Memory Usage Verification Using Hip/Sleek

Guanhua He[1], Shengchao Qin[1], Chenguang Luo[1], and Wei-Ngan Chin[2]

[1] Durham University, Durham DH1 3LE, UK
[2] National University of Singapore
{guanhua.he,shengchao.qin,chenguang.luo}@durham.ac.uk,
chinwn@comp.nus.edu.sg

Abstract. Embedded systems often come with constrained memory footprints. It is therefore essential to ensure that software running on such platforms fulfils memory usage specifications at compile-time, to prevent memory-related software failure after deployment. Previous proposals on memory usage verification are not satisfactory as they usually can only handle restricted subsets of programs, especially when shared mutable data structures are involved. In this paper, we propose a simple but novel solution. We instrument programs with explicit memory operations so that memory usage verification can be done along with the verification of other properties, using an automated verification system Hip/Sleek developed recently by Chin et al. [10,19]. The instrumentation can be done automatically and is proven sound with respect to an underlying semantics. One immediate benefit is that we do not need to develop from scratch a specific system for memory usage verification. Another benefit is that we can verify more programs, especially those involving shared mutable data structures, which previous systems failed to handle, as evidenced by our experimental results.

1 Introduction

Ubiquitous embedded systems are often supplied with limited memory and computation resources due to various constraints on, e.g., product size, power consumption and manufacture cost. The consequences of violating memory safety requirements can be quite severe because of the close coupling of these systems with the physical world; in some cases, they can put human lives at risk. The Mars Rover's anomaly problem was actually due to a memory leak error and it took fifteen days to fix the problem and bring the Rover back to normal [21]. For applications running on resource-constrained platforms, a challenging problem would be how to make memory usage more predictable and how to ensure that memory usage fulfils the restricted memory requirements.

To tackle this challenge, a number of proposals have been reported on memory usage analysis and verification, with most of them focused on functional programs where data structures are mostly immutable and thus easier to handle [1,2,5,7,15,23]. Memory usage verification for imperative/OO languages can be more challenging due to mutability of states and object sharing. Existing solutions to this are mainly type-based [11,12,16]. Instead of capturing all aliasing

Z. Liu and A.P. Ravn (Eds.): ATVA 2009, LNCS 5799, pp. 166–181, 2009.

information, they impose restrictions on object mutability and sharing. Therefore, they can only handle limited subsets of programs manipulating shared mutable data structures.

The emergence of separation logic [17,22] promotes scalable reasoning via explicit separation of structural properties over the memory heap where recursive data structures are dynamically allocated. Since then, dramatic advances have been made in automated software verification via separation logic, e.g. the Smallfoot tool [3] and the Space Invader tool [6,13,24] for the analysis and verification on pointer safety (i.e. shape properties asserting that pointers cannot go wrong), the Hip/Sleek tool [10,18,19] for the verification of more general properties involving both structural (shape) and numerical (size) information, the verification on termination [4], and the verification for object-oriented programs [9,14,20].

Given these significant advances in the field, a research question that we post to ourselves is: can we make use of some of these state-of-the-art verification tools to do a better job for memory usage verification, without the need of constructing a memory usage verifier from scratch? This paper addresses this question by proposing a simple but novel mechanism to memory usage verification using the Hip/Sleek system developed by Chin et al. [10,19]. Separation logic offers a powerful and expressive mechanism to capture structural properties of shared mutable data structures including aliasing information. The specification mechanism in Hip/Sleek leverages structural properties with numerical information and is readily capable for the use of memory usage specification.

Approach and contributions. Memory usage occur in both the heap and stack spaces. While heap space is used to store dynamically allocated data structures, stack memory is used for local variables as well as return addresses of method calls. On the specification side, we assume that two special global variables `heap` and `stk` of type `int` are reserved to represent respectively the available heap and stack memory in the pre-/post-conditions of each method. On the program side, we *instrument* the program to be verified with explicit operations over variables `heap` and `stk` using rewriting rules. We call the instrumented programs as *memory-aware programs*. The memory usage behaviour of the original program is now mimicked and made explicit in its memory-aware version via the newly introduced primitive operations over `heap` and `stk`. We also show that the original program and its memory-aware version are observationally equivalent modulo the behaviour of the latter on the special variables `heap` and `stk` as well as a fixed memory cost for storing the two global variables. Instead of constructing and implementing a fresh set of memory usage verification rules for the original program, we can now pass to Hip/Sleek as inputs the corresponding memory-aware program together with the expected memory specification for automated memory usage verification.

In summary, this paper makes the following contributions:

- We propose a simple but novel solution to memory usage verification based on a verification tool Hip/Sleek by first rewriting programs to their memory-aware counterparts.

- We demonstrate that the syntax-directed rewriting process is sound in the sense that the memory-aware programs are observationally equivalent to their original programs with respect to an instrumented operational semantics.
- We have integrated our solution with HIP/SLEEK and conducted some initial experiments. The experimental results confirm the viability of our solution and show that we can verify the memory safety of more programs compared with previous type-based approaches.

The rest of the paper is structured as follows. We introduce our programming and specification languages in Section 2. In Section 3 we present our approach to memory usage verification in HIP/SLEEK. Section 4 defines an underlying semantics for the programming language and formulates the soundness of our approach w.r.t. the semantics. Experimental results are shown in Section 5, followed by related work and concluding remarks afterwards.

2 Language and Specifications

In this section, we first introduce a core imperative language we use to demonstrate the work, and then depict the general specification mechanism used by HIP/SLEEK and show how memory usage specifications can be incorporated in.

2.1 Programming Language

To simplify presentation, we focus on a strongly-typed C-like imperative language in Figure 1.

