rangle$ holds for a program point $pt$, a reference variable x and a reference field f, it guarantees that it is *safe* to issue a x.f = null statement immediately after $pt$. That is, it guarantees that adding such an x.f = null statement preserves the semantics of the original program (for a more formal treatment of semantic preserving transformations see [33]). As in the free property case, our assign-null property can also handle arbitrary reference expressions (e.g., of the form exp.f), by introducing a
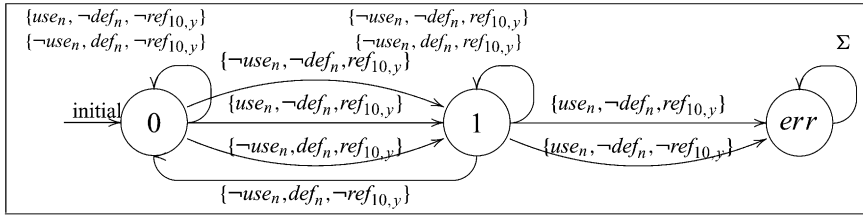
Fig. 6. A heap safety automaton $A^{an}_{10,\,y,\,n}$ for assign null to $y.n$ at 10.

```
[1]   Node root = CreateTree();
[2]   processTree(root.right);
      ... // no further uses of root
```

Fig. 7. A code snippet demonstrating the importance of assign-null analysis.

new program variable z, assigned with exp, and verifying the z.f may be issued just after the statement z = exp.

The potential for space savings beyond GC is demonstrated using the code snippet in Fig. 7. A tree of objects is allocated, but only the right side of the tree is processed. We assume that the type Node contains two instance fields: left and right. After line 1 all tree objects are reachable; thus GC cannot reclaim the entire left subtree of the root. However, it is easy to see that the assign-null property $\langle 1, \texttt{root}, \texttt{left}\rangle$ holds; thus it is safe to insert a root.left = null statement after line 1 allowing GC to collect the left side of the tree before the processing at line 2.

### 5.1.1. Assign-null property for the running example

We now demonstrate how an assign-null property is verified using our running example shown in 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 $\langle 10, \texttt{y}, \texttt{n}\rangle$ property. Our implementation verifies the assign-null $\langle 10, \texttt{y}, \texttt{n}\rangle$ 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 that 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. In particular, for a free property $\langle pt, \texttt{x}, \texttt{f}\rangle$, the corresponding HSA $A^{an}_{pt,\,x,\,f}$ may be obtained from the automaton in Fig. 6 by replacing 10 with $pt$, and by replacing $y$ with $x$, and $n$ with $f$.

As in the case for the free automaton (see Section 2), the alphabet of the assign-null automaton consists of sets of observable object attributes. For the purpose of verifying the assign-null property, we maintain the following object attributes in the instrumented semantics (see Section 3) for an object $o$: (i) $use_n$ attribute, which holds for $o$ if a reference

expression of the form x.f is used in the current statement execution and x references $o$; (ii) $def_n$, which holds for $o$ if a reference expression of the form x.f is defined in the current statement execution and x references $o$; (iii) $ref_{10,y}$ attribute, which holds for $o$ if the program execution is immediately after execution of the statement at line 10 and y references $o$ after the execution of the statement at line 10.

On the basis of the above object attributes we define the alphabet of the HSA $A_{10,y,n}^{an}$ to be $\Sigma = \{\{use_n, \neg def_n, \neg ref_{10,y}\}, \{use_n, \neg def_n, ref_{10,y}\}, \{\neg use_n, def_n, \neg ref_{10,y}\},$ $\{\neg use_n, def_n, ref_{10,y}\}, \{\neg use_n, \neg def_n, ref_{10,y}\}\}$.[2] For example, the alphabet symbol $\{use_n, \neg def_n, \neg ref_{10,y}\}$, denotes that the attribute $use_n$ holds for an object (i.e., the field $n$ of that object is used in the current statement), while the attribute $def_n$ does not hold for that object (i.e., the field $n$ of that object is not defined in the current statement), and also the attribute $ref_{10,y}$ does not hold for that object (i.e., either the current statement is not at 10, or this object is not referenced by y after the current statement is executed). Note that $use_n, def_n$ attributes cannot hold at the same time for an object, since we assume the code is normalized to a 3-address form, thus an object field cannot be used and defined in the same statement.