A program P in our language consists of user-defined data types *tdecl*, global variables *gVar* and method definitions *meth*. The notation *datat* stands for the standard data type declaration used in programs, for example as below:

```
data node { int val; node next }
data node2 { int val; node2 prev; node2 next }
data node3 { int val; node3 left; node3 right; node3 parent }
```

The notation *spred* denotes a user-defined predicate which may be recursively defined and can specify both structural and numerical properties of data structures involved. The syntax of *spred* is given in Figure 2.

$$
\begin{array}{lll}
P & ::= tdecl^* \ gVar^* \ meth^* & tdecl ::= datat \mid spred \\
datat & ::= \texttt{data}\ c\ \{\ field^*\ \} & field ::= t\ v \qquad t ::= c \mid \tau \\
\tau & ::= \texttt{int} \mid \texttt{bool} \mid \texttt{void} & gVar ::= t\ v \\
meth & ::= t\ mn\ ((\texttt{[ref]}\ t\ v)^*)\ mspec\ \{e\} & \\
e & ::= \texttt{null} \mid k^{\tau} \mid v \mid v.f \mid v{:=}e \mid v_1.f{:=}v_2 \mid \texttt{new}\ c(v^*) \mid \texttt{free}(v) & \\
 & \quad \mid e_1; e_2 \mid t\ v;\ e \mid mn(v^*) \mid \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 &
\end{array}
$$

Fig. 1. A Core (C-like) Imperative Language

Note that a parameter can be either pass-by-value or pass-by-reference, distinguished by the `ref` before a parameter definition. The method specification *mspec*, written in our specification language in Figure 2, specifies the expected behaviour of the method, including its memory usage behaviour. Our aim is to verify the method body against this specification. Our language is expression-oriented, so the body of a method is an expression composed of standard instructions and constructors of an imperative language. Note that the instructions `new` and `free` explicitly deal with memory allocation and deallocation, respectively. The term k^{τ} denotes a constant value of type τ. While loops are transformed to tail-recursive methods in a preprocessing step.

2.2 Specification Language

Our specification language is given in Figure 2. Note *spred* defines a new separation predicate c in terms of the formula Φ with a given pure invariant π. Such user-specified predicates can be used in the method specifications. The method specification *requires* Φ_{pr} *ensures* Φ_{po} comprises a precondition Φ_{pr} and a postcondition Φ_{po}.

The separation formula Φ, which appears in the predicate definition *spred* or in the pre-/post-conditions of a method, is in disjunctive normal form. Each disjunct consists of a $*$-separated heap constraint κ, referred to as *heap part*, and a heap-independent formula π, referred to as *pure part*. The pure part does not contain any heap nodes and is restricted to pointer equality/disequality γ and Presburger arithmetic ϕ. As we will see later, γ is used to capture the alias information of pointers during the verification, and ϕ is to record the numerical information of data structures, such as length of a list or height of a tree. Furthermore, Δ denotes a composite formula that could always be normalized into the Φ form [19].

The formula `emp` represents an empty heap. If `c` is a data node, the formula $\mathtt{p}{::}\mathtt{c}\langle \mathtt{v}^* \rangle$ represents a singleton heap $\mathtt{p} \mapsto [(\mathtt{f} : \mathtt{v})^*]$ with $\mathtt{f}^*$ as fields of data declaration `c`. For example, $\mathtt{p}{::}\mathtt{node}\langle 0, \mathtt{null} \rangle$ denotes that `p` points to a `node` structure in the heap, whose fields have values 0 and `null`, respectively. If `c` is a (user-specified) predicate, $\mathtt{p}{::}\mathtt{c}\langle \mathtt{v}^* \rangle$ stands for the formula $\mathtt{c}(\mathtt{p}, \mathtt{v}^*)$ which signifies that

$$
\begin{array}{lll}
\mathit{spred} & ::= & \mathtt{root}{::}c\langle v^* \rangle \equiv \Phi \ \mathtt{inv}\ \pi \\
\mathit{mspec} & ::= & \mathit{requires}\ \Phi_{pr}\ \mathit{ensures}\ \Phi_{po} \\
\Phi & ::= & \bigvee (\exists v^* \cdot \kappa \wedge \pi)^* \qquad\qquad \pi ::= \gamma \wedge \phi \\
\gamma & ::= & v_1 = v_2 \mid v = \mathtt{null} \mid v_1 \neq v_2 \mid v \neq \mathtt{null} \mid \gamma_1 \wedge \gamma_2 \\
\kappa & ::= & \mathtt{emp} \mid v{::}c\langle v^* \rangle \mid \kappa_1 * \kappa_2 \\
\Delta & ::= & \Phi \mid \Delta_1 \vee \Delta_2 \mid \Delta \wedge \pi \mid \Delta_1 * \Delta_2 \mid \exists v \cdot \Delta \\
\phi & ::= & b \mid a \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg \phi \mid \exists v \cdot \phi \mid \forall v \cdot \phi \\
b & ::= & \mathtt{true} \mid \mathtt{false} \mid v \mid b_1 = b_2 \quad a ::= s_1 = s_2 \mid s_1 \leq s_2 \\
s & ::= & k^{\mathtt{int}} \mid v \mid k^{\mathtt{int}} \times s \mid s_1 + s_2 \mid -s \mid \mathit{max}(s_1, s_2) \mid \mathit{min}(s_1, s_2)
\end{array}
$$

Fig. 2. The Specification Language

the data structure pointed to by p has the shape c with parameters v*. As an example, one may define the following predicate for a singly linked list with length n:

$$\texttt{root::ll}\langle\texttt{n}\rangle\equiv(\texttt{root=null}\wedge\texttt{n=0})\vee(\exists\texttt{i},\texttt{m},\texttt{q}\cdot\texttt{root::node}\langle\texttt{i},\texttt{q}\rangle*\texttt{q::ll}\langle\texttt{m}\rangle\wedge\texttt{n=m+1})\ \textbf{inv}\ \texttt{n}\geq\texttt{0}$$

The above definition asserts that an ll list either can be empty (the base case root=null where root is the "head pointer" pointing to the beginning of the whole structure described by ll), or consists of a head data node (specified by root::node⟨i, q⟩) and a separate tail data structure which is also an ll list (q::ll⟨m⟩ saying that q points to an ll list with length m). The separation conjunction ∗ introduced in separation logic signifies that two heap portions are domain-disjoint. Therefore, in the inductive case of ll's definition, the separation conjunction ensures that the head node and the tail ll reside in disjoint heaps. A default invariant n≥0 is specified which holds for all ll lists. Existential quantifiers are for local values and pointers in the predicate, such as i, m and q.