Initially, when an object $o$ is allocated it is assigned the initial state of $A_{10,y,n}^{an}$. Then, uses or definitions of the $n$ field of an object $o$ (a $use_n$ or a $def_n$ attribute holds for $o$, respectively) do not change the state of $A_{10,y,n}^{an}$ for $o$ (the self-loop in state 0 is taken). When the program is immediately after line 10 and y references an object $o$ ($ref_{10,y}$ attribute holds for $o$), $o$'s automaton state is set to 1. Now, if the $n$ field of $o$ is further defined (i.e., a $def_n$ attribute holds for $o$ in the subsequent program configurations along the execution path), and $o$'s automaton state is 1, the automaton state for $o$ gets back to the initial state (state 0). However, if the $n$ field of $o$ is used further (i.e., a $use_n$ attribute holds for $o$ in a subsequent program configuration along the execution path) before this field is redefined, and $o$'s automaton state is 1 the automaton state for $o$ reaches the *violation state* of the automaton. However, in the program of Fig. 1, the $n$-field references emanating from objects referenced by y at line 10 are not used further before being redefined; hence the property is not violated, and it is safe to add a y.n = null statement at this program point.

### 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 that $HSA_1, \ldots, HSA_k$ describe $k$ verification properties, then $k$ automaton states that $s_1, \ldots, s_k$ are maintained for every program object, where $s_i$ maintains an automaton state for $HSA_i$. Technically, as described in Section 3, a state $s_i$ is represented by automaton state predicates $s_i[q]$, where $q$ ranges over the states of $HSA_i$. The events associated with the automata $HSA_1, \ldots, HSA_k$ at a

---

[2] And in the alphabet for the free automaton, an equivalent way of writing the alphabet would be $\Sigma = \{\{use_n\}, \{use_n, ref_{10,y}\}, \{def_n\}, \{def_n, ref_{10,y}\}, \{ref_{10,y}\}\}$, where only attributes that hold for an object are shown.

program point are triggered simultaneously, updating the corresponding automaton state predicates of individuals.

The worst-case cost of simultaneous verification of properties is higher than the worst-case cost of verifying the same properties one by one. However, verifying properties one by one ignores the potential of computing overlapping heap information just once, where in simultaneous verification of properties this overlap is taken into consideration. Thus, we believe that in practice simultaneous verification of properties may achieve a lower cost than verifying the properties one by one. In fact, our initial findings in Section 6 show that verifying two properties one by one takes close to double the time it takes to verify these properties simultaneously. This is because most of the analysis time is spent on computing heap information.

Interestingly, if we limit our verification of free $\langle pt, \mathtt{x} \rangle$ properties to ones where $\mathtt{x}$ is used at $pt$ (i.e., $\mathtt{x}$ 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; this object cannot be freed earlier since $\mathtt{x}$ references the object, and a use of $\mathtt{x}$ occurs at $pt$; (ii) an object is freed "exactly once", i.e., there are no redundant frees of variables referencing the same object; this is immediate from the first feature, as an object is freed if and only if it is last referenced. 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 [36], showing an average of 15% potential space savings beyond a runtime 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 [36] we show an average of 39% potential space savings beyond a runtime 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 benchmark programs were used as a proof of concept. Due to scalability issues our benchmarks only provide a way to verify that our analysis is able to locate the static information at points of interest, and we do not measure the total savings. In particular the benchmarks provide three kinds of proofs of concept:

- We use small programs manipulating a linked-list to demonstrate the precision of our technique; moreover, we show that less precise analyses such as points-to analysis is insufficient for proving free and assign-null properties for these programs.
- We demonstrate how our techniques could be used to verify/automate manual space-saving rewritings. In particular, in our previous work [35] 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`.

• 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 [25]; (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 [38]. 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 [31]. For other programs, allocation-site information for heap objects suffices for the verification procedure.

In both abstractions interprocedural information is computed. In order to enable interprocedural analysis we explicitly represent stack frames and a corresponding set of predicates following [29]. Since this does not interfere with the material in this paper, to simplify presentation we do not describe these predicates.

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. 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 four programs involve manipulations of a singly linked list. DLoop, DPairs involve a doubly linked list manipulation. `small javac` is motivated by our previous work [35], 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 [22]. In `JavaPurse` a fixed set of loyalty stores is maintained, so every purchase grants loyalty points at the corresponding store. `GuessNumber` [28] 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.