A slightly more complicated shape, a doubly linked-list with length n, is described by:

$$\texttt{root::dll}\langle\texttt{p},\texttt{n}\rangle\equiv(\texttt{root=null}\wedge\texttt{n=0})\vee(\texttt{root::node2}\langle\_,\texttt{p},\texttt{q}\rangle*\texttt{q::dll}\langle\texttt{root},\texttt{n}-\texttt{1}\rangle)\ \textbf{inv}\ \texttt{n}\geq\texttt{0}$$

The dll predicate has a parameter p to represent the prev field of the root node of the doubly linked list. This shape includes node root and all the nodes reachable through the next field starting from root, but not the ones reachable through prev from root. Here we also can see some shortcuts that underscore _ denotes an anonymous variable, and non-parameter variables in the right hand side of the shape definition, such as q, are implicitly existentially quantified.

As can be seen from the above, we can use κ to express the shape of heap and ϕ to express numerical information of data structures, such as length. This allows us to specify data structures with sophisticated invariants. For example, we may define a non-empty sorted list as below:

$$\begin{aligned}&\texttt{root::sortl}\langle\texttt{n},\texttt{min}\rangle\equiv(\texttt{root::node}\langle\texttt{min},\texttt{null}\rangle\wedge\texttt{n=1}\ \vee\\&\qquad(\texttt{root::node}\langle\texttt{min},\texttt{q}\rangle*\texttt{q::sortl}\langle\texttt{m},\texttt{k}\rangle\wedge\texttt{n=m+1}\wedge\texttt{min}\leq\texttt{k})\ \ \textbf{inv}\ \texttt{n}\geq\texttt{0}\end{aligned}$$

The sortedness property is captured with the help of an additional parameter min denoting the minimum value stored in the list. The formula min≤k ensures the sortedness. With the aforesaid predicates, we can now specify the insertion-sort algorithm as follows:

```
node insert(node x, node vn)
  requires x::sortl⟨n, min⟩ ∗ vn::node⟨v, _⟩
  ensures  res::sortl⟨n+1, min(v, min)⟩;
{···}
```

```
node insertion_sort(node y)
  requires  y::ll⟨n⟩ ∧ n>0
  ensures   res::sortl⟨n, _⟩;
{···}
```

where a special identifier res is used in the postcondition to denote the result of a method. The postcondition of insertion_sort shows that the output list is sorted and has the same number of nodes. We can also specify that the input

and output lists contain the same set of values by adding another parameter to the `sortl` predicate to capture the bag of values stored in the list [10].

The semantics of our specification formula is similar to the model given for separation logic [22] except that we have extensions to handle user-defined shape predicates. We assume sets Loc of memory locations, Val of primitive values, with $0 \in \mathsf{Val}$ denoting `null`, Var of variables (program and logical variables), and ObjVal of object values stored in the heap, with $c[f_1 \mapsto \nu_1, .., f_n \mapsto \nu_n]$ denoting an object value of data type c where $\nu_1, .., \nu_n$ are current values of the corresponding fields $f_1, .., f_n$. Let $s, h \models \Phi$ denote the model relation, i.e. the stack s and heap h satisfy Φ, with h, s from the following concrete domains:

$$h \in \mathit{Heaps} =_{df} \mathsf{Loc} \rightharpoonup_{fin} \mathsf{ObjVal} \qquad s \in \mathit{Stacks} =_{df} \mathsf{Var} \rightarrow \mathsf{Val} \cup \mathsf{Loc}$$

Note that each heap h is a finite partial mapping while each stack s is a total mapping, as in the classical separation logic [17,22]. The detailed definitions of the model relation can be found in Chin et al. [10].

2.3 Memory Usage Specification

To incorporate memory usage into the specification mechanism of Hip/Sleek, we employ two global variables `heap` and `stk` to represent the available heap and stack memory (in bytes). The memory requirement of a method can then be specified as a pure constraint over `heap` and `stk` in the precondition of the method. The remaining memory space upon the return from a method call can also be exhibited using a pure formula over $\texttt{heap}'$ and $\texttt{stk}'$ in the postcondition.[1] Due to perfect recovery of stack space upon return from a method call, $\texttt{stk}'$ in a method's postcondition will always be the same as its initial value `stk`. As an example, the method `new_list(int n)`, which creates a singly linked list with length `n`, is given as follows together with its memory usage specification:

```
node new_list(int n)
  requires  heap≥8 * n ∧ n≥0 ∧ stk≥12 * n+4
  ensures   res::ll⟨n⟩ ∧ heap′=heap−8 * n ∧ stk′=stk
{ node r := null; if (n>0) { r := new_list(n−1); r := new node(n, r)}; r }
```

where the `node` was declared earlier in Sec 2.1. We assume that we use a `32-bit` architecture; therefore, one `node` requires 8 bytes of memory. This assumption can be easily changed for a different architecture. The precondition specifies that the method requires at least $8 * \texttt{n}$ bytes of heap space and $12 * \texttt{n} + 4$ stack space before each execution with `n` denoting the size of the input.[2] After method

[1] A primed variable $\texttt{x}'$ in a specification formula denotes the latest value of variable `x`, with `x` representing its initial value.

[2] When a new local variable `r` is declared, 4 bytes of stack memory is consumed. Later when the method `new_list` is invoked recursively, its parameters, return address and local variables are all placed on top of the stack. This is why it requires at least $12 * \texttt{n}+4$ bytes of stack space.

execution, $8 * \mathtt{n}$ bytes of heap memory is consumed by the returned list, but the stack space is fully recovered. This is reflected by the formula ($\mathtt{heap}' = \mathtt{heap} - 8 * \mathtt{n} \wedge \mathtt{stk}' = \mathtt{stk}$) in the postcondition.

As another example, the following method `free_list` deallocates a list:

```
void free_list(node2 x)
  requires  x::dll⟨n⟩ ∧ heap≥0 ∧ stk≥12 * n
  ensures   emp ∧ heap'=heap+12 * n ∧ stk'=stk
{ if (x ≠ null) { node t := x; x := x.next; free(t); free_list(x) } }
```

We can see that $12 * \mathtt{n}$ bytes of heap space is expected to be claimed back by the method as signified in the postcondition. Notice here the stack and heap memory are specified in terms of the logical variable `n` denoting the length of the list `x`, showing the possible close relation between the separation (shape and size) specification and the memory specification. Next we will show how to rewrite the program to its memory-aware version by using the two global variables `heap` and `stk` to mimic the memory behaviour, so that Hip/Sleek can step in for memory usage verification.

3 Memory Usage Verification

In this section, we first present the instrumentation process which converts programs to be verified to memory-aware programs. We then briefly introduce the automated verification process in Hip/Sleek.