Table 3
Analysis cost for the benchmark programs

| 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 |

Space is measured in MB, and time is measured in seconds.

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 runtime 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. For `Delete, small javac` and `JavaPurse` Table 3 shows the time and space cost for simultaneous verification of two properties. Later in this section we compare this cost to the time and space cost of verifying these properties one by one. For other benchmarks Table 3 shows the time and space cost for verifying a single property.

In `Loop` we verify our free $\langle 10, y \rangle$ and assign-null $\langle 10, y, n \rangle$ properties. For `CReverse` we verify that an element of the original list can be freed as soon it is copied to the reversed list. In `Delete` we show that 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 rewritten 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.

We have also tried our benchmarks using a points-to based heap abstraction, which is considered relatively cheap and scalable. We use a flow-sensitive, field-sensitive points-to analysis with unbounded context information [13]. Our results indicate that in all cases but one (assign-null properties for `JavaPurse` benchmark), points-to analysis is insufficient for proving the free and assign-null properties of interest. For `JavaPurse` the points-to analysis is able to prove the assign-null properties of interest since (i) we try to assign null to fields emanating from a singleton object, and (ii) field-sensitive information allows disambiguation of the fields emanating from the singleton object.

For `Delete`, `small javac` and `JavaPurse` we experiment with the simultaneous verification of properties. Table 3 shows the time and space cost for the simultaneous verification of two assign-null properties (*Assign Null* column for `Delete`, `small javac` and `JavaPurse`) and the time and space cost for the simultaneous verification of two free properties (*Free* column for `Delete` and `JavaPurse`). We compared the cost of simultaneous verification to the cost of verifying these properties one by one. Verifying the properties one by one takes close to double the time it takes to verify the same properties simultaneously. In addition, the space cost for verifying two properties simultaneously is close to the space cost of verifying a single property. This is because most of the analysis time (and space) is spent on computing heap information.

## 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 [12] 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 demonstrated in Section 7.1.

The Bandera project [9] uses the Bandera specification language (BSL) [10] to specify properties of software systems. Bandera constructs a finite state model of the program and uses existing model-checkers (e.g., SPIN [19]) to perform verification. BSL allows universally quantified specifications which are similar to our local heap safety properties. However, the abstractions currently applied by Bandera to verify these properties may generally lead to results that are less precise than ours.

The SLIC specification language [4] from MSR's SLAM project [26] is a low-level specification language which defines a (possibly infinite) state machine for tracking

temporal safety properties. Although SLIC is more powerful than our local heap safety properties (e.g., it allows counting), the abstraction applied by SLAM to verify SLIC properties may produce results that are less precise than ours.

In [40,39] a more general framework is presented for the specification and verification of properties of heap-manipulating programs using a first-order temporal logic named ETL (evolution temporal logic). The properties we addressed in this paper could be formulated as ETL formulae with a specific limited form — safety properties using a single universal quantifier. The algorithms presented in this paper efficiently handle the verification of properties in this subset.

Field et al. [14] investigate the problem of precise typestate checking in the presence of aliasing for shallow programs. They propose several abstraction techniques for precise typestate checking in such programs, and relate the cost of verification to the nature of the property being verified. In contrast, we handle arbitrary programs (not necessarily shallow) and arbitrary typestate properties, but do not guarantee precise results, and use more expensive techniques.

Some prior work used automata to dynamically monitor program execution and throw an exception when the property is violated (e.g.,[32,8]). 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 [20,30] (approximating flow dependencies) and [24] (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 [5,21,15,17,23]. 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 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 [6,37,2,16], the lifetime of an object is predicted at compile time. An object is associated with a memory region, and the allocation 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 [27] may be used in the context of a runtime 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 [3,1,18] liveness information for root references is used to reclaim more space.

In [36] 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 runtime garbage collector. We conclude there that heap liveness information yields a potential for space savings significantly larger than that

```
[1]   while (...) {
[2]     x = new T();
[3]     x.foo();
        ...
      }
```

Fig. 8. A code snippet demonstrating why a two-phase approach of a pointer analysis followed by an automaton state analysis leads to overconservative results.