3.1 The Instrumentation Process

The instrumentation process makes use of primitive operations over the global variables `heap` and `stk` to simulate the memory usage behaviour of the original program. It is conducted via the rewriting rules given in Figure 3.

These rewriting rules form a transformer $\mathcal{M}$ which takes in a program and returns its memory-aware version. Note that $\mathcal{M}$ conducts identical rewriting except for the following four cases: (1) heap allocation $\mathtt{new}\ c(v^*)$; (2) heap deallocation $\mathtt{free}(\mathtt{v})$; (3) local block $\{\mathtt{t\ v;\ e}\}$; (4) method declaration $\mathtt{t_0\ mn(t_1\ v_1, .., t_n\ v_n)\{e\}}$.

$$
\begin{array}{lll}
\mathcal{M}(E) & ::= & E \text{ where } E \in \{\mathtt{null},\ k^{\tau},\ v,\ v.f,\ v_1.f{:=}v_2,\ mn(v^*)\} \\
\mathcal{M}(\mathtt{new}\ c(v^*)) & ::= & \mathit{dec\_hp}(\mathit{ssizeof}(c));\ \mathtt{new}\ c(v^*) \\
\mathcal{M}(\mathtt{free}(v)) & ::= & \mathtt{free}(v);\ \mathit{inc\_hp}(\mathit{ssizeof}(\mathit{type}(v))) \\
\mathcal{M}(\{t\ v; e\}) & ::= & \mathit{dec\_stk}(\mathit{sizeof}(t));\ \{t\ v;\ \mathcal{M}(e)\};\ \mathit{inc\_stk}(\mathit{sizeof}(t)) \\
\mathcal{M}(v{:=}e) & ::= & v{:=}\mathcal{M}(e) \\
\mathcal{M}(e_1; e_2) & ::= & \mathcal{M}(e_1); \mathcal{M}(e_2) \\
\mathcal{M}(\mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2) & ::= & \mathtt{if}\ v\ \mathtt{then}\ \mathcal{M}(e_1)\ \mathtt{else}\ \mathcal{M}(e_2) \\
\mathcal{M}(t_0\ mn(t_1\ v_1, .., t_n\ v_n)\{e\}) & ::= & t_0\ mn(t_1\ v_1, .., t_n\ v_n)\{ \\
\multicolumn{3}{l}{\quad \mathit{dec\_stk}(\mathit{sizeof}(t_0, t_1, .., t_n){+}4); \mathcal{M}(e); \mathit{inc\_stk}(\mathit{sizeof}(t_0, t_1, .., t_n){+}4)\}}
\end{array}
$$

Fig. 3. Rewriting Rules for Instrumentation

To simulate the memory effect of `new` $c(v^*)$, we employ a primitive method over variable `heap`, called *dec_hp*, which is subject to the specification:

$$\texttt{void}\ \ \textit{dec\_hp}(\texttt{int}\ n)\ \ \textit{requires}\ \ \texttt{heap}{\geq}n \wedge n{\geq}0\ \ \textit{ensures}\ \ \texttt{heap}'{=}\texttt{heap}{-}n$$

To successfully call *dec_hp*(`n`), the variable `heap` must hold a value no less than the non-negative integer `n` at the call site. Upon return, the value of `heap` is decreased by `n`.

To simulate the memory effect of `free`(`v`), we employ a primitive method over `heap`, called *inc_hp*:

$$\texttt{void}\ \ \textit{inc\_hp}(\texttt{int}\ n)\ \ \textit{requires}\ \ n{\geq}0\ \ \textit{ensures}\ \ \texttt{heap}'{=}\texttt{heap}{+}n$$

The memory effect of local blocks and method bodies can be simulated in a similar way, and the difference is that they count on the stack instead of heap. For code blocks, we employ *dec_stk* to check the stack space is sufficient for the local variable to be declared, and decrease the stack space; meanwhile, at the end of the block, we recover such space by *inc_stk* due to the popping out of the local variables. As for method body, stack space is initially acquired (and later recovered) for method parameters and return address (four bytes), as the last rewriting rule suggests. The specifications for these two primitive methods are as follows:

$$\texttt{void}\ \ \textit{dec\_stk}(\texttt{int}\ n)\ \ \textit{requires}\ \ \texttt{stk}{\geq}n \wedge n{\geq}0\ \ \textit{ensures}\ \ \texttt{stk}'{=}\texttt{stk}{-}n$$
$$\texttt{void}\ \ \textit{inc\_stk}(\texttt{int}\ n)\ \ \textit{requires}\ \ n{\geq}0\ \ \textit{ensures}\ \ \texttt{stk}'{=}\texttt{stk}{+}n$$

Note that two different functions *sizeof* and *ssizeof* are used in the rewriting rules: *sizeof* is applied to both primitive and reference types, while *ssizeof* is applied to (user-defined) data types, by summing up the sizes of all declared fields' types obtained via *sizeof*. For example, *sizeof*(`int`) = 4, *sizeof*(`node`) = 4, and *ssizeof*(`node`) = 8, since the `node` data structure (defined in Section 2) contains an `int` field and a reference to another `node`. We also abuse these functions by applying them to a list of types, expecting them to return the sum of the results when applied to each type.

We present below the memory-aware versions for the two examples given in Section 2.

```
node new_list(int n)
  requires emp ∧ heap≥8 * n∧
           n≥0 ∧ stk≥12 * n + 4
  ensures  res::ll⟨n⟩ ∧ stk′=stk∧
           heap′=heap−8 * n;
{ dec_stk(4);
  node r := null;
  if (n > 0) {
    dec_stk(8); r := new_list(n−1);
    inc_stk(8); dec_hp(8);
    r := new node(n, r) };
  inc_stk(4); r }
```

Fig. 4. Example 1

```
void free_list(node2 x)
  requires x::dll⟨p, n⟩ ∧ heap≥0∧
           stk≥12 * n
  ensures emp ∧ stk′=stk ∧
          heap′=heap+12 * n;
{ if (x ≠ null) {
    dec_stk(4);
    node2 t := x; x := x.next;
    free(t); inc_hp(12);
    dec_stk(8); free_list(x);
    inc_stk(8); inc_stk(4) }
}
```

Fig. 5. Example 2

Note that the memory effect is simulated via explicit calls to the afore-mentioned four primitive methods over `heap` and `stk`, which are highlighted in bold.