Fig. 9. Two-phase analysis example.

achieved by communicating liveness information for stack and global variables. One way of communicating heap liveness information to a runtime GC is by assigning null to heap references. In this paper we present a static analysis algorithm for assigning null to heap references.

### 7.1. Advantages of integrated analysis

Our framework uses an integrated pointer and typestate analysis (called hereafter a *one-phase approach*). As discussed above, in [12,4,9] a pointer analysis is applied as a preliminary phase, followed by a phase of automaton state analysis via finite state verification (called hereafter a *two-phase approach*). Generally, it is well known that the analysis of combined abstract domains (e.g., our one-phase approach) is more precise than the combination of separate analyses of abstract domains (e.g., the two-phase approach) [11]. In particular, in this section we demonstrate that even when applying a more limited points-to analysis (in contrast to the shape analysis used in earlier sections) to a simple program it may be profitable to use an integrated analysis.

Consider the code snippet in Fig. 8. A new object is allocated and used in every loop iteration. Fig. 9 shows the result of applying the two-phase approach for the purpose of verifying the free property $\langle 3, x \rangle$. We omit the information for line 1, since no automaton events are triggered at line 1. For the pointer analysis phase we assume an allocation-site based abstract domain used in points-to algorithms, e.g. [13]. The column *Pointer Analysis Phase* shows the results of the pointer analysis. The predicate *site*[2] holds for individuals allocated at line 2. We see that at both line 1 and line 2, x may reference objects allocated at line 2.

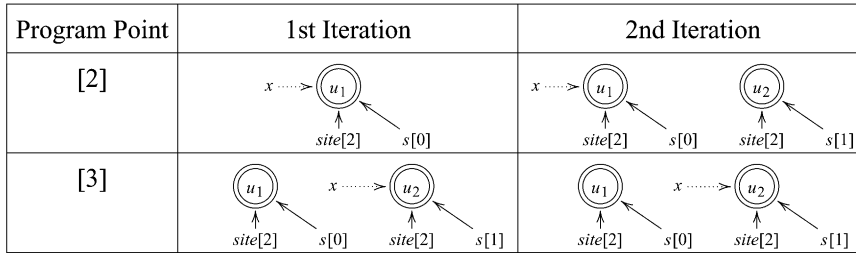| Program Point | 1st Iteration | 2nd Iteration |
|---|---|---|
| [2] | | |
| [3] | | |

Fig. 10. One-phase analysis example.

In the finite verification phase, we start with the pointer information at line 2 and initialize the automaton state of $u_1$ to $s[0]$ due to the allocation in 2. Then, at line 3, we need to trigger the automaton event $\{use, ref_{3,x}\}$ for the objects that may be referenced by x (i.e., the objects represented by $u_1$). For objects referenced by x the automaton state is changed to 1 (due to the $\{use, ref_{3,x}\}$ edge connecting states 0 and 1 in the $A_{3,x}^{free}$ automaton). However, not all the objects represented by $u_1$ are necessarily referenced by x, and for those their automaton state 0 is unchanged. Thus, we conclude that the objects represented by $u_1$ after line 3 may either be in state 0 or in state 1, as shown by the dashed edges emanating from $s[0]$, $s[1]$. Next, in the second verification iteration, the allocation at line 2 does not change the possible automaton states for $u_1$. Finally, we consider again the effect of line 3. We again trigger the automaton event $\{use, ref_{3,x}\}$, and conclude that the objects referenced by x may reach the *err* state (since for objects in state 1 the $\{use, ref_{3,x}\}$ edge leading to *err* state is taken), leading to an overconservative result, i.e., we fail to validate that `free(x)` can be safely inserted after line 3.

We now show how the free property $\langle 3, x \rangle$ is successfully verified using a one-phase approach. Fig. 10 shows how the analysis works. Again, we omit the information for line 1. Here an allocation site based abstract domain is used, refined with the automaton state. Thus, objects allocated at the same allocation site, but in different automaton state are abstracted to different elements in the abstract domain. First, at line 2, x may reference objects allocated at line 2 that are in automaton state 0 (the latter fact is represented by the solid edge from $s[0]$ to $u_1$). Then, at line 3, we need to trigger the automaton event $\{use, ref_{3,x}\}$ for the objects that may be referenced by x. Thus, after triggering these events x may only reference objects allocated at line 2 in automaton state 1 (these objects are represented by the individual $u_2$). In addition, there may be objects allocated at line 2 in automaton state 0. These latter objects may not be referenced by x and are represented by the individual $u_1$. Next, in the second iteration at line 2, x may reference objects allocated at 2 in automaton state 0. This is due to the allocation in 2. Finally, the second iteration for line 3 yields the same structure as in the previous iteration for line 3, and the third iteration for the structure at line 2 (not shown in Fig. 10) yields the same structure as in the previous iteration; thus we conclude that the *err* may not be reached, and therefore the free property $\langle 3, x \rangle$ is verified, and it is safe to insert `free(x)` after line 3.

## 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 also be interesting to explore opportunities for deallocating space using richer constructs than `free(exp)`. For example, using a new `free-list` construct for deallocating an entire list.

## References

[1] O. Agesen, D. Detlefs, E. Moss, Garbage collection and local variable type-precision and liveness in Java virtual machines, in: SIGPLAN Conf. on Prog. Lang. Design and Impl., June 1998, ACM Press, 1998, pp. 269–279.

[2] A. Aiken, M. Fahndrich, R. Levien, Better static memory management: Improving region-based analysis of higher-order languages, in: SIGPLAN Conf. on Prog. Lang. Design and Impl., June 1995, ACM Press, 1995, pp. 174–185.

[3] A.W. Appel, Compiling with Continuations, CUP, 1992, pp. 205–214 (Chapter 16).

[4] T. Ball, 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) (1977) 513–518.

[6] L. Birkedal, M. Tofte, M. Vejlstrup, From region inference to von Neumann machines via region representation inference, in: Symp. on Princ. of Prog. Lang., ACM Press, 1996, pp. 171–183.

[7] B. Blanchet, Escape analysis for object oriented languages. application to Java[tm], in: Conf. on Object-Oriented Prog. Syst., Lang. and Appl., ACM Press, 1998, pp. 20–34.

[8] T. Colcombet, P. Fradet, Enforcing trace properties by program transformation, in: Symp. on Princ. of Prog. Lang., January 2000, ACM Press, 2000, pp. 54–66.

[9] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, R. Shawn, L. Hongjun, Bandera: Extracting finite-state models from Java source code, in: Int. Conf. on Soft. Eng., June 2000, ACM Press, 2000, pp. 439–448.

[10] J.C. Corbett, M.B. Dwyer, J. Hatcliff, Robby, A language framework for expressing checkable properties of dynamic software, in: Int. Spin Workshop on Model Check. of Soft., in: Lec. Notes in Comp. Sci., vol. 1885, Springer-Verlag, 2000, pp. 205–223.

[11] P. Cousot, R. Cousot, Systematic design of program analysis frameworks, in: Symp. on Princ. of Prog. Lang., ACM Press, 1979, pp. 269–282.

[12] M. Das, S. Lerner, M. Seigle, ESP: Path-sensitive program verification in polynomial time, in: SIGPLAN Conf. on Prog. Lang. Design and Impl., June 2002, ACM Press, 2002, pp. 57–68.

[13] M. Emami, R. Ghiya, L. Hendren, Context-sensitive interprocedural points-to analysis in the presence of function pointers, in: SIGPLAN Conf. on Prog. Lang. Design and Impl., ACM Press, 1994, pp. 242–256.

[14] J. Field, D. Goyal, G. Ramalingam, E. Yahav, Typestate verification: Abstraction techniques and complexity results, in: Proc. of SAS'03, June 2003, in: LNCS, vol. 2694, Springer, 2003, pp. 439–462.

[15] I. Foster, W. Winsborough, Copy avoidance through compile-time analysis and local reuse, in: Proceedings of International Logic Programming Symposium, MIT Press, 1991, pp. 455–469.

[16] N. Hallenberg, M. Elsman, M. Tofte, Combining region inference and garbage collection, in: SIGPLAN Conf. on Prog. Lang. Design and Impl., ACM Press, 2002, pp. 141–152.

[17] G.W. Hamilton, Compile-time garbage collection for lazy functional languages, in: Memory Management, International Workshop IWMM 95, in: Lec. Notes in Comp. Sci., vol. 637, Springer-Verlag, 1995.

[18] M. Hirzel, A. Diwan, A.L. Hosking, On the usefulness of type and liveness accuracy for garbage collection and leak detection, Trans. Prog. Lang. Syst. 24 (6) (2002) 593–624.