As one more example, we show in Figure 6 a program with more complicated memory usage behaviour. The program translates a doubly linked list (`node2`) into a singly linked list (`node`), by deallocating `node2 x` and then creating a singly linked list with the same length and content. A heap memory of $4 * n$ bytes is reclaimed back since each `node2` object has one more field (which takes 4 bytes) than a `node` object.

```
node dl2sl(node2 x)
  requires  x::dll⟨_, n⟩ ∧ stk≥20*n ∧ heap≥0
  ensures   res::ll⟨n⟩ ∧ stk′=stk ∧ heap′=heap+4*n;
{ dec_stk(4); node r := null;
  if (x ≠ null) { dec_stk(4); int v := x.val; dec_stk(4);
      node2 t := x; x := x.next; free(t); inc_hp(12);
      dec_stk(8); r := dl2sl(x); inc_stk(8);
      dec_hp(8); r := new node(v, r); inc_stk(4); inc_stk(4) };
  inc_stk(4); r }
```

Fig. 6. Example 3

The instrumented programs are then passed to HIP/SLEEK for automated verification.

3.2 The Hip/Sleek Automated Verification System

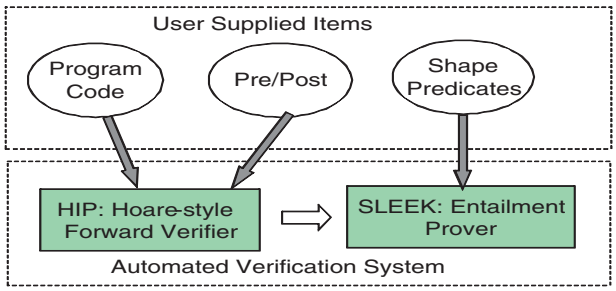

Fig. 7. The HIP/SLEEK Verification System

An overview of the HIP/SLEEK automated verification system is given in Figure 7. The front-end of the system is a standard Hoare-style forward verifier HIP, which invokes the entailment prover SLEEK. The HIP verifier comprises a set of forward verification rules to systematically check that the precondition is satisfied at each call site, and that the declared postcondition is successfully verified (assuming the given precondition) for each method definition. The forward verification rules are of the form $\vdash \{\Delta_1\}\, e\, \{\Delta_2\}$ which expect the symbolic abstract state Δ_1 to be given before computing Δ_2. Given two separation formulas Δ_1 and Δ_2, the entailment prover SLEEK attempts to prove that Δ_1 entails Δ_2; if it succeeds, it returns a frame `R` such that $\Delta_1 \vdash \Delta_2 * \mathtt{R}$. More details of the HIP and SLEEK provers can be found in Chin et al. [10].

4 Soundness

This section presents the soundness of our approach with respect to an underlying operational semantics given in Figure 8. Note that we instrument the state with memory size information, so a program state is represented by $\langle s, h, \sigma, \mu, e\rangle$, where s, h denote respectively the current stack and heap state as mentioned earlier, σ (μ) represents current available stack (heap) memory in bytes, and e is the program code to be executed. If the execution leads to an error, we denote the error state as er_1 if it is due to memory inadequacy, or as er_2 for all other errors (e.g. null pointer dereference). Note also that an intermediate construct $\texttt{ret}(\texttt{v}^*, \texttt{e})$ is introduced to denote the return value of call invocation and local blocks as in Chin et al. [10]. Later, we use $\hookrightarrow^*$ to denote the composition of any non-negative number of transitions, and $\uparrow$ for program divergence.

$$\langle s,h,\sigma,\mu,v\rangle\hookrightarrow\langle s,h,\sigma,\mu,s(v)\rangle \qquad \langle s,h,\sigma,\mu,k\rangle\hookrightarrow\langle s,h,\sigma,\mu,k\rangle$$

$$\langle s,h,\sigma,\mu,v{:=}k\rangle\hookrightarrow\langle s[v{\mapsto}k],h,\sigma,\mu,()\rangle \qquad \langle s,h,\sigma,\mu,();e\rangle\hookrightarrow\langle s,h,\sigma,\mu,e\rangle$$

$$\frac{s(v)\in dom(h)}{\langle s,h,\sigma,\mu,v.f\rangle\hookrightarrow\langle s,h,\sigma,\mu,h(s(v))(f)\rangle} \qquad \frac{s(v)\notin dom(h)}{\langle s,h,\sigma,\mu,v.f\rangle\hookrightarrow er_2}$$

$$\frac{\langle s,h,\sigma,\mu,e_1\rangle\hookrightarrow\langle s_1,h_1,\sigma_1,\mu_1,e_3\rangle}{\langle s,h,\sigma,\mu,e_1;e_2\rangle\hookrightarrow\langle s_1,h_1,\sigma_1,\mu_1,e_3;e_2\rangle} \qquad \frac{\langle s,h,\sigma,\mu,e\rangle\hookrightarrow\langle s_1,h_1,\sigma_1,\mu_1,e_1\rangle}{\langle s,h,\sigma,\mu,v{:=}e\rangle\hookrightarrow\langle s_1,h_1,\sigma_1,\mu_1,v{:=}e_1\rangle}$$

$$\frac{s(v){=}\texttt{true}}{\langle s,h,\sigma,\mu,\texttt{if } v \texttt{ then } e_1 \texttt{ else } e_2\rangle\hookrightarrow\langle s,h,\sigma,\mu,e_1\rangle}$$

$$\frac{s(v){=}\texttt{false}}{\langle s,h,\sigma,\mu,\texttt{if } v \texttt{ then } e_1 \texttt{ else } e_2\rangle\hookrightarrow\langle s,h,\sigma,\mu,e_2\rangle}$$

$$\frac{s(v_1)\in dom(h)\quad r=h(s(v_1))[f{\mapsto}s(v_2)]\quad h_1=h[s(v_1){\mapsto}r]}{\langle s,h,\sigma,\mu,v_1.f:=v_2\rangle\hookrightarrow\langle s,h_1,\sigma,\mu,()\rangle} \qquad \frac{s(v_1)\notin dom(h)}{\langle s,h,\sigma,\mu,v_1.f:=v_2\rangle\hookrightarrow er_2}$$

$$\frac{s(v){\mapsto}l\in h\quad h_1{=}h\backslash[s(v){\mapsto}l]\quad \mu_1{=}\mu{+}ssizeof(type(v))}{\langle s,h,\sigma,\mu,\texttt{free}(v)\rangle\hookrightarrow\langle s,h_1,\sigma,\mu_1,()\rangle} \qquad \frac{s(v)\notin dom(h)}{\langle s,h,\sigma,\mu,\texttt{free}(v)\rangle\hookrightarrow er_2}$$

$$\frac{\begin{array}{c}\texttt{data } c\ \{t_1\ f_1,..,t_n\ f_n\}{\in}P\quad \iota{\notin}dom(h)\\ \mu{\geq}ssizeof(c)\quad \mu_1{=}\mu{-}ssizeof(c)\qquad r{=}c[f_i{\mapsto}s(v_i)]_{i=1}^n\end{array}}{\langle s,h,\sigma,\mu,\texttt{new } c(v^*)\rangle\hookrightarrow\langle s,h{+}[\iota\mapsto r],\sigma,\mu_1,\iota\rangle} \qquad \frac{\mu{<}ssizeof(c)}{\langle s,h,\sigma,\mu,\texttt{new } c(v^*)\rangle\hookrightarrow er_1}$$

$$\langle s,h,\sigma,\mu,\texttt{ret}(v_1,..,v_n,k)\rangle\hookrightarrow\langle s{-}\{v_1,..,v_n\},h,\sigma{+}sizeof(type(v_1),..,type(v_n)),\mu,k\rangle$$

$$\frac{\langle s,h,\sigma,\mu,e\rangle\hookrightarrow\langle s_1,h_1,\sigma_1,\mu_1,e_1\rangle}{\langle s,h,\sigma,\mu,\texttt{ret}(v^*,e)\rangle\hookrightarrow\langle s_1,h_1,\sigma_1,\mu_1,\texttt{ret}(v^*,e_1)\rangle}$$

$$\frac{\sigma{\geq}sizeof(t)\quad \sigma_1{=}\sigma{-}sizeof(t)}{\langle s,h,\sigma,\mu,\{t\ v;\ e\}\rangle\hookrightarrow\langle s{+}[v{\mapsto}\bot],h,\sigma_1,\mu,\texttt{ret}(v,e)\rangle} \qquad \frac{\sigma{<}sizeof(t)}{\langle s,h,\sigma,\mu,\{t\ v;\ e\}\rangle\hookrightarrow er_1}$$

$$\frac{\begin{array}{c}s_1{=}s{+}[w_i{\mapsto}s(v_i)]_{i=m}^n\\ \sigma{\geq}\Sigma_{i=m}^n sizeof(t_i)\quad \sigma_1{=}\sigma{-}\Sigma_{i=m}^n sizeof(t_i)\\ t_0\ mn((\texttt{ref } t_i\ w_i)_{i=1}^{m-1},(t_i\ w_i)_{i=m}^n)\ \{e\}\end{array}}{\begin{array}{l}\langle s,h,\sigma,\mu,mn(v_1,..,v_n)\rangle\hookrightarrow\\ \quad\langle s_1,h,\sigma_1,\mu,\texttt{ret}(\{w_i\}_{i=m}^n,[v_i/w_i]_{i=1}^{m-1}e)\rangle\end{array}} \qquad \frac{\sigma{<}\Sigma_{i=m}^n sizeof(t_i)}{\langle s,h,\sigma,\mu,mn(v^*)\rangle\hookrightarrow er_1}$$

Fig. 8. Underlying Semantics

As shown in the transition rule, a successful execution of $\texttt{free}(v)$ increases the heap size μ by *ssizeof*(*type*(v)). Note that we use $h \setminus [s(v) \mapsto l]$ to erase $s(v)$ from h's domain. The execution of $\texttt{new}\ c(v^*)$ first checks if the current heap space is sufficient for the allocation; if it succeeds, the heap size is decreased by *ssizeof*(c). Here we adds $\iota \mapsto r$ into h by the notation $h + [\iota \mapsto r]$.

The stack space may be changed when the program enters into or exits from a local block $\{t\ v;\ e\}$, or invokes a method, or returns from a method call. Upon exit from a block or a method call, all local variables are popped out from the stack $(s - \{v_1, .., v_n\})$ and the corresponding stack space is recovered $(\sigma + \mathit{sizeof}(\mathit{type}(v_1), .., \mathit{type}(v_n)))$. Conversely, entering a block or invoking a method may require some stack space to store newly declared local variables or returning address of the method. So the relevant semantic rule first checks whether the stack space is sufficient to cater for a new block or a method invocation, if so, the program state is transformed. Otherwise a memory inadequacy error is reported.

Due to the recording of memory size information in program state, we need an extended model to link the underlying semantics with the separation formula, which is defined as follows:

$$s, h, \sigma, \mu \models \Phi \quad =_{def} \quad s, h \models [\sigma/\texttt{stk}', \mu/\texttt{heap}']\Phi$$

where $s, h \models \Phi$ was defined in Chin et al. [10].

Next, we show that the instrumented program $\mathcal{M}(e)$ is observationally equivalent to the original program e w.r.t. the semantics in Figure 8.

Theorem 1 (Observational Equivalence). *For any stack s, heap h, stack size σ, heap size μ, and program e and its instrumented version $\mathcal{M}(e)$, one and only one of the following cases holds:*

1. $\exists s_1, h_1, \sigma_1, \mu_1 \cdot \langle s, h, \sigma, \mu, e\rangle \hookrightarrow^* \langle s_1, h_1, \sigma_1, \mu_1, \nu\rangle \Longleftrightarrow \langle s[\texttt{stk}\mapsto\sigma, \texttt{heap}\mapsto\mu], h, \sigma, \mu, \mathcal{M}(e)\rangle \hookrightarrow^* \langle s_1[\texttt{stk}\mapsto\sigma_1, \texttt{heap}\mapsto\mu_1], h_1, \sigma_1, \mu_1, \nu\rangle$ *where value ν is the evaluation result of e;*
2. $\langle s, h, \sigma, \mu, e\rangle \hookrightarrow^* er_1 \Longleftrightarrow \langle s[\texttt{stk}\mapsto\sigma, \texttt{heap}\mapsto\mu], h, \sigma, \mu, \mathcal{M}(e)\rangle \hookrightarrow^* er_1$*;*
3. $\langle s, h, \sigma, \mu, e\rangle \hookrightarrow^* er_2 \Longleftrightarrow \langle s[\texttt{stk}\mapsto\sigma, \texttt{heap}\mapsto\mu], h, \sigma, \mu, \mathcal{M}(e)\rangle \hookrightarrow^* er_2$*;*
4. $\langle s, h, \sigma, \mu, e\rangle \uparrow \Longleftrightarrow \langle s[\texttt{stk}\mapsto\sigma, \texttt{heap}\mapsto\mu], h, \sigma, \mu, \mathcal{M}(e)\rangle \uparrow$.