[19] G. Holzmann, The model checker SPIN, IEEE Trans. Softw. Eng. 23 (5) (1997) 279–294.

[20] S. Horwitz, P. Pfeiffer, T. Reps, Dependence analysis for pointer variables, in: SIGPLAN Conf. on Prog. Lang. Design and Impl., ACM Press, 1989, pp. 28–40.

[21] K. Inoue, H. Seki, H. Yagi, Analysis of functional programs to detect run-time garbage cells, Trans. on Prog. Lang. and Syst. 10 (4) (1988) 555–578.

[22] Java card 2.2 development kit. Available at http://java.sun.com/products/javacard.

[23] R. Jones, Garbage Collection. Algorithms for Automatic Dynamic Memory Management, John Wiley and Sons, 1999.

[24] T. Lev-Ami, T.W. Reps, R. Wilhelm, M. Sagiv, Putting static analysis to work for verification: A case study, in: Int. Symp. on Soft. Testing and Anal., ACM Press, 2000, pp. 26–38.

[25] T. Lev-Ami, M. Sagiv, TVLA: A system for implementing static analyses, in: Static Analysis Symp., in: Lec. Notes in Comp. Sci., vol. 1824, Springer-Verlag, 2000, pp. 280–301.

[26] Microsoft Research, The SLAM project, http://research.microsoft.com/slam/, 2001.

[27] S. Muchnick, Advanced Compiler Design and Implementation, Morgan Kaufmann, 1997.

[28] Oberthur card systems, http://www.oberthurcs.com.

[29] N. Rinetzky, M. Sagiv, Interprocedural shape analysis for recursive programs, in: Int. Conf. on Comp. Construct., in: Lec. Notes in Comp. Sci., vol. 2027, Springer-Verlag, 2001, pp. 133–149.

[30] J. Ross, M. Sagiv, Building a bridge between pointer aliases and program dependences, in: European Symp. on Prog., March 1998, in: Lec. Notes in Comp. Sci., vol. 1381, Springer-Verlag, 1998, pp. 221–235. Available at http://www.math.tau.ac.il/~sagiv.

[31] M. Sagiv, T. Reps, R. Wilhelm, Parametric shape analysis via 3-valued logic, Trans. Prog. Lang. and Syst. 24 (3) (2002) 217–298.

[32] F. Schneider, Enforceable security policies, ACM Trans. Inform. Syst. Security 3 (1) (2000) 30–50.

[33] R. Shaham, Heap-liveness-based memory management: Potential, tools, and algorithms, Ph.D. Thesis, Tel Aviv University, 2004.

[34] R. Shaham, E.K. Kolodner, M. Sagiv, Automatic removal of array memory leaks in Java, in: Int. Conf. on Comp. Construct., April 2000, in: Lec. Notes in Comp. Sci., vol. 1781, Springer-Verlag, 2000, pp. 50–66.

[35] R. Shaham, E.K. Kolodner, M. Sagiv, Heap profiling for space-efficient Java, in: SIGPLAN Conf. on Prog. Lang. Design and Impl., ACM Press, 2001, pp. 104–113.

[36] R. Shaham, E.K. Kolodner, M. Sagiv, Estimating the impact of heap liveness information on space consumption in Java, in: Int. Symp. on Memory Management, June 2002, ACM, 2002, pp. 171–182.

[37] M. Tofte, J.-P. Talpin, Implementation of the typed call-by-value lambda-calculus using a stack of regions, in: Symp. on Princ. of Prog. Lang., January 1994, ACM Press, 1994, pp. 188–201.

[38] R. Vallée-Rai, L. Hendren, V. Sundaresan, E.G.P. Lam, P. Co. Soot—a Java optimization framework, in: Proceedings of CASCON 1999, 1999, pp. 125–135.

[39] E. Yahav, A. Pnueli, T. Reps, M. Sagiv, Efficient verification of temporal heap properties, Technical Report 339/04, Tel Aviv University, December 2003.

[40] E. Yahav, T. Reps, M. Sagiv, R. Wilhelm, Verifying temporal heap properties specified via evolution logic, in: Proc. of the 12th European Symposium on Programming, ESOP 2003, April 2003, in: LNCS, vol. 2618, 2003.