Note that the stack mapping $s[\texttt{stk}\mapsto\sigma, \texttt{heap}\mapsto\mu]$ is the same as s except that it maps `stk` *to σ and* `heap` *to μ.*

Proof. By structural induction over e. □

We assume that the global variables, such as `heap` and `stk`, reside in the top frame of the run-time stack when a program starts to run. Note that invocations of the four primitive methods, namely *inc_hp*($\cdot$), *inc_stk*($\cdot$), *dec_hp*($\cdot$) and *dec_stk*($\cdot$), modify only the values of `heap` and `stk`, but not the rest of the stack. Each invocation of these methods requires eight bytes of stack space, which is immediately recovered after the invocation.[3]

[3] Because of this, a memory-aware program may require an additional stack space of 8 bytes. For simplicity, we assume this has been taken into account implicitly.

Finally, the following theorem ensures the soundness of our memory usage verification:

Theorem 2. *For any method* $t\ mn\ ((\texttt{[ref]}\ t\ v)^*)$ *requires* Φ_{pr} *ensures* $\Phi_{po}\ \{e\}$*, if we can verify* $\mathcal{M}(e)$ *against specification* (Φ_{pr}, Φ_{po})*, then we have* $\forall s, h, \sigma, \mu \cdot (s, h, \sigma, \mu \models \Phi_{pr} \wedge \langle s, h, \sigma, \mu, e\rangle \hookrightarrow^* \langle s_1, h_1, \sigma_1, \mu_1, \nu\rangle) \Longrightarrow s_1, h_1, \sigma_1, \mu_1 \models \Phi_{po}$.

Proof. It follows from Theorem 1 and the soundness of the Hip/Sleek verification process given in Chin et al. [10]. □

5 Experimental Results

We have implemented our proposal and integrated it with the Hip/Sleek system to support memory usage verification. We have evaluated the system using a number of benchmarks, by first converting them to memory-aware programs and then passing them to the Hip/Sleek system for memory usage verification (which is done as one pass along with the verification of other safety properties). One set of programs that we have tested are taken from Nguyen et al. [19]. Despite of small-size, these programs are composed of methods manipulating shared mutable data structures, such as (doubly) linked lists, cyclic linked lists, binary search trees, most of which cannot be handled by previous type-based memory usage verifiers. Another set of programs that we have tested are taken from the Olden Benchmark Suite [8]. These programs are of medium-size and quite often contain sophisticated memory usage behaviour. For all programs, we have manually supplied their memory specifications which are precise when validated through some sample runs. The initial experimental results have shown that the memory usage specification is expressive and the memory usage verification via Hip/Sleek is powerful, especially in dealing with mutable data structures with sophisticated sharing.

Programs	**Code (lines)**	**Verified Methods**	**Verification (in sec.)**
Benchmark programs from Nguyen et al. [19]			
singly linked list	72	4/4	0.42
doubly linked list	104	4/4	1.20
binary search tree	62	2/2	0.32
cyclic linked list	78	2/2	0.48
Olden Benchmark suite			
treeadd	195	4/4	0.58
bisort	340	6/6	2.80
em3d	462	20/20	1.52
mst	473	22/22	1.64
tsp	545	9/9	3.44
health	562	15/15	7.35
power	765	19/19	5.17

Fig. 9. Experiment Results

Figure 9 summarises some statistics obtained during the experimental study. The statistics shows that our approach is general enough to handle many interesting data structures such as single linked lists, double linked lists, trees and cyclic linked lists. Column 4 shows the CPU times used (in seconds) for the verification. Our experiments were done under Linux platform on Intel Core Quad 2.66 GHz with 8 GB main memory. All programs take under 10 seconds to verify, even for medium-sized programs with sophisticated memory usage behaviour.

6 Related Work

Previous research on memory usage analysis and verification [1,2,5,7,15] mainly focuses on functional programs where data structures are mostly immutable and easier to deal with. Amadio et al. [1] define a simple stack machine for a first-order functional language and discuss the performance of type, size and termination verifications at bytecode level of the machine. Their contribution is to verify a system of annotations for the bytecode at loading time, and ensure both time and space resource bound required by its execution. Their work only takes into account stack bounds but not heap memory. Another related work is the research in the MRG (Mobile Resource Guarantees) project [2,5], which focuses on building a proof-carrying code system to guarantee that bytecode programs are free from run-time violations of resource bounds. The analysis is developed for a linearly typed bytecode language which is compiled from a first-order functional language, where the bounds are restricted to a linear form.

Hofmann and Jost [15] present a mechanism to obtain linear bounds on the heap space usage of first-order functional programs. It uses an amortised analysis by assigning hypothetical amounts of free space to data structures in proportion to their sizes. The analysis relies on a type system with resource annotations, and takes space reuse by explicit deallocation. With this approach, memory recovery can be supported within each function, but not across functions unless the dead objects are explicitly passed. Their analysis does not consider stack usage and is limited to a linear form without disjunction. Recently, Campbell [7] gives a type-based approach to stack space analysis. It uses the depth of data structures and adds extra structures to typing contexts to describe the form of the bounds. Heap memory is not considered in his work.

Previous works on memory usage verification [11,12,16] for imperative/OO programming languages mainly use type-based approaches. Chin et al. [12] propose a modular memory usage verification system for object-oriented programs. The system can check whether a certain amount of memory is adequate for safe execution of a given program. However, the verification framework requires alias control mechanism to overcome the mutability and sharing problems. Therefore, it can only handle restricted subsets of programs manipulating shared mutable data structures. Recently, Chin et al. [11] propose a memory bound analysis system for low-level programs. The system tries to infer both stack and heap space bounds, using fixpoint analyses for recursive methods and loops. However, the system does not handle shared objects. Hofmann and Jost [16] propose a type-based heap space analysis for Java style OO programs with explicit deallocation.

It uses an amortised analysis, and a potential is assigned to each datum according to its size and layout. Heap memory usage is calculated by an LP-solver based on function inputs during the type inference.

Different from previous works which try to build a memory usage verification system, we re-use a general-purpose verification system Hip/Sleek for memory usage verification, where shape, size and alias information can be readily obtained from the specifications given in separation logic. With this tool, we can verify quite a number of programs that can not be handled by previous approaches, such as doubly linked lists, cyclic linked lists and binary trees.

7 Conclusion

In this paper we have proposed an approach to memory usage verification, by resorting to a general-purpose verification system Hip/Sleek based on separation logic, where memory usage specifications can be depicted using two special variables `heap` and `stk`. Given a program to verify against its memory usage specifications, instead of constructing and implementing verification rules to conduct the verification, we rewrite the program to its memory-aware version where memory usage behaviours are mimicked by explicit operations over variables `heap` and `stk`. The obtained memory-aware program can then be passed to Hip/Sleek for automated verification. Due to the fact that the memory-aware program is observationally equivalent to its original program, the memory safety for the original program follows directly from the memory safety proof of the instrumented program. We have implemented the rewriting process and integrated it with Hip/Sleek. Our initial experimental study shows that we can verify quite a number of programs which can not be handled by previous memory usage verification systems mainly due to the manipulation of sophisticated shared mutable data structures.

As for future work, we aim to automatically infer memory usage specifications, where possible, to reduce the burden on users and also improve the level of automation for memory usage verification. We have just started another EPSRC-funded project aiming to automatically infer method specifications and loop invariants in a combined separation and numerical domain, which would benefit our memory usage analysis and verification.

Acknowledgement. This work was supported in part by the EPSRC projects [EP/E021948/1, EP/G042322/1] and the A*STAR grant R-252-000-233-305.

References

1. Amadio, R.M., Coupet-Grimal, S., Dal Zilio, S., Jakubiec, L.: A Functional Scenario for Bytecode Verification of Resource Bounds. In: Marcinkowski, J., Tarlecki, A. (eds.) CSL 2004. LNCS, vol. 3210, pp. 265–279. Springer, Heidelberg (2004)

2. Aspinall, D., Gilmore, S., Hofmann, M., Sannella, D., Stark, I.: Mobile resource guarantees for smart devices. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 1–26. Springer, Heidelberg (2005)
3. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: Modular automatic assertion checking with separation logic. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 115–137. Springer, Heidelberg (2006)
4. Berdine, J., Cook, B., Distefano, D., O'Hearn, P.W.: Automatic termination proofs for programs with shape-shifting heaps. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 386–400. Springer, Heidelberg (2006)
5. Beringer, L., Hofmann, M., Momigliano, A., Shkaravska, O.: Automatic certification of heap consumption. In: Baader, F., Voronkov, A. (eds.) LPAR 2004. LNCS (LNAI), vol. 3452, pp. 347–362. Springer, Heidelberg (2005)
6. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. In: ACM POPL, pp. 289–300 (2009)
7. Campbell, B.: Amortised memory analysis using the depth of data structures. In: ESOP. LNCS, vol. 5502, pp. 190–204. Springer, Heidelberg (2009)
8. Carlisle, M.C., Rogers, A.: Software caching and computation migration in Olden. ACM SIGPLAN Notices 30(8), 29–38 (1995)
9. Chin, W.-N., David, C., Nguyen, H.H., Qin, S.: Enhancing modular oo verification with separation logic. In: ACM POPL, pp. 87–99 (2008)
10. Chin, W.-N., David, C., Nguyen, H.H., Qin, S.: Automated verification of shape, size and bag properties via user-defined predicates in separation logic. Under Consideration by Science of Computer Programming (2009), `http://www.dur.ac.uk/shengchao.qin/papers/SCP-draft.pdf`
11. Chin, W.-N., Nguyen, H.H., Popeea, C., Qin, S.: Analysing memory resource bounds for low-level programs. In: International Symposium on Memory Management (ISMM), pp. 151–160. ACM Press, New York (2008)
12. Chin, W.-N., Nguyen, H.H., Qin, S., Rinard, M.: Memory usage verification for oo Programs. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 70–86. Springer, Heidelberg (2005)
13. Distefano, D., O'Hearn, P.W., Yang, H.: A local shape analysis based on separation logic. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 287–302. Springer, Heidelberg (2006)
14. Distefano, D., Parkinson, M.J.: jStar: towards practical verification for Java. In: ACM OOPSLA, pp. 213–226 (2008)
15. Hofmann, M., Jost, S.: Static prediction of heap space usage for first order functional programs. In: ACM POPL, January 2003, pp. 185–197 (2003)
16. Hofmann, M., Jost, S.: Type-based amortised heap-space analysis. In: Sestoft, P. (ed.) ESOP 2006. LNCS, vol. 3924, pp. 22–37. Springer, Heidelberg (2006)
17. Ishtiaq, S., O'Hearn, P.W.: BI as an assertion language for mutable data structures. In: ACM POPL, January 2001, pp. 14–26 (2001)
18. Nguyen, H.H., Chin, W.-N.: Enhancing program verification with lemmas. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 355–369. Springer, Heidelberg (2008)
19. Nguyen, H.H., David, C., Qin, S., Chin, W.-N.: Automated verification of shape and size properties via separation logic. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 251–266. Springer, Heidelberg (2007)
20. Parkinson, M.J., Bierman, G.M.: Separation logic, abstraction and inheritance. In: ACM POPL, pp. 75–86 (2008)

21. Reeves, G., Neilson, T., Litwin, T.: Mars exploration rover spirit vehicle anomaly report. Jet Propulsion Laboratory Document No. D-22919 (July 2004)
22. Reynolds, J.: Separation logic: a logic for shared mutable data structures. In: IEEE LICS, July 2002, pp. 55–74 (2002)
23. Xi, H.: Imperative programming with dependent types. In: IEEE LICS, June 2000, pp. 375–387 (2000)
24. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W.: Scalable shape analysis for systems code. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 385–398. Springer, Heidelberg (2008